Universität der Bundeswehr München

A Pattern-based Approach for the Combination of Different Layout Algorithms in Diagram Editors

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Sonja Maier
im Juni 2012

Vorsitzender der Kommission:	Prof. Dr. Peter Hertling
1. Berichterstatter:	Prof. Dr.-Ing. Mark Minas
2. Berichterstatter:	Prof. Paolo Bottoni
1. Prüfer:	Prof. Dr. Gunnar Teege
2. Prüfer:	Prof. Dr.-Ing. Wolfgang Reinhardt

Tag der mündlichen Prüfung: 24. September 2012

Universität der Bundeswehr München
Fakultät für Informatik

Abstract

Nowadays, visual languages are widely used. Examples of visual languages are graphs and class diagrams. The usage of meta-tools minimizes the effort that is needed for the creation of visual language editors. By the help of an abstract specification, a large amount of the functionality of an editor may be described. Unfortunately, in most commonly used meta-tools, the definition of the layout behavior is only insufficiently supported. In order to be able to develop a fully functional visual language editor, which lets the editor user draw comprehensible and visual appealing diagrams, this part is of great importance.

In this thesis, a *pattern-based layout approach* for the specification of layout behavior is described. The approach is based on meta-models and enables the combination of all commonly used approaches for the definition of layout behavior. Amongst others, graph drawing algorithms and constraint-based approaches can be used. Furthermore, a newly developed rule-based approach can easily be integrated.

Layout patterns are the main concept of the approach: Each layout pattern encapsulates certain layout behavior. Several layout patterns may be applied to a diagram simultaneously, even to diagram parts that overlap. A *control algorithm* that is included in the approach deals with such situations.

One import characteristic of the approach is that the layout is continuously maintained during diagram modification, and that it is updated at runtime. The possibility to reuse layout patterns, and to integrate them in a huge variety of different visual language editors are two more characteristics.

Based on the layout approach, several layout features were developed: *User-controlled layout behavior* allows the editor user to influence the layout at runtime by applying layout patterns to certain parts of the diagram. For instance, the editor user can align components horizontally by applying the horizontal alignment pattern. This layout behavior is preserved until the editor user explicitly removes it again. The editor is capable of suggesting layout patterns being applied to certain parts of the diagram. This feature is called *layout suggestions*. For instance, the editor suggests to apply the horizontal

alignment pattern to components that are almost horizontally aligned. Furthermore, the editor can even automatically apply these suggestions. This feature is named *ad-hoc layout*.
In this thesis, the practical relevance of the introduced approach is demonstrated: Several layout patterns are specified and used to define the layout behavior of four DiaMeta editors, namely a graph editor, a class diagram editor, a GUI forms editor, and a VEX diagram editor. Additionally, they are used to define the layout behavior of one GEF editor, namely a graph editor.

Contents

Chapter 1

Introduction

"A picture is worth a thousand words." This sentence dates back to Ivan Turgenev, who states in his book *Fathers and Sons* [110]: "A picture shows me at a glance what it takes dozens of pages of a book to expound."
Following this principle, diagrams are used almost everywhere. In computer science, for instance, UML diagrams are used for the specification, the construction and the documentation of software parts.
Diagrams can either be drawn with the help of pen and paper, or they can be created with the help of a computer. Here, we distinguish software that solely allows to draw diagrams, and software that not only allows to draw diagrams, but also provides further functionality, such as syntax-directed editing.
Diagrams are an abstract representation of information [118]: The structure of a diagram usually has meaning, while the layout of a diagram, i.e. the shape and the arrangement of components, usually has no meaning. Instead, the user may define the layout in order to emphasize certain aspects and (or) to improve readability.
Some visual languages have particular drawing conventions, e.g. generalization in class diagrams is usually drawn from top to bottom. Others allow the user to freely shape and arrange components, e.g. nodes in a graph are usually positioned arbitrarily. Therefore, for some types of diagrams, the layout is typically created automatically. For other types of diagrams, an *automatic layout* only partially makes sense. For those, a *user-controlled layout* is indispensable.
In most visual language editors, users are forced to manually maintain the layout desired. As an alternative, the layout engine could maintain the layout automatically, which means that so-called *permanent layout* is supported.
In order to enable automatic layout, user-controlled layout and permanent layout, a diagram editor is needed that incorporates a powerful and flexible layout engine. In an interactive environment, performance is usually the

limiting factor. Throughout the last years, computers got more and more powerful, which clears the way for a powerful and flexible layout engine.
In this thesis, an approach for layout computation is introduced that is specifically tailored to an interactive environment, such as visual language editors. With the help of this approach, the user is able to create a diagram editor and its layout engine with small effort.
This chapter is structured as follows: The context of this thesis is outlined in Section 1.1. An overview of the proposed approach is given in Section 1.2. The scientific contributions of this thesis are summarized in Section 1.3, and an outline of it is given in Section 1.4.

1.1 Context of the Work

The approach for layout computation presented in this thesis is tailored to the interactive nature of visual language editors. It is designed for metamodel based visual language editors, whose syntax is defined by the help of EMF [102], as it is done, for instance, in DiaMeta [86, 87] editors and in GEF [30] editors.

1.1.1 Visual Language Editors

As already stated, visual languages are used almost everywhere. Some examples of visual languages are graphs, class diagrams, GUI forms and VEX [20] diagrams. These visual languages will serve as the running examples throughout this thesis.

Abstract and Concrete Syntax

The core of a visual language is its abstract and concrete syntax specification. Literature often provides different understandings of the terms *abstract syntax* and *concrete syntax*. In this thesis, these terms are used as follows: The abstract syntax (AS) of a visual language describes the "underlying" structure of a diagram of the visual language. The concrete syntax (CS) describes the visual appearance of the visual language elements.
For instance, the abstract syntax describes a graph as a set of nodes and a set of directed edges, where each edge exactly connects two nodes. As can be seen in Figure 1.1, the graph consists of the nodes A, B, C, D and E, and the four edges that connect the nodes A and B, A and C, B and D and B and E. In contrast, the concrete syntax of a visual language describes graphs as circles, whose center is at a certain (x, y)-position (e.g. $(0, 0)$),

and arrows, whose start point and end point are at certain (x, y)-positions (e.g.$[(0, 0), (-1, 1)]$). As can be seen in Figure 1.1, the graph consists of circles at the positions $(0, 0)$, $(-1, 1)$, $(1, 1)$, $(-2, 2)$ and $(0, 2)$, and arrows at the positions $[(0, 0), (-1, 1)]$, $[(0, 0), (1, 1)]$, $[(-1, 1), (-2, 2)]$ and $[(-1, 1), (0, 2)]$.

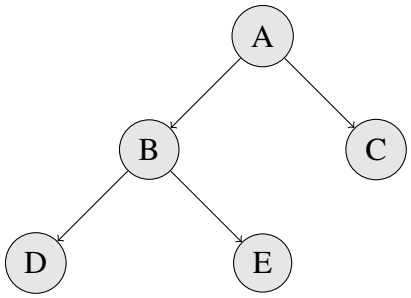


Figure 1.1: Visual Language: Graphs

One might ask, whether or not one of the equations $AS \subseteq CS$ or $CS \subseteq AS$ holds. In general, none of these equations must hold. The concrete syntax may contain some information that is not part of the abstract syntax, for example, information about the position or size of components. The abstract syntax may contain some information that is not visualized, for instance, in case of class diagrams, program code that is associated with a method.

Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) [102] is a modeling framework that provides a code generation facility: From a model specification, code is generated, which comprises a set of classes that enable creating and editing model instances. EMF comprises a meta-model, called Ecore, for describing the models and it comprises the runtime support for the models. In this thesis, meta-models are described by the help of Ecore. Hence, Ecore serves as a meta-meta-model in this case.

DiaMeta

DiaGen and DiaMeta [86, 87] are frameworks for prototyping visual language editors. In DiaMeta, the abstract syntax of a visual language is defined with Ecore, whereas in DiaGen [85], it is defined via a grammar. The rest of the editor is specified by the help of the DiaGen-DiaMeta framework. One characteristic of DiaGen and DiaMeta is that they support structured editing as well as freehand editing. Structured editors offer the user editing operations

that transform correct diagrams into other correct diagrams. Freehand editors, on the other side, allow to arrange diagram components on the screen without any restrictions.

Graphical Editing Framework (GEF)

The Graphical Editing Framework (GEF) [30] is a framework that enables the creation of graphical editors, such as visual language editors, for the Eclipse platform. The abstract syntax of a visual language of such an editor can, for instance, be defined with EMF.

1.1.2 Layout in Visual Language Editors

Literature often provides different understandings of the term *layout*. In this thesis it is defined as follows: Layout comprises all facets of the shape and the arrangement of diagram components. This also comprises the shape and the arrangement of a single component. More precisely, the layout of a component is defined by a set of attributes and their corresponding values. E.g. a node is defined by its (x, y)-position and by its radius. The layout of a diagram is defined by the layout of all components, the diagram consists of, and hence, a set of attributes and their corresponding values.
Updating the layout of a diagram usually implies the change of the concrete syntax of the diagram, only. It should not imply the change of the abstract syntax.

Layout in a Static and a Dynamic Context

Layout computation in a static context means that a diagram is visualized, often without having any previous layout information available. Layout computation in a static context is usually less time-critical than in a dynamic one. In contrast, layout computation in a dynamic context usually means that a user changes a diagram, and afterwards the layout engine updates the diagram. The layout engine should take previous layout information into account, and should present the result immediately. Hence, performance is a big issue.
Automatic graph layout dominates the research in the area of layout. As a consequence, graph aesthetics [109] are usually the main measure for the quality of the layout, and hence, for the quality of the layout algorithm. Examples of aesthetic criteria are symmetry and edge crossing minimization. The overall goal of these criteria is that the layout of a diagram maximally

supports the user to understand and remember the information that is visualized [114].
Graph aesthetics are a good measure in a static context, but not in a dynamic one, such as in graph editors or visual language editors. Here, other criteria should be used, such as mental map preservation [16]: When looking at a drawing, the user has to build a mental map. This essentially means that he or she has to understand the structure and the meaning of the diagram. In a dynamic context, the diagram as well as the layout of a diagram change over time, and the user has to update his or her mental map. The user can be supported in two ways: Layout changes can be minimized, or they can be highlighted and animated.
In summary, in a static context, the following is very important:

- Graph aesthetics and other aesthetics criteria.

In a dynamic context, the following is important:

- Performance in terms of response time of the system.
- Mental map preservation, which means that the user can follow the changes performed in the diagram.

Layout Modifications

Layout algorithms are used for the definition (implementation) of layout behavior. Examples are graph drawing algorithms, such as the Sugiyama algorithm [109]. Layout algorithms change the layout of a diagram in a sense that attribute values are changed. One can distinguish layout algorithms that imply "big changes", and layout algorithms that imply rather "small changes".
Usually, only a small number of layout algorithms, which imply "big changes" in the diagram, are applied simultaneously to a diagram. Examples are graph drawing algorithms that rearrange the whole diagram.
Usually, a high number of layout algorithms, which imply rather "small changes" in the diagram, are applied simultaneously to a diagram. Examples would be an algorithm that aligns components vertically or an algorithm that takes care of the correct connection of edges. This kind of layout behavior is usually neglected by editor developers, and often hand-coded and hard-wired in the specific system.

Standardization and Formalization of Layout

There is a lack of standardization and formalization concerning the layout functionality of visual language editors. As a consequence, each editor developer is challenged by the layout engine, and often neglects this part of the editor. It is a common fact that layout is a demanding task. Nevertheless, developers tend to reinvent the wheel, meaning that the layout engine is built from scratch for every tool. To avoid this needless effort, a framework is needed, which enables and simplifies the reuse of certain layout behavior.

Layout Approaches

Several layout approaches that can be used in a diagram editor exist. As experience shows, developers tend to use only one layout approach and utilize it for the specification of every layout behavior they want. Each layout approach has its own strengths, and therefore a combination of different approaches is more reasonable than the usage of only one of them.
Most tools either solely provide automatic layout, which may not be influenced by the user, or only provide a quite restricted form of user-controlled layout. In contrast, we are convinced that tools should provide the possibility to allow the user to strongly influence the layout at runtime.

1.2 Overview of the Proposed Approach

In the following, an overview of the pattern-based layout approach, which is introduced in this thesis, is given. The cornerstones of the approach are the concept of layout patterns and an algorithm that controls the combination of different layout patterns. Based on the layout approach, several layout features were developed. The most important ones are user-controlled layout behavior, layout suggestions and ad-hoc layout. In [72], an overview of the layout approach and its integration into an editor is given.
The purpose of a visual language editor is to draw diagrams. As shown in Figure 1.2, internally, a diagram is represented by a language-specific model (LM). This language-specific model is an instance of a language-specific metamodel (LMM), which defines the abstract and concrete syntax of the visual language. The current layout of a diagram is represented by the LM, whereas the layout behavior of a diagram editor is defined by a set of layout pattern instances. Each layout pattern, in turn, encapsulates certain layout behavior. Editing a diagram comprises the creation and deletion of diagram components. It also comprises the modification of the layout of a diagram, e.g. the editor user can move or reshape diagram components. As visualized in

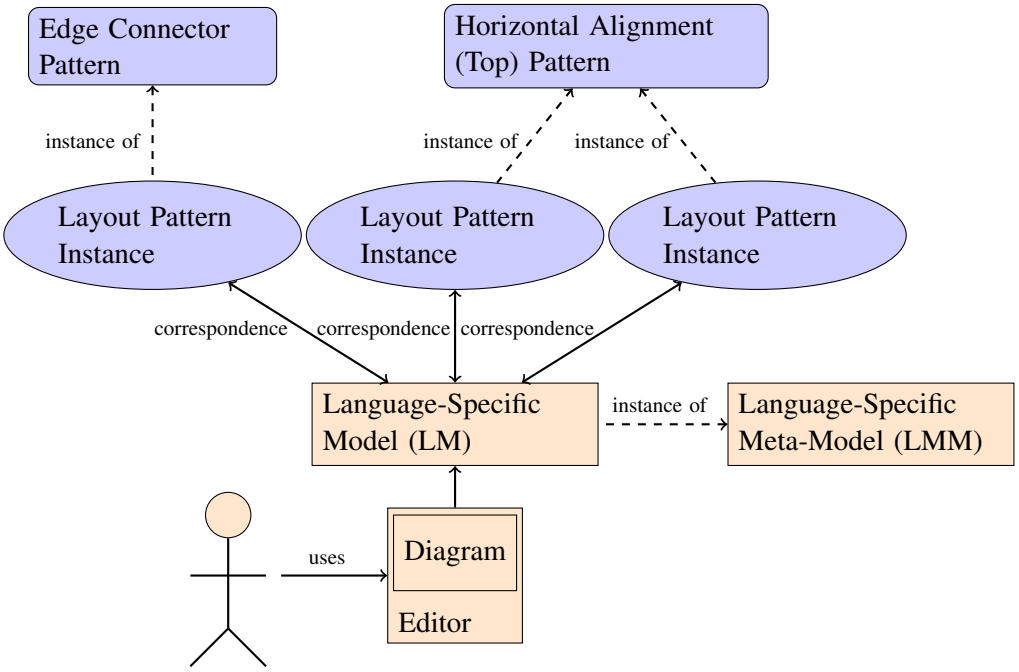


Figure 1.2: Overview of the Approach

Figure 1.2, after the user has edited a diagram, the LM that corresponds to this diagram is automatically created (updated). Furthermore, several layout pattern instances are automatically created (updated). Based on the LM and the layout pattern instances, the new layout of the diagram is automatically computed, and the diagram is updated accordingly. For instance, if the editor user moves circle A in the diagram shown in Figure 1.3, the four outgoing arrows are automatically updated.

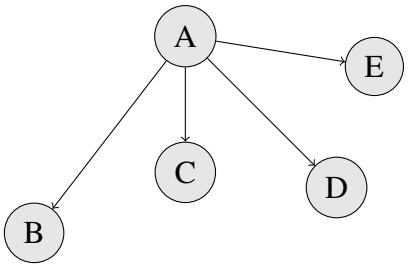


Figure 1.3: Graphs: Example Diagram

1.2.1 Concept of Layout Patterns

A layout pattern encapsulates certain layout behavior. An example is the edge connector pattern, which ensures that edges are correctly connected to nodes. Another example is the horizontal alignment (top) pattern, which makes sure that a certain set of nodes is horizontally aligned at the top.
After the user has edited a diagram, several pattern instances are created, and a correspondence between the LM and each of these pattern instances is established (cf. Figure 1.2).
A layout pattern may be instantiated one or more times for the same diagram. The instantiation of a layout pattern is either performed automatically, or it is triggered by the user. For instance, the edge connector pattern is instantiated automatically. For the diagram shown in Figure 1.3, one pattern instance is created for the nodes A, B, C, D and E together with the four edges. In contrast, the instantiation of the horizontal alignment (top) pattern is triggered by the user. For the diagram shown in Figure 1.3, for instance, two pattern instances could be created - one for the nodes A and E, and one for the nodes B, C and D. The edge connector pattern instance and the two horizontal alignment (top) pattern instances are shown in Figure 1.2.
A layout pattern (cf. Figure 1.4) is defined on top of a pattern-specific meta-model (PMM). It has one or more associated p-constraints (pattern-constraints), and each p-constraint consists of one predicate, which has one or more associated rules. The predicates of all pattern instances present in a diagram assure the layout of the diagram. The layout of a diagram is correct if all predicates hold. The layout of a diagram is incorrect and needs to be updated if one or more predicates are broken. The rules give guidance on how to repair broken predicates.

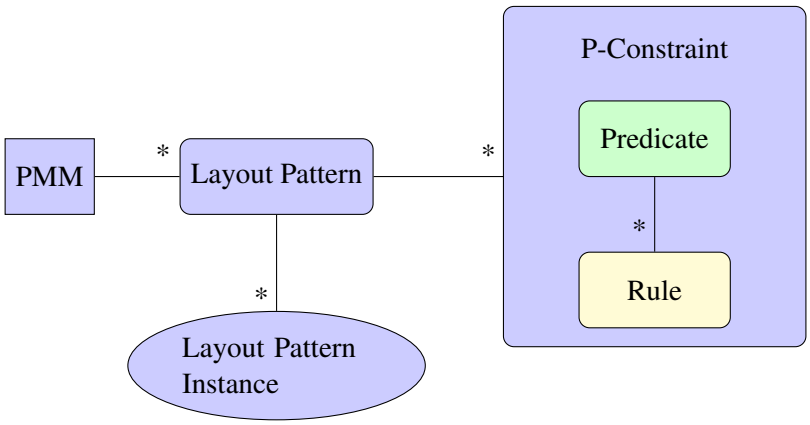


Figure 1.4: Layout Pattern

For instance, the horizontal alignment (top) pattern has the associated predicate $c_1.y = c_2.y$. This predicate has the two associated rules $c_1.y := c_2.y$ and $c_2.y := c_1.y$. c_1 and c_2 are two components. $c_1.y$ is the y-position of the first component, and $c_2.y$ is the y-position of the second component. For the example mentioned above, the predicate as well as the associated rules are "instantiated" several times: For the alignment of the components A and E, they are instantiated for the pair of components $\{A, E\}$. For the alignment of the components B, C and D, they are instantiated for the pairs of components $\{B, C\}$ and $\{C, D\}$.
First ideas of the layout approach are described in [63, 64, 65, 61, 69, 67]. A more detailed description can be found in Chapter 4.

Layout Algorithms

Rules give some guidance on how to repair broken predicates. Each of these rules encapsulates a layout algorithm. The most commonly used types of layout algorithms are graph drawing algorithms, constraint-based (layout) algorithms and rule-based (layout) algorithms. Rule-based algorithms are introduced in this thesis, and are specifically tailored to the interactive nature of diagram editors.
The different types of algorithms and their integration into the pattern-based layout approach are described in [68]. A more detailed description can be found in Chapter 4.

1.2.2 Control Algorithm for Pattern Combination

The control algorithm for pattern combination is essentially a local propagation-based constraint solver that uses backtracking. It gets the set of layout pattern instances that are present in a diagram as input. The purpose of the control algorithm is to find a valid layout after user modification. The algorithm checks all (potentially violated) predicates. For each violated predicate, it applies one associated rule, which in turn triggers the application of the encapsulated layout algorithm. This way, the rule changes one or more attribute values, in order to repair the predicate.
In [70], the control algorithm is described. A more detailed description can be found in Chapter 5.

1.2.3 User-Controlled Layout Behavior

To support the user in an interactive environment, it is not sufficient to apply the same layout patterns in every situation. Instead, the user wants to alter

the layout behavior at runtime. To do so, the editor user has the possibility to manually create and delete pattern instances: He or she may choose a part of the diagram and the layout pattern to be applied to this part. For instance, the editor user may apply the horizontal alignment (top) pattern to the circles B, C and D of Figure 1.3. As a consequence, the three circles are horizontally aligned.
A detailed description of user-controlled layout behavior can be found in Chapter 6.

1.2.4 Layout Suggestions and Ad-hoc Layout

Layout suggestions and ad-hoc layout are two features that are built upon the pattern-based layout approach.
After the editor user selects a certain part of a diagram, the layout engine is able to determine all layout patterns that can potentially be applied to this part. The layout engine can then further compute the attribute changes that become necessary after one of these layout patterns is applied. Based on this information, the layout engine can suggest layout patterns whose application results in small modifications of the layout of the diagram. For instance, the editor user selects the components B, C and D of Figure 1.3. As these components are almost horizontally aligned, the layout engine suggests to apply the horizontal alignment (top) pattern.
Automatic ad-hoc layout goes even one step further: While the editor user edits the diagram, layout patterns whose application results in small modifications of the layout of the diagram are automatically applied. The layout patterns are not only applied to the components selected, but to the components selected together with some other components. For instance, the editor user moves the component B upwards (cf. Figure 1.3). As soon as the components B, C and D are almost horizontally aligned, the layout engine automatically applies the horizontal alignment (top) pattern.
In [73], layout suggestions and ad-hoc layout are described. A more detailed description can be found in Chapter 7.

1.3 Scientific Contributions

This thesis provides the following scientific contributions:

- A pattern-based layout approach has been developed. The approach enables the definition of layout patterns, which are language-independent, and which may be integrated into different visual language editors.

- A control algorithm has been developed, which allows for a flexible combination of different layout patterns.
- Based on the pattern-based layout approach and the control algorithm, several layout features were developed. The most import ones are user-controlled layout behavior, layout suggestions and ad-hoc layout.

The applicability of the presented approach is justified by the following:

- The pattern-based layout approach as well as the layout features were integrated in DiaMeta editors as well as GEF editors.

1.3.1 Research Methodology

The goal of this thesis was the development of a layout approach, which seamlessly integrates with the editor generation framework DiaMeta. In this environment, the layout is updated continuously, beautifying the diagram while the user modifies the diagram. Furthermore, the editor user is able to influence the layout by moving or reshaping components and by applying layout patterns to certain parts of the diagram.
The conceptual design and the implementation of the new layout approach was supported by a user-centered design process. In several iterations, a prototype of the layout framework was created, taking the findings of a user study and the feedback of editor users into account. This iterative process focused on the user requirements and was driven by an evaluation of each iteration.

1.4 Thesis Outline

- Related work is discussed in Chapter 2. Related work concerning visual language editors and generation frameworks for visual language editors is sketched in Section 2.1. An overview of related work in the area of patterns and pattern formalization is given in Section 2.2. In Section 2.3, related work concerning constraints in visual language editors is discussed. Related work concerning layout in visual language editors is described in detail in Section 2.4. Related work in the area of human computer interaction is outlined in Section 2.5.
- In Chapter 3, four visual languages are introduced: Graphs (Section 3.1), class diagrams (Section 3.2), GUI forms (Section 3.3) and VEX diagrams (Section 3.4). In Section 3.5, the layout behavior is described that is included in editors for these four visual languages.

- Chapter 4 introduces layout patterns as means to encapsulate certain layout behavior. Based on this concept, an algorithm automatically computes the layout, as described in Chapter 5. The general concept of the pattern-based layout approach is outlined in Section 4.1. The different meta-models, namely the language-specific meta-model, the pattern-specific meta-models and their correlation are described in Section 4.2. In Section 4.3, the concept of layout patterns is discussed in detail. In Section 4.4, it is explained how the different types of algorithms, namely graph drawing algorithms, constraint-based algorithms and rule-based algorithms, are specified and integrated. In Section 4.5, the concept of atomic layout patterns is introduced.

- In Chapter 5, details are given about the control algorithm whose purpose it is to control the combination of different layout patterns. The general idea of this control algorithm is sketched in Section 5.1. In Section 5.2, some definitions are given. The control algorithm is described in detail in Section 5.3, and two examples are given in Section 5.4. Some characteristics of the algorithm are discussed in Section 5.5. In Section 5.6, future work is outlined.

- The user-controlled instantiation of layout patterns and some special features that are useful in the context of user-controlled instantiation of layout patterns are described in Chapter 6. The automatic and user-controlled instantiation of layout patterns is discussed in Section 6.1. In Section 6.2, two examples of user-controlled instantiation are given. Some special features that are useful in the context of user-controlled instantiation of layout patterns are described in Section 6.3. In Section 6.4, future work is outlined.

- In Chapter 7, the concept of layout suggestions and the concept of ad-hoc layout are described, two features that are based on the idea of user-controlled layout behavior. The concept of layout suggestions is discussed in Section 7.1. In Section 7.2, the concept of ad-hoc layout is described. In Section 7.3, future work is outlined.

- In Chapter 8, several layout patterns are described, and their integration in diagram editors is discussed. In Section 8.1, several layout patterns are described in detail. In Section 8.2, it is discussed how these layout patterns are integrated in the editors described in Chapter 3.

- An evaluation of the approach in terms of usability and in terms of performance is given in Chapter 9. A user study, which aims at identifying layout patterns that are commonly "needed" in diagram editors,

is described in Section 9.1. In Section 9.2, a performance evaluation is presented.

- The thesis is concluded in Chapter 10. A summary is given in Section 10.1. In Section 10.2, some related areas are mentioned, which had a point of contact with this thesis. Some future directions are discussed in Section 10.3.

- In Appendix A, some specification and implementation details are given. Details about the layout framework itself are given in Section A.1. In Section A.2, some details are given about the integration of a layout engine, which is defined by the help of this framework, into DiaMeta editors and into GEF editors.

Chapter 2

Related Work

In this chapter, related work is discussed. Related work concerning visual language editors and generation frameworks for visual language editors is sketched in Section 2.1. An overview of related work in the area of patterns and pattern formalization is given in Section 2.2. In Section 2.3, related work concerning constraints in visual language editors is discussed. Related work concerning layout in visual language editors is described in detail in Section 2.4. Related work in the area of human computer interaction is outlined in Section 2.5.

2.1 Visual Language Editors

The aim of this thesis was to design a layout approach that is tailored to diagram editors (i.e. visual language editors), and also (to some extent) to diagram drawing tools.
A huge variety of diagram editors exist. Examples are graph editors, class diagram editors, GUI forms editors, VEX [20] diagram editors, mind map editors and business process model editors. Examples of diagram drawing tools are commercial tools such as Paint, Powerpoint or Visio.

2.1.1 Meta Tools

Diagram editors provide tool support for modeling techniques. They provide support for editing, simulation, validation, transformation, code generation, and so on. Meta tools provide support for specifying visual modeling techniques. They allow for the generation of visual modeling environments. They also allow for an automated or semi-automated support for developing diagram editors. They provide support for developing editors, simulation tools,

validation tools, transformation tools, code generators, and so on.
One can distinguish three kinds of meta tools: Generic tools, such as Rational Rose, are tools for one visual modeling technique with variants. Frameworks, such as GEF [30], are reusable and incomplete applications that can be completed and customized in order to create specialized tools. Generators, such as DiaGen [85], DiaMeta [86, 87], MetaEdit+ [53] or VL-Eli [98], enable the specification of visual modeling environments, and specific tools may be generated from this specification.
The two most commonly used approaches for the specification of the abstract syntax of a visual language are graph transformation-based approaches, such as used within DiaGen, and meta-model-based approaches, such as used within GEF (in combination with EMF) and DiaMeta. In the first approach, symbols and relations are defined by type graphs, and the abstract syntax is defined by graph grammars. In the second approach, symbols and relations are described by class diagrams, and the abstract syntax is defined by well-formedness rules.

2.1.2 Comparison

The pattern-based layout approach presented in this thesis is integrated within DiaMeta editors and GEF editors, two types of editors that are defined using the meta-model-based approach. DiaMeta editors are built via a generator, whereas GEF editors are built upon a framework.
The meta-model-based approach was chosen, because it is more commonly used than the grammar-based approach. Another criterion for this choice is that EMF can be used for syntax specification in both cases. An editor that is built via a generator as well as an editor that is built upon a framework was chosen in order to cover a wide spectrum of editors.

2.2 Design Patterns

Design patterns [45] serve as the formal basis of the pattern-based layout approach that is described in this thesis. A design pattern is a general reusable solution to a common problem within a certain context. It is not a finished solution that can be transformed directly into code. It is rather a template (or description), which gives guidance on how to solve a problem, and which can be used in many different situations. Hence, design patterns are formalized best practices.
Design patterns have their origins in interaction patterns [111, 108], which were invented by Alexander [1]. Interaction patterns are quite important in

practice today, and huge libraries of interaction patterns exist. Each interaction pattern (design rule) is a small part of the complete (design) knowledge. A "full realization" of interaction patterns as well as of design patterns is enabled by a formalization of the notion of patterns. Several approaches that aim at formalizing interaction patterns as well as design patterns exist. For instance, in [13], a formalization of interaction patterns is detailed, and in [14], a formalization of design patterns is described by the same authors. Both approaches are based on meta-models. Other formalizations are described, for instance, in [57, 76, 75, 24, 38], to name just a few.
In [11, 12], a suite of meta-models is proposed, which describe common types of visual languages. Amongst others, an editor for a certain visual language can be defined by specializing some of these meta-models that are relevant for the visual language.

2.2.1 Pattern-based Model Transformation

In [21], pattern-based model-to-model transformations are presented. The approach is based on patterns which describe positive and negative conditions that have to be satisfied by two models in order to be considered being consistent. This means that transforming the first model results in the second one. Patterns have a high-level semantics that enables the decision of whether or not two models are consistent. The patterns are translated into operational mechanisms, which are based on triple graph grammar rules [99].

2.2.2 Pattern-based Layout

In VL-Eli [98], tree grammars are used as the basis for specifying visual languages and layout. The approach presented in this thesis uses meta-models instead, because they are more widely used today. Similar to the approach presented in this thesis, layout is encapsulated in certain patterns.

2.2.3 Comparison

In this thesis, a new type of patterns, namely layout patterns, are introduced. A layout pattern is essentially a general reusable solution to a common layout problem. This solution is a finished solution, already, which can be directly transformed into code. The formalization of layout patterns is based on meta-models.
The pattern-based layout approach that is described in this thesis requires the transformation between different models. The transformation between these models follows a model-based approach.

2.3 Constraints

Constraints have been used to maintain relationships between components from the very beginning of graphical user interfaces [106]. Constraints are declarative in a sense that they permit the developer to express what they wish to hold true, rather than to describe how to maintain these constraints. Constraints are especially well suited for graph and diagram layout. A thorough overview of the usage of constraints in diagram editors is given in [4]. A constraint system is usually limited by the expressiveness and the performance of the underlying constraint solver. One challenge a constraint solver has to tackle is dealing with under-constrained systems. In such a system, multiple possible solutions exist. In the course of the choice of a solution, the stability of the system has to be maintained: In case a system is under-constrained, and therefore several solutions exist, a solution should be chosen that "minimizes" the changes performed. Constraint hierarchies [9] address this issue, and remove ambiguities by over-constraining the system with constraints at a "lower" level.

2.3.1 Theory of Constraints

Three types of constraint solvers [4] are mainly used in diagram editors: Local propagation-based solvers, iterative numeric solvers and direct numeric solvers.

Local Propagation-based Solvers

Local propagation-based solvers are among the earliest developed solvers, and are quite simple: In a local propagation-based solver, changes are propagated through the diagram. The limitation of propagation-based solvers is their inability to consider more than one constraint at a time. Their strengths are their efficiency and their ability to handle constraints over arbitrary domains. Local propagation-based solvers that handle one-way constraints are distinguished from those that handle multi-way constraints. E.g. the first kind of solver may maintain the constraint $a = 2b + 3c$ by setting a, whereas the second kind of solver may maintain this constraint by setting a, b and (or) c. The second kind of solvers is more powerful, and the underlying algorithms are more complex. Lessons learned from programmer's experiences in the use of one-way constraints are described in [120]: Constraints should be allowed to contain arbitrary code, constraints are difficult to debug and better debugging tools are needed, and developers will use constraints to specify layout behavior, but must be trained to use them in another context. In

[97] and [119], the use of one-way constraints and multi-way constraints in diagram editors is compared. The user studies presented give evidence that both - one-way as well as multi-way constraints - are useful in the context of the definition of layout behavior. They also give evidence that multi-way constraints are more useful than one-way constraints.
There exists a huge variety of local propagation-based solvers. Prominent examples of such constraint solvers that handle multi-way constraints are Blue [40], DeltaBlue [39] and SkyBlue [96], just to mention a few of them. Other examples are Sketchpad [106] and ThingLab [8], two tools that include a local propagation-based solver.

Iterative Numeric Solvers

Iterative numeric solvers have also been used for quite a while. Their strength is that they are very general. Furthermore, they allow solving non-linear constraints simultaneously. Their drawback is that they are relatively slow, and hence, are not especially well suited for an interactive environment.
Behind the scenes, relaxation is used. Relaxation is some sort of iterative "hill climbing" algorithm, and is used to find a local minimum. It does not find a global one.
The tools Sketchpad as well as ThingLab, for instance, use an iterative numeric solver as a fallback technique. This means that if the local propagation-based solver is not successful, an iterative numeric solver is used. Another tool that uses an iterative numeric solver is GLIDE [95].

Direct Numeric Solvers

Direct numeric solvers avoid the problems of iterative numeric solvers. Essentially, direct numeric solvers try to find a solution through direct manipulation of the constraints.
The easiest direct numeric solvers use Gaussian elimination. This type of constraint solver does only find solutions for a certain class of constraint satisfaction problems (CSPs). Furthermore, it only finds a unique solution if the system is not under-constrained. If there are no cycles, propagation can be used instead of Gaussian elimination.
An alternative is the Simplex algorithm. This type of constraint solver also does only find solutions for a certain class of CSPs. It is able to compute a solution for under-constrained systems. An "optimal" solution is found by optimizing a goal function. In a first step, the algorithm computes an initial solution, and in a second step, it determines the optimal one.

Orange [40], Cassowary [10, 5] and QOCA [77] are three prominent examples that use the Simplex algorithm as the underlying concept.

QOCA

QOCA [77] is based on the active set method, which is similar to the Simplex algorithm. It is incremental in a sense that it permits adding and removing constraints while maintaining the solved form. QOCA is also able to handle cycles.

IPSep-CoLa

IPSep-CoLa [26] provides constrained force-directed layout, and is incremental. It has the benefits of force-directed layout methods: Strongly connected components are placed close together, and weekly connected components are separated. The constrained force-directed layout method is some sort of constraint solver, as the approach incorporates so-called separation constraints, which can be used to specify drawing styles and placement relationships.

2.3.2 Constraint-based Tools

In the following, several tools are described that use constraints as the underlying concept for layout computation.

Commercial Tools

Several commercial tools that use constraints for layout computation exist. Examples are Powerpoint, Visio and ConceptDraw. They all support one-way constraints, only, and hence, only provide restricted layout functionalities. For instance, all three tools allow to align components horizontally. In Powerpoint, the components are aligned only once. In Visio and ConceptDraw, persistent layout is realized via guidelines. When the guideline is moved, all components that "belong" to this guideline are also moved. This behavior can be defined via one-way constraints. When one of the components is moved, the other components and the guideline do not move. This is due to the limitations of one-way constraints.

Sketchpad

Sketchpad [106] is the earliest interactive constraint-based system that permits constraints explicitly being specified for the components in the diagram. Sketchpad uses a local propagation-based solver, namely the so-called "one

pass method", whenever possible. If this solver is unable to compute a solution in one pass, it uses an iterative numeric solver instead.

GLIDE

GLIDE [95] is an interactive system for graph layout that uses constraints. It provides some user-controlled layout behavior, for instance, a mechanism for aligning nodes. All constraints available, such as alignment and even spacing, specify local relationships among small groups of nodes. The idea is that the editor user is responsible for a global layout, and the layout algorithm focuses on local optimization. GLIDE uses an iterative numeric solver, which is force-based and which uses a generalized spring model.

Dunnart

Dunnart [118, 29] is an interactive system for drawing graph-based diagrams. It provides some user-controlled layout behavior, and supports permanent layout, which they call persistent in contrast to once-off placement. Dunnart formerly used the constraint solver QOCA [77], but now uses the force-based constraint solver IPSep-CoLa [26] instead. So-called separation constraints can be used to specify layout behavior.

2.3.3 Comparison

For the use in an interactive environment, a constraint solver should be fast, expressive, understandable and reusable. For the design of a constraint solver, the advantages of diagram editors should be taken into account: users are good at direct manipulation and at global search, whereas interactive systems should focus on local layout improvement. Hence, the constraint solver does not need to do everything.
The algorithm used for constraint solving in the approach presented in this thesis is a simple local propagation-based solver. The approach is based on the assumption that the user is responsible for the global layout, whereas the layout engine focuses on local improvement of the layout. A local propagation-based algorithm is well suited for this context.
In contrast to the usage of the constraint solver in the tools described in the last paragraphs, the constraint solver is used on some sort of "meta-level" in the approach described in this thesis.
Further details on rule-based constraint propagation are given in [15], and it is argued that rule-based approaches for constraint propagation are useful

for explanation as well as implementation. This argumentation influenced our design decision.

2.4 Layout in Visual Language Editors

In visual language editors, the layout is usually either defined by graph drawing algorithms, by constraints, or by some sort of self-implemented algorithm. In [25], for instance, some details are given on how to embed graph drawing algorithms into visual language editors.

2.4.1 Graph Drawing Libraries

Graph drawing libraries are usually built for the purpose to create an optimal layout in terms of graph aesthetics. Therefore, they are usually not well suited for the use in an interactive environment: They are far away from optimal in terms of mental map preservation and are unsuitable in terms of performance.
A huge variety of graph drawing libraries exist. Typical types of graph drawing algorithms are layered graph drawing algorithms, force-directed graph drawing algorithms, and orthogonal graph drawing algorithms.
Two libraries that are implemented in Java are, for instance, yFiles [117], a commercial library, and Jung [90], an open source library. Both libraries were included in the approach presented in this thesis, and provide many typical graph drawing algorithms. The yFiles library provides organic layout algorithms, circular layout algorithms, hierarchical layout algorithms, tree layout algorithms and orthogonal layout algorithms. Besides, it provides several incremental layout algorithms and several edge routers. The Jung library provides the Kamada-Kawai algorithm, the Fruchterman-Rheingold algorithm, a simple force-directed layout algorithm, Meyer's Self-Organizing Map layout algorithm, and a circular layout algorithm.
Kieler [43] is a graph drawing library that is tailored to GEF editors, and that is based on meta-models. Their approach focuses on completely automatic diagram layout for specific visual languages. This way, further functionality, such as the layout computation after diagram import or layout computation after structured editing, is enabled.
ZEST [32] is part of the GEF framework and comprises typical graph drawing algorithms, namely a spring layout algorithm, a tree layout algorithm, a radial layout algorithm and a grid layout algorithm. ZEST is usually used for diagram visualization, and is only limited useful in a dynamic context.

Graphviz [49] is a graph visualization software, and comprises a handful of common layout algorithms: a Sugiyama-style hierarchical layout algorithm, the Kamadama-Kawai algorithm and the Fruchtermann-Reingold algorithm for symmetric layouts, a radial layout algorithm described by Wills, and a circular layout algorithm combining the approach described by Six and Tollis and the approach described by Kaufmann and Wiese.

2.4.2 Constraint-based Approaches

A special type of graph drawing algorithms are linear constraints, which provide a declarative approach to layout. For this kind of layout algorithm, a constraint solver is needed in order to compute the layout. Here, the techniques are used that were described in Section 2.3.

2.4.3 Language-specific Diagram Layout

A variety of layout algorithms exist that are tailored to one specific visual language. Most of these special-purpose layout algorithms are designed for completely automatic diagram layout, only.
For instance, special-purpose algorithms for UML diagrams and Euler diagrams [37] exist: In [34], a topology-shapes-metrics approach for the automatic layout of UML class diagrams is described. In [43], automatic layout and structure-based editing of UML diagrams is described. In [101], a layout algorithm is described that is specifically tailored to Euler diagrams.

2.4.4 Comparison

With the pattern-based approach presented in this thesis, the approaches described in the last paragraphs may be combined. This way, the pattern-based approach benefits from the strengths of each of these approaches. Furthermore, the reuse of algorithms is fostered.

2.5 Human Computer Interaction

When creating a layout algorithm for a visual language, some criteria that guide the design decisions need to be identified. These criteria might differ from visual language to visual language. For instance, in [103], Stoerrle presents a user study that discusses the impact of layout quality to understanding UML diagrams. As a second example, in [37], a user study is

presented that gives details on the comprehension of Euler diagrams, which may guide the design of a layout algorithm for Euler diagram editors.
The similarity between a layout that is automatically computed by a layout algorithm and a layout that is manually created by a user can also serve as a quality criterion. In [27], a comparison of user-generated and automatic graph layouts is presented.
In general, graph aesthetics give some guidance for the layout of diagrams in a static environment, whereas mental map preservation gives some guidance for the layout in an interactive one.

2.5.1 Graph Aesthetics

Developers of graph drawing algorithms are usually concerned with creating algorithms that take into account aesthetic criteria. This is because following aesthetic rules is crucial for the understanding of diagrams [109]. Examples of aesthetic rules are few edge crossings, high orthogonality, short edges and few edge segments. The fulfillment of all these aesthetic rules is usually not possible, as some of them contradict each other. Only a few user studies have been performed that give evidence that following these rules leads to more comprehensible diagrams. In [93], Purchase *et al.* show that the aesthetic criteria mentioned above are the most important ones in UML diagrams.

2.5.2 Mental Map Preservation

Dynamic graph layout [16] aims at creating incremental layout techniques that are specifically tailored to an interactive context. The underlying idea is that the user's mental map of the graph is preserved, rather than following aesthetic rules [92].

2.5.3 Comparison

The approach presented in this thesis is a framework that allows for the creation of language-specific layout engines. Its focus lies on mental map preservation rather than following aesthetic rules.

2.6 Summary

In this chapter, related work was discussed. Related work concerning visual language editors and generation frameworks for visual language editors was sketched in Section 2.1. An overview of related work in the area of patterns

and pattern formalization was given in Section 2.2. In Section 2.3, related work concerning constraints in visual language editors was discussed. Related work concerning layout in visual language editors was described in detail in Section 2.4. Related work in the area of human computer interaction was outlined in Section 2.5.

Chapter 3

Running Examples

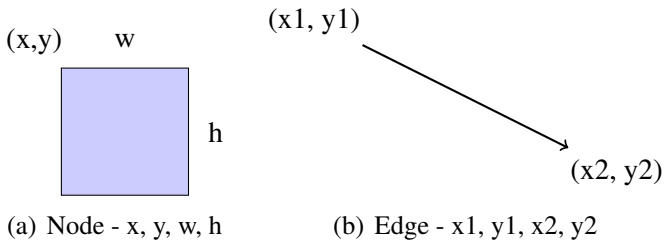
Visual languages can be roughly divided into two different categories: Graph-based, i.e. box-and-arrow-based, and other visual languages, e.g. containment-based, adjacency-based or position-based. Most visual languages combine characteristics of both categories. The approach presented in this thesis supports graph-based and other visual languages and is particularly well suited for visual languages that combine characteristics of both categories.

In this chapter, four visual languages are introduced: Graphs (Section 3.1), class diagrams (Section 3.2), GUI forms (Section 3.3) and VEX diagrams (Section 3.4). In Section 3.5, the layout behavior is described that is included in editors for these four visual languages.

The introduced visual languages will serve as the running examples throughout this thesis. The first two languages, graphs and class diagrams, are mainly graph-based visual languages, whereas the second two languages, GUI forms and VEX diagrams, feature many non-graph-based characteristics.

For each visual language, the language itself is described, and a meta-model is presented, which specifies the abstract syntax of the visual language. This meta-model is called abstract syntax meta-model (ASMM) in the following. It is visualized as a class diagram.

In the following, a diagram component is defined via its attributes. If not mentioned otherwise, a node is defined via the attributes x, y, w and h, as can be seen in Figure 3.1(a). The point with coordinates (x, y) is the top left corner of the component. w is the width of the component, and h is its height. An edge is defined via the attributes $x1$, $y1$, $x2$ and $y2$, as can be seen in Figure 3.1(b). The coordinates $(x1, y1)$ are the starting point of the edge, the coordinates $(x2, y2)$ are its end point.



(a) Node - x, y, w, h (b) Edge - x1, y1, x2, y2

Figure 3.1: Nodes and Edges

3.1 Graphs

The first visual language introduced are graphs. This visual language serves as the main running example throughout this thesis. Although the example is quite simple, most of the layout approach is described with the help of this example. Especially node overlap removal and the integration of graph drawing algorithms will be highlighted with the help of this example.

3.1.1 Language

As can be seen in Figure 3.2, graphs consist of nodes and edges. Nodes are visualized as rounded rectangles, whereas edges are visualized as arrows. In addition, a node may have a name, which is visualized as plain text in the middle of the corresponding node. The example diagram shown consists of four nodes that hold the names A, B, C and D. Besides, the diagram contains an edge that connects the nodes A and C, and an edge that connects the nodes B and C.

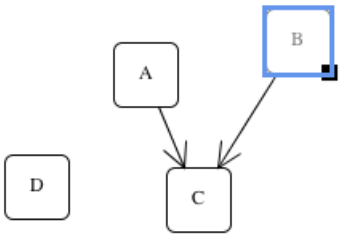


Figure 3.2: Graphs

3.1.2 Meta-Model

The abstract syntax of the visual language is described by a meta-model, which is shown in Figure 3.3. Nodes and edges are represented by the classes `Node` and `Edge`. A node may have a name, which is stored in the attribute `text` of class `Node`. An edge connects two nodes. This connection is represented via the associations `from` and `to`.

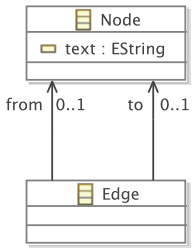


Figure 3.3: Meta-Model of Graphs

3.2 Class Diagrams

Class diagrams are the second visual language that serves as a running example. This visual language is a "real-world" example: The abstract syntax of the visual language is described by a predefined meta-model, the UML2 Ecore model [102]. This meta-model is more or less aligned with OMG's EMOF (Essential MOF), which comprises the essential parts of OMG's MOF [84]. MOF is a so-called closed meta-modeling architecture, which defines the UML. In contrast to the graphs introduced in the last section, in class diagrams, nesting of components is frequently used. Therefore, class diagrams are used to describe aspects of the layout approach that relate to nesting of components.

3.2.1 Language

Class diagrams may contain a huge variety of different components, and therefore, the Ecore model is quite complex. To keep the example simple, only a subset of components is considered, namely packages, classes, attributes, generalizations and associations. As can be seen in Figure 3.4, packages are visualized as boxes with a label at the top. Classes are visualized as boxes with two compartments. The upper compartment contains a label, the lower one may contain a list of attributes. Attributes are visualized

as plain text, consisting of a label and a type, separated by ":". Generalizations are visualized as lines with a triangle at one end and associations are visualized as lines that may have an arrow at no, one or two ends. Additionally, both ends of the line may have a label and a multiplicity. The example diagram shown consists of the packages *pack1* and *pack2*. Furthermore, it contains the classes *A*, *B*, and *C*. Class *A* contains the attributes *attr1* of type *int* and *attr2* of type *String*. The diagram contains a generalization from class *C* to *A* and a unidirectional association from class *B* to class *A* with the role *ref1* and the multiplicity *0..1*.

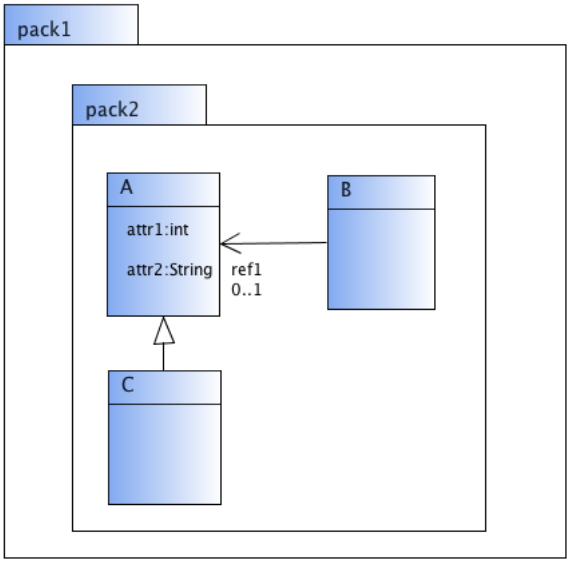


Figure 3.4: Class Diagrams

3.2.2 Meta-Model

The part of the UML2 Ecore model relevant for the example diagrams presented in this thesis is shown in Figure 3.5. As can be seen in Table 3.1, in the meta-model, each component and each correlation between components is either represented by a class or an association in the meta-model.[1]
Types of attributes, as well as multiplicities and roles of associations are also represented in the meta-model, but are not shown in Figure 3.5.

[1] `xxx$yyy` stands for a bidirectional association with the roles `xxx` and `yyy`.

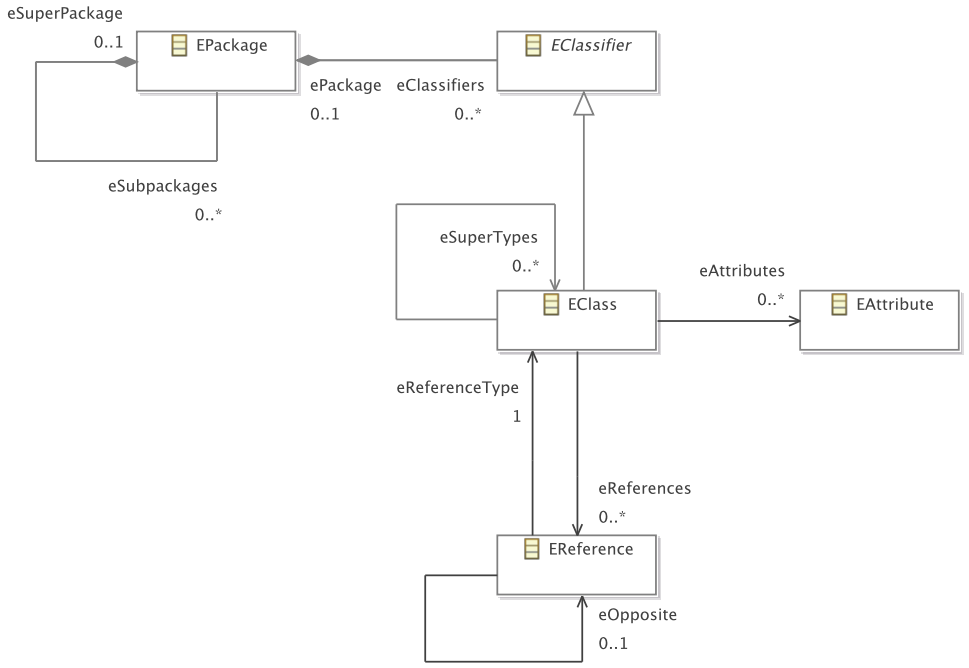


Figure 3.5: Excerpt of the Meta-Model of Class Diagrams

Component	Represented by
packages	class `EPackage`
correlation “package in package”	assoc. `eSuperPackage$eSubpackages`
classes	class `EClass`
correlation “class in package”	association `eClassifiers$ePackage`
attributes	class `EAttribute`
correlation “attribute of class”	association `eAttributes`
associations	class `EReference`
correlation “reference from class”	association `eReferences`
correlation “reference to class”	association `eReferenceType`
bidirectional references	class `EReference` (2x)
	association `eOpposite`
generalizations	association `eSuperTypes`

Table 3.1: Meta-Model of Class Diagrams

3.3 GUI Forms

A somewhat different visual language are GUI forms. With this example, it will be shown that the approach is applicable in very diverse domains. A basic layout engine for this visual language may be created with the approach straightforwardly. A more sophisticated layout engine would also be feasible with this approach, but will not be described in this thesis.

3.3.1 Language

As can be seen in Figure 3.6, GUI forms may consist of frames, panels, buttons, labels, text fields, checkboxes and radio buttons. In contrast to the other components, frames and panels may contain components.

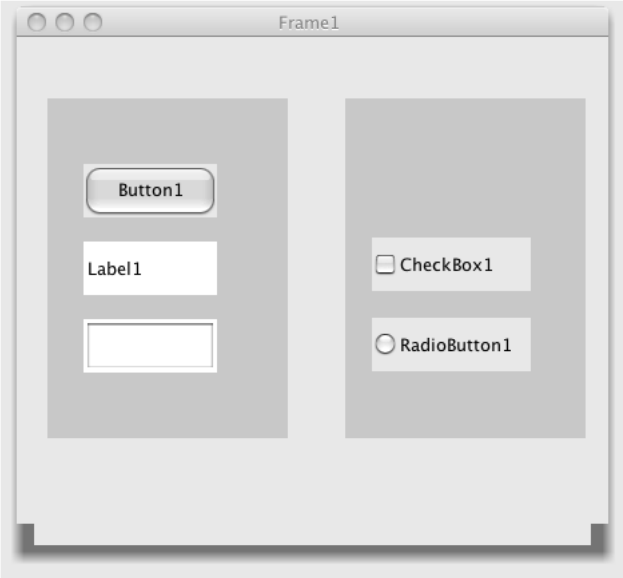


Figure 3.6: GUI Forms

Each component is visualized as some sort of rectangle. In addition, each component may have a label (called text in the meta-model), which is visualized as plain text. Panels and text fields do not have a label. The example diagram shown consists of a frame containing two panels. The left panel contains one button, one label and one text field (from top to bottom). The right panel contains one checkbox and one radio button (from top to bottom). The frame has the label *Frame1*, the button *Button1*, the label *Label1*, the checkbox *CheckBox1*, and the radio button *RadioButton1*.

3.3.2 Meta-Model

The abstract syntax of the visual language is described by a meta-model, which is shown in Figure 3.7. It establishes a tree structure, and therefore mainly consists of the abstract class `AbstractNode` and the two abstract subclasses `SingleNode` and `NestedNode`. The structure of the meta-model follows the composite pattern: The "container" components *Frame* and *Panel* are represented by the abstract class `NestedNode`, whereas the other components, such as *Button* or *Label*, are represented by the abstract class `SingleNode`.

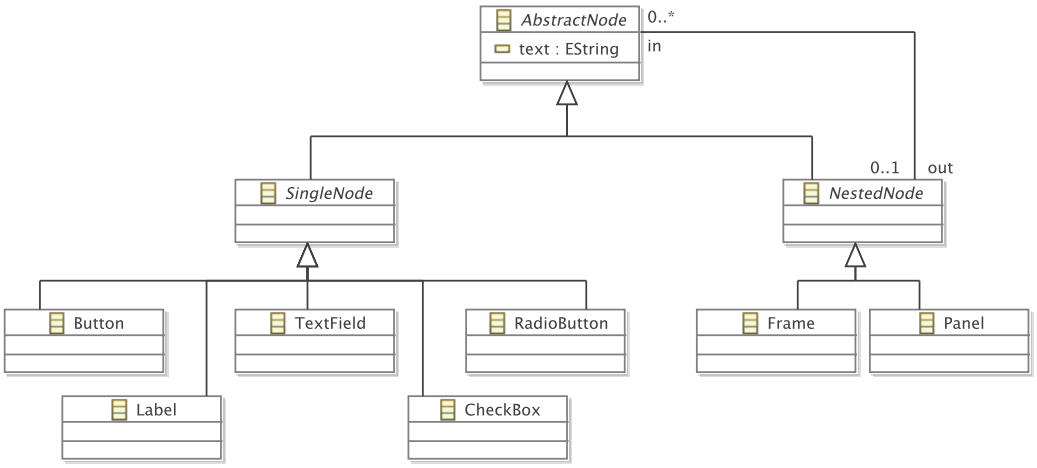


Figure 3.7: Meta-Model of GUI Forms

3.4 VEX Diagrams

VEX diagrams serve as the last running example in this thesis. VEX diagrams are a visual notation of lambda calculus [20]. This visual language has been chosen because it requires a somewhat different and complex layout behavior. Note that a slightly modified variant of this visual language is used here, which can be defined straightforwardly by the help of a meta-model: Each application is surrounded by an additional circle.

3.4.1 Language

Each VEX diagram represents a λ-expression. As can be seen in Figure 3.8, VEX diagrams consist of thin circles (circles with a thin border), bold circles

(circles with a bold border), lines and arrows. A λ-expression is inductively defined. It is either a variable, an application of one λ-expression to another one or an abstraction that consists of the parameter and the abstraction body. For these three types, visual representations are defined:

- In VEX diagrams, free variables are represented by a line, which connects a bold circle, namely the variable, and a thin circle, which again is a λ-expression. On the left side of Figure 3.8, an example is shown: The diagram stands for the λ-expression (x), where x is the name of the variable represented by the bold circle.

- Application is visualized by a thin circle that contains two thin circles. The two contained circles are located next to each other, and are connected by an arrow. The two contained circles are again two λ-expressions. The expression at the end of the arrow represents the function being applied, and the expression at the start represents the argument. In the middle of Figure 3.8, an example is shown: The diagram stands for the λ-expression (xy). x and y are the names of the two variables that are represented by the two bold circles.

- Abstraction is visualized as a thin circle, which contains two thin circles - one attached to the inner side of its border and one located in its center. The two contained circles are connected by a line. The circle located in its center is again a λ-expression. The circle that is attached to the inner side of its border represents the parameter, and the circle that is located in its center represents the abstraction body. On the right side of Figure 3.8, an example is shown: The diagram stands for the λ-expression $\lambda x.(x)$.

Figure 3.9 shows two more complex examples. The first diagram consists of seven thin circles, two lines and one arrow. The diagram stands for the λ-expression $\lambda x.\lambda y.(xy)$. The second diagram also consists of seven thin circles, two lines and one arrow. The diagram stands for the λ-expression $(\lambda x.x)(\lambda y.y)$.

3.4.2 Meta-Model

The abstract syntax of the visual language is described by a meta-model, which is shown in Figure 3.10. A λ-expression is represented by the abstract class `Expr`. A λ-expression may be a free variable (class `FreeVar`), an application (class `Appl`) or an abstraction (class `Abstr`). Free variables are represented by the class `FreeVar`. The association `bound` references

the binding of the variable. Applications are represented by the class `Appl`. The two associations `first` and `second` reference the two λ-expressions, the application consists of. Abstractions are represented by the class `Abstr`. The two associations `param` and `body` reference the parameter and the λ-expression, the abstraction consists of.

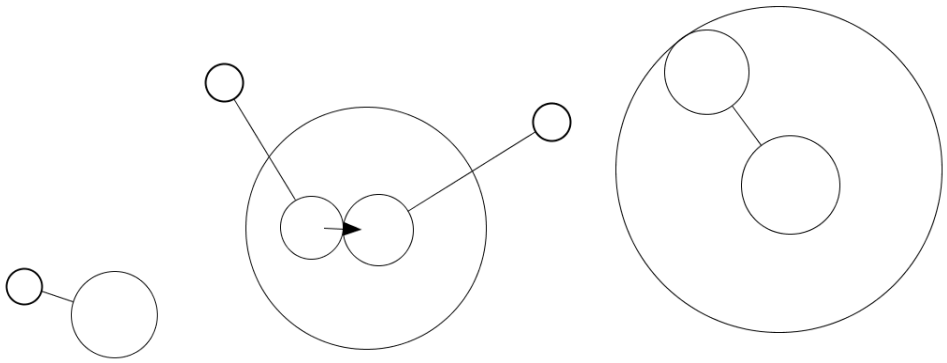


Figure 3.8: VEX Diagrams: Variable Binding, Application and Abstraction

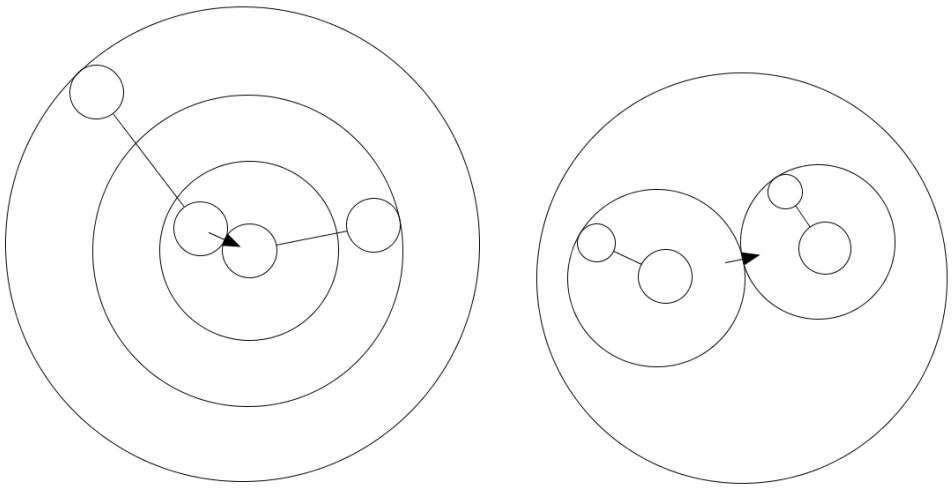


Figure 3.9: VEX Diagrams

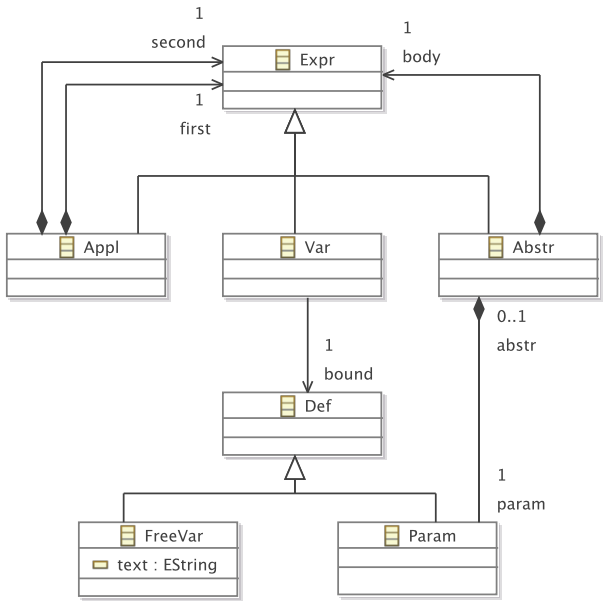


Figure 3.10: Meta-Model of VEX Diagrams

3.5 Layout Behavior

A layout pattern encapsulates certain layout behavior. It defines the layout of a set of components. This also comprises the layout of a single component. The layout patterns, which are included in editors for the four visual languages introduced in the last sections, are described in the following. An overview can be seen in Table 3.2. The description of all layout patterns and their encapsulated layout behavior is based on the graph editor, although not all of them are included in the graph editor.
Visual language editors, such as the ones for the languages described in the last sections, require layout patterns that are tailored to the visual language. E.g. the node overlap removal pattern is used in the graph editor and in the class diagram editor as follows: In the graph editor, nodes should not overlap. In the class diagram editor, some packages and classes should not overlap, while others should overlap, as they should be correctly nested. Nevertheless, some layout patterns can be reused straightforwardly in different editors. E.g. the edge connector pattern is used in the graph editor and in the class diagram editor as follows: In the graph editor, edges should stay attached to their corresponding nodes, and in the class diagram editor, associations and generalizations should stay attached to their corresponding classes.
Many of the layout patterns described in the next paragraphs are available

Pattern	Graph	Class	GUI	VEX
Tree Layout		x		
Layered Layout	x			
Circular Layout	x			
Node Overlap Removal	x	x		
Edge Connector	x	x		x
Equal Horizontal Distance	x	x	x	
Equal Vertical Distance	x	x	x	
Quadratic Component	x	x		
Minimal Size Component	x	x	x	x
Equal Height	x	x	x	
Equal Width	x	x	x	
Align in a Row	x	x	x	
Align in a Column	x	x	x	
Horizontal Alignment	x	x	x	
Vertical Alignment	x	x	x	
List		x	x	
Rectangular Containment		x	x	
Circular Containment				x

Table 3.2: Patterns in Diagram Editors

in different variants. For instance, a spacing between the container and the contained element can be defined for the rectangular containment pattern. Details about variants of layout patterns, and about the integration of patterns into the editors described in the last sections will be given in Chapter 8.

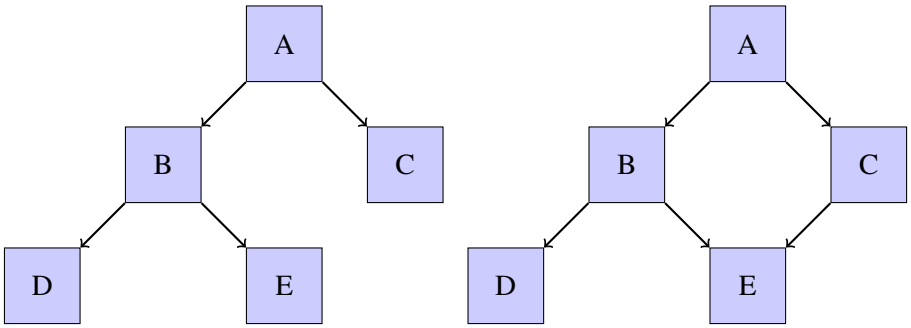
3.5.1 Tree Layout

The first pattern presented, the tree layout pattern, establishes a tree structure in the graph. To do so, it applies a standard graph drawing algorithm, e.g. one of the tree layout algorithms that are included in the yFiles library [117]. The pattern is restricted to trees, and cannot be applied to general graphs. Figure 3.11(a) shows a diagram after applying the pattern.

3.5.2 Layered Layout

The layered layout pattern establishes a layered structure in the graph. To do so, it applies a standard graph drawing algorithm, e.g. one of the hierarchical layout algorithms that are included in the yFiles library [117]. The pattern is quite similar to the tree layout pattern in a sense that nodes are positioned

on one or more horizontal or vertical lines, namely the layers. In contrast to the tree layout pattern, it may be applied to arbitrary graphs. Hence, it is (more or less) a generalized version of the tree layout pattern. Figure 3.11(b) shows a diagram after applying the layered layout pattern.



(a) Tree Layout (b) Layered Layout

Figure 3.11: Tree Layout and Layered Layout

3.5.3 Circular Layout

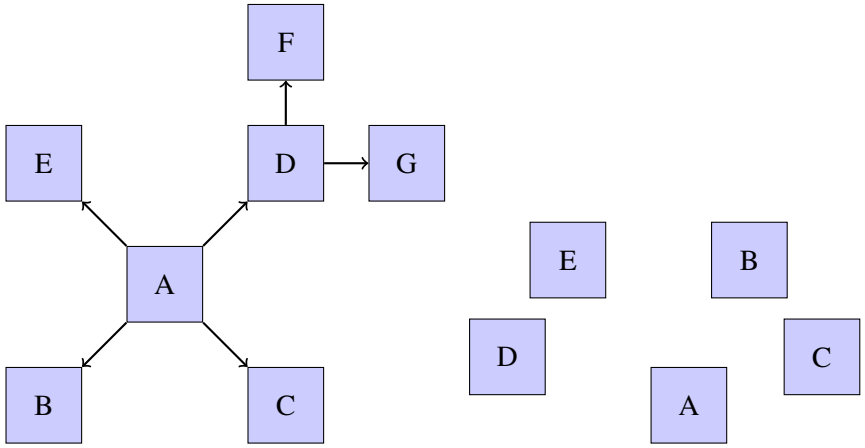
The circular layout pattern is also applicable to arbitrary graphs, and establishes a circular structure in the graph. This means that nodes are positioned on one or more circles. To do so, it applies a standard graph drawing algorithm, e.g. one of the circular layout algorithms that are included in the yFiles library [117]. Figure 3.12(a) shows a diagram after applying the circular layout pattern.

3.5.4 Node Overlap Removal

The node overlap removal pattern is a somewhat different layout pattern. It is applied to nodes only and eliminates all node overlaps. The size and the shape of a component are crucial and need to be taken into account. Node overlap removal is usually achieved by pulling nodes apart. To do so, it applies a force-directed layout algorithm, e.g. one that is included in the Jung library [90]. Figure 3.12(b) shows a diagram after applying the node overlap removal pattern.

3.5.5 Edge Connector

The edge connector pattern ensures that edges are correctly connected to nodes. Figure 3.13 shows a diagram after applying the pattern.



(a) Circular Layout (b) Node Overlap Removal

Figure 3.12: Circular Layout and Node Overlap Removal

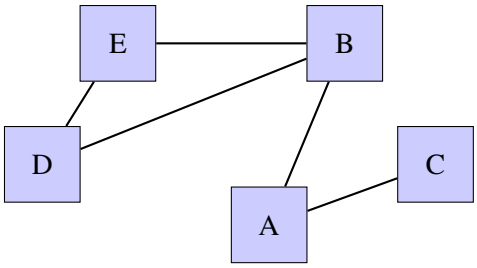


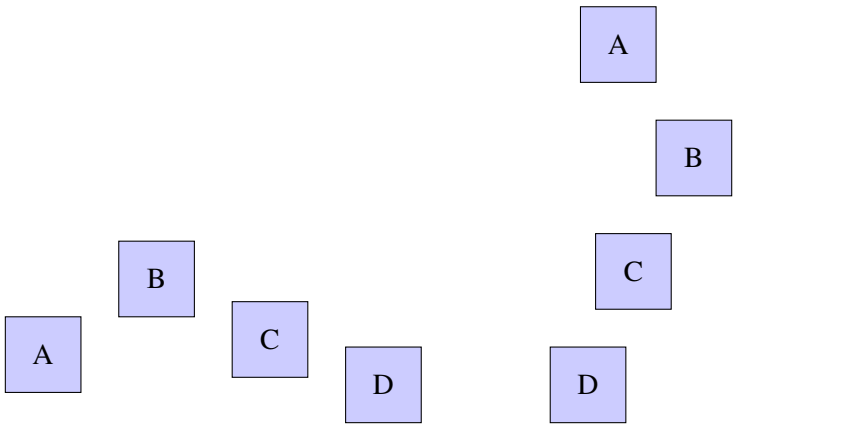
Figure 3.13: Edge Connector

3.5.6 Equal Horizontal Distance

The equal horizontal distance pattern is applied to nodes, and establishes an equal horizontal distance between them. One characteristic of this layout pattern is that the horizontal ordering of nodes is preserved. Figure 3.14(a) shows a diagram after applying the equal horizontal distance pattern.

3.5.7 Equal Vertical Distance

Analogously, the equal vertical distance pattern establishes an equal vertical distance between nodes. One characteristic of this pattern is that the vertical ordering of nodes is preserved. Figure 3.14(b) shows a diagram after applying the equal vertical distance pattern.



(a) Equal Horizontal Distance (b) Equal Vertical Distance

Figure 3.14: Equal Horizontal and Equal Vertical Distance

3.5.8 Quadratic Component

The quadratic component pattern ensures that all nodes are squares. Figure 3.15 shows a diagram after applying the quadratic component pattern. It consists of some nodes of different sizes, whose width and height is equal. This layout pattern is different in a sense that it can be applied to a single component, and its purpose is to change the component itself.

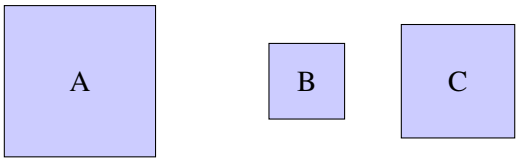


Figure 3.15: Quadratic Component

3.5.9 Minimal Size Component

The minimal size component pattern ensures that nodes are bigger than a certain minimal size. Figure 3.16 shows a diagram after applying the minimal size component pattern. It consists of some nodes of different sizes that are all bigger than a certain minimal size.

Similar to the quadratic component pattern, the pattern can be applied to a single component, and its purpose is to change the component itself.

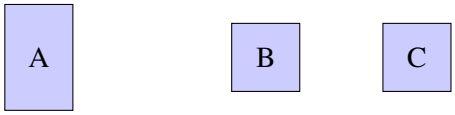


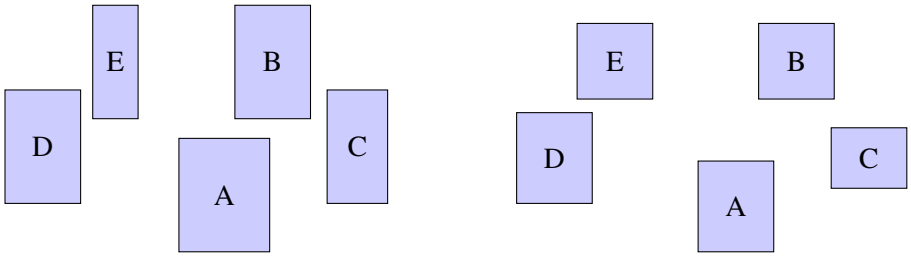
Figure 3.16: Minimal Size Component

3.5.10 Equal Height

The equal height pattern causes all nodes to have the same height. Figure 3.17(a) shows a diagram, in which an equal height of components is established. It consists of some nodes that all have the same height.

3.5.11 Equal Width

The same width of a set of nodes can be achieved by applying the equal width pattern. Figure 3.17(b) shows a diagram, in which an equal width of nodes is established. It consists of some nodes that all have the same width.



(a) Equal Height (b) Equal Width

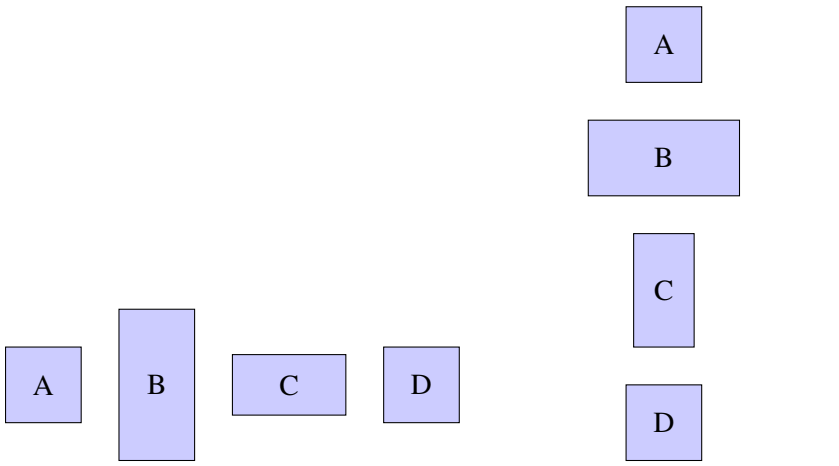
Figure 3.17: Equal Height and Equal Width

3.5.12 Align in a Row

Via the align in a row pattern, nodes are aligned at the top, in the center or at the bottom. Furthermore, nodes have a fixed and equal distance between them. Figure 3.18(a) shows a diagram after applying the pattern. It consists of some nodes that are aligned in the center.

3.5.13 Align in a Column

Via the align in a column pattern, nodes are aligned at the left side, in the center or at the right side. Furthermore, nodes have a fixed and equal distance between them. Figure 3.18(b) shows a diagram after applying the pattern. It consists of some nodes that are aligned in the center.



(a) Align in a Row (b) Align in a Column

Figure 3.18: Align in a Row and Align in a Column

3.5.14 Horizontal Alignment

Nodes are aligned horizontally via the horizontal alignment pattern. It is possible to align nodes at the top, in the center or at the bottom of a component. Figure 3.19(a) shows a diagram after applying the horizontal alignment pattern. It consists of some nodes that are aligned at the top, in the center and at the bottom. This is possible, because all nodes have the same height.

3.5.15 Vertical Alignment

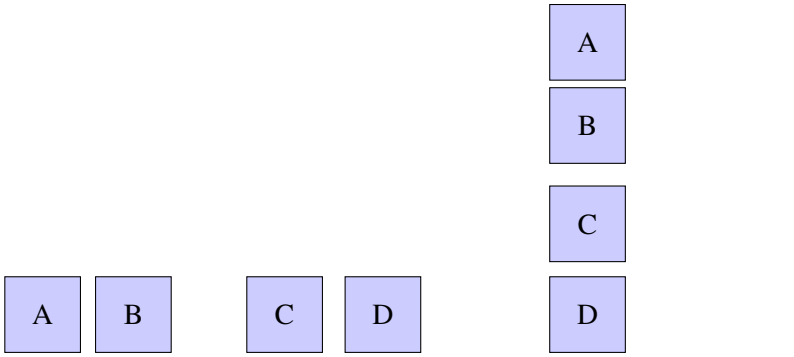
Nodes are aligned vertically via the vertical alignment pattern. It is possible to align nodes at the left side, in the center or at the right side of a component. Figure 3.19(b) shows a diagram after applying the vertical alignment pattern. It consists of some nodes that are aligned at the left side, in the center and at the right side. This is possible, because all nodes have the same width.

3.5.16 List

The list pattern establishes a list structure. All list elements are aligned either vertically or horizontally and have a fixed and equal distance between them. The list pattern is almost equal to the align in a row pattern. The main difference is that a list is always contained in some sort of container. Figure 3.20(a) shows a diagram after applying the list pattern. It consists of some nodes that are arranged as a list, which is aligned vertically.

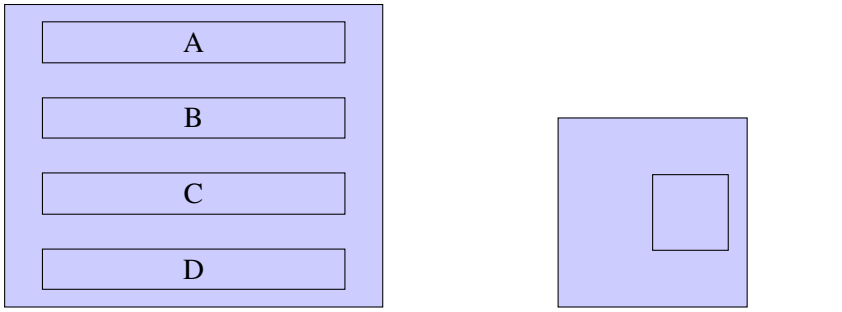
3.5.17 Rectangular Containment

The rectangular containment pattern establishes the correct nesting of nodes with a rectangular shape. Figure 3.20(b) shows a diagram after applying the rectangular containment pattern. It consists of a small node that is contained in a big node. This pattern also allows for an arbitrary nesting of several nodes, meaning that one node may contain more than one node and (or) that a node may contain a node that again contains a node.



(a) Horizontal Alignment (b) Vertical Alignment

Figure 3.19: Horizontal and Vertical Alignment



(a) List (b) Rectangular Containment

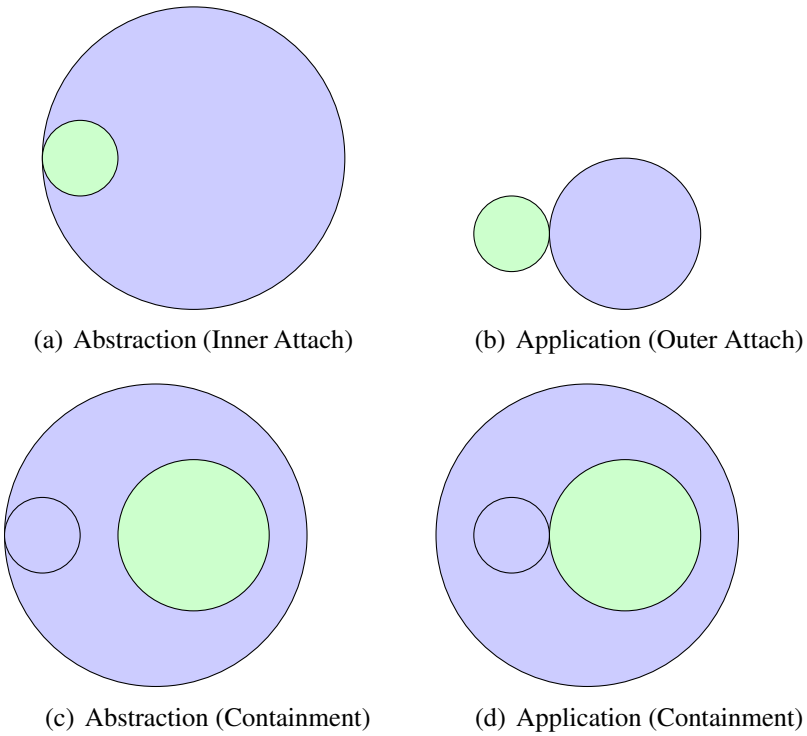
Figure 3.20: List and Rectangular Containment

3.5.18 Circular Containment

The circular containment pattern ensures that the nesting of nodes with a circular shape is correct.
This pattern was specifically designed for the VEX diagram language. Hence, each circle either stands for an application or an abstraction. This means

that it exactly contains two circles. In case of an application, both inner circles may contain other circles. In case of an abstraction, only the inner circle in the middle may contain other circles.
The pattern consists of four different parts. Figure 3.21 shows four diagrams after "applying" one of these parts.

- **Abstraction (Inner Attach)**: A circle is attached to the inner border of a circle (Figure 3.21(a)). The green circle is the one that is attached to the inner border.
- **Application (Outer Attach)**: A circle is attached to the outer border of a circle (Figure 3.21(b)). The green circle is the one that is attached to the outer border.
- **Abstraction (Containment) and Application (Containment)**: One circle is contained in another circle (Figure 3.21(c) and Figure 3.21(d)). In both cases, the green circle is the one contained in another circle. In addition, another circle is attached to the inner border of the outer circle (cf. Figure 3.21(c)), or another circle is attached to the outer border of the inner circle (cf. Figure 3.21(d)).



(a) Abstraction (Inner Attach) (b) Application (Outer Attach)

(c) Abstraction (Containment) (d) Application (Containment)

Figure 3.21: Circular Containment

This layout pattern could also be defined differently, meaning that other parts are defined. The editor developer has to make such design decisions.

3.6 Summary

In this chapter, four visual languages were introduced: Graphs (Section 3.1), class diagrams (Section 3.2), GUI forms (Section 3.3) and VEX diagrams (Section 3.4). In Section 3.5, the layout behavior was described that is included in editors for these four visual languages.
The visual languages introduced in this chapter will serve as the running examples throughout this thesis. A huge variety of other visual languages exist that are also supported by the approach presented in this thesis. Examples are mindmaps, business process models, and class diagrams. These visual languages are explored in a user study in Chapter 9. Other examples of visual languages are entity-relationship diagrams, flowcharts, Nassi Shneiderman diagrams [89], organizational charts, petri nets, reducer rules (a visual language for editor specification [85]), circuit diagrams, and state charts. For all these visual languages, a DiaMeta editor exists. More exotic examples are, for instance, webpages, text (such as newspapers or books) and mathematical formulas.
Most of the layout patterns described in this chapter focus on nodes: They change the position and (or) shape of nodes. Only one layout pattern described focuses on the edges, namely the edge connector pattern, and none focuses on edge labels. It is worth mentioning that the pattern based layout approach, which will be described in Chapter 4, and the control algorithm, which will be presented in Chapter 5, can handle all types of layout patterns.
All layout patterns described in this chapter are suitable for an interactive environment. They take previous layout information, such as the position, shape or size of a component, into account. In addition, most of them perform only minor diagram modifications, and hence mental map preservation is achieved. One example is the equal horizontal distance pattern, as can be seen in Figure 3.14(a): The order of the nodes A, B, C and D is used for the computation of a "correct" layout: In the example diagram shown, for instance, the distance between A and B and between B and C is equal, whereas the distance between A and B and between A and C is not equal.
More details about the specification of each of the layout patterns described in this chapter will be given in Chapter 8.

Chapter 4

Pattern Concept and Reusability

This chapter introduces layout patterns as means to encapsulate certain layout behavior. One or more layout patterns can be applied to one or more parts of a diagram. Each application means that a layout pattern instance is created. The set of instances present in a diagram defines the layout behavior of the diagram. Based on these instances, an algorithm automatically computes the layout, as described in Chapter 5. The first ideas of the pattern-based layout approach are described in [63, 64, 65, 61, 69, 67]. A more detailed description of the approach is given in this chapter.
The chapter is structured as follows: The general concept of the pattern-based layout approach is outlined in Section 4.1. The different meta-models, namely the language-specific meta-model, the pattern-specific meta-models and their correlation are described in Section 4.2. In Section 4.3, the concept of layout patterns is discussed in detail. In Section 4.4, it is explained how the different types of algorithms, namely graph drawing algorithms, constraint-based algorithms and rule-based algorithms, are specified and integrated. In Section 4.5, the concept of atomic layout patterns is introduced.

4.1 General Idea

The core of a visual language editor is the *language-specific meta-model* (LMM). This meta-model contains all information needed for layout computation. It consists of two parts, the abstract syntax meta-model (ASMM) and the concrete syntax meta-model (CSMM). The abstract syntax meta-model defines the abstract syntax of the visual language. For the four running examples, the abstract syntax meta-model was already described in Chapter 3. The concrete syntax meta-model defines all aspects of the concrete syntax of the visual language in focus needed for layout computation.

Layout behavior could directly be defined on the basis of the LMM. This would have the drawback that layout behavior would need to be defined from scratch for every visual language. Furthermore, the definition of layout behavior might not be straightforward, because the information needed for layout computation might not be available directly in the meta-model.
Therefore, a layout pattern is defined on a language-independent, but *pattern-specific meta-model* (PMM), instead of being defined on the LMM. This way, reuse of layout behavior is made possible, and the complexity of the layout specification is usually decreased, as information is directly available. Graph drawing algorithms, constraint-based algorithms and rule-based algorithms are defined on top of these PMMs.
All the described meta-models may be instantiated: The *language-specific model* (LM) is an instance of the LMM, the *abstract syntax model* (ASM) is an instance of the ASMM, the *concrete syntax model* (CSM) is an instance of the CSMM, and the *pattern-specific model* (PM) is an instance of the PMM. In terms of the meta-modeling hierarchy, LMs, ASMs, CSMs and PMs are on the M_1 level. LMMs, ASMMs, CSMMs and PMMs are on the M_2 level. In this thesis, LMs, ASMs, CSMs and PMs are visualized as object diagrams, whereas LMMs, ASMMs, CSMMs and PMMs are visualized as class diagrams.

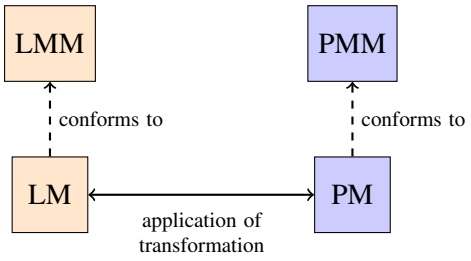


Figure 4.1: LMM, PMM, LM, PM

In order to use a layout pattern for a certain visual language, a correspondence between the language's LM and the pattern's PM must be established. This correspondence identifies the roles that the components of the visual language play in the layout pattern. Following this correspondence, a LM can be transformed into a PM (cf. Figure 4.1). The correlation between the diagram, the language-specific model, the pattern-specific model, and such a *language-independent layout pattern instance* is visualized in Figure 4.2. It is also possible to define layout patterns directly on top of the language-specific meta-model. This option is usually chosen, if an "exotic" layout pattern is required. As will be seen in Chapter 8, this option is needed only in very rare

cases. The correlation between the diagram, the language-specific model and such a *language-dependent layout pattern instance* is visualized in Figure 4.3.

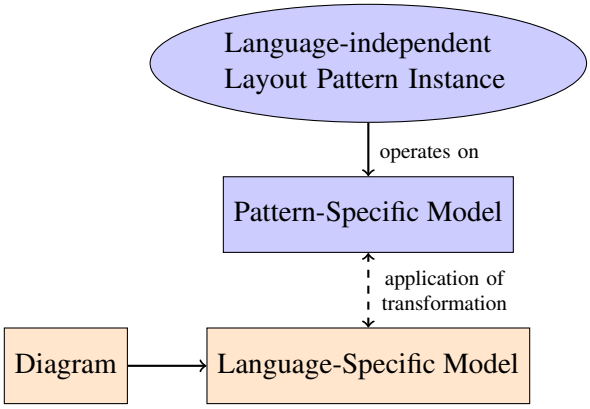


Figure 4.2: Correlation: Diagram, LM, PMs and Language-independent Pattern Instance

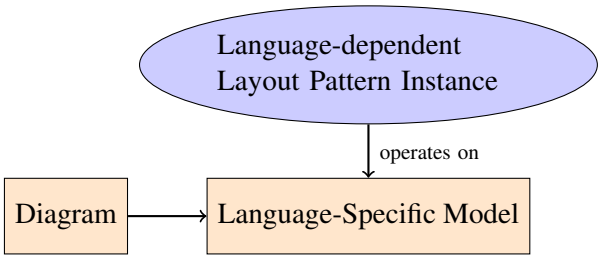


Figure 4.3: Correlation: Diagram, LM and Language-dependent Pattern Instance

4.2 Meta-Models and their Correlation

In the following, the LMM, the PMMs, and the correlation between those meta-models will be described.

4.2.1 Language-Specific Meta-Model

A visual language is defined via its LMM. During layout computation, information about the abstract as well as the concrete syntax is required, and may be accessed via the LMM. In the class diagram editor example, at the abstract syntax level, for instance, information about nesting of packages and classes is directly available. In contrast, information about the visual

appearance of diagram components is only available at the concrete syntax level. E.g. a class is represented by a rectangle with a certain size, and attributes are visualized in a certain order (cf. Figure 4.4).

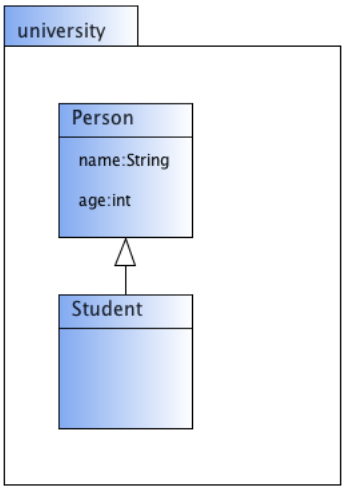


Figure 4.4: Example Class Diagram

Although information about both, the abstract and the concrete syntax is needed for layout computation, and hence, is available in the LMM, a separation between the two parts is preserved. Therefore, the LMM consists of two parts:

- *Abstract syntax meta-model* (ASMM)
- *Concrete syntax meta-model* (CSMM)

The ASMM is exactly the meta-model that specifies the abstract syntax of a visual language, as described in Chapter 3. The CSMM specifies the concrete syntax of a visual language, and contains geometric information that is needed for layout computation. The ASMM and the CSMM are connected via `modelObject` links. This way, the ASMM can simply be reused. Only the CSMM and the `modelObject` links need to be defined, in order to use the layout engine. In the CSMM, each visual component is represented by a class. Variables that define the position or shape of a visual component are represented by its attributes. Correlations between visual components are represented by associations.
An excerpt of the ASMM for class diagrams is shown in Figure 3.5. As already mentioned, it mainly consists of the classes `EPackage`, `EClass`,

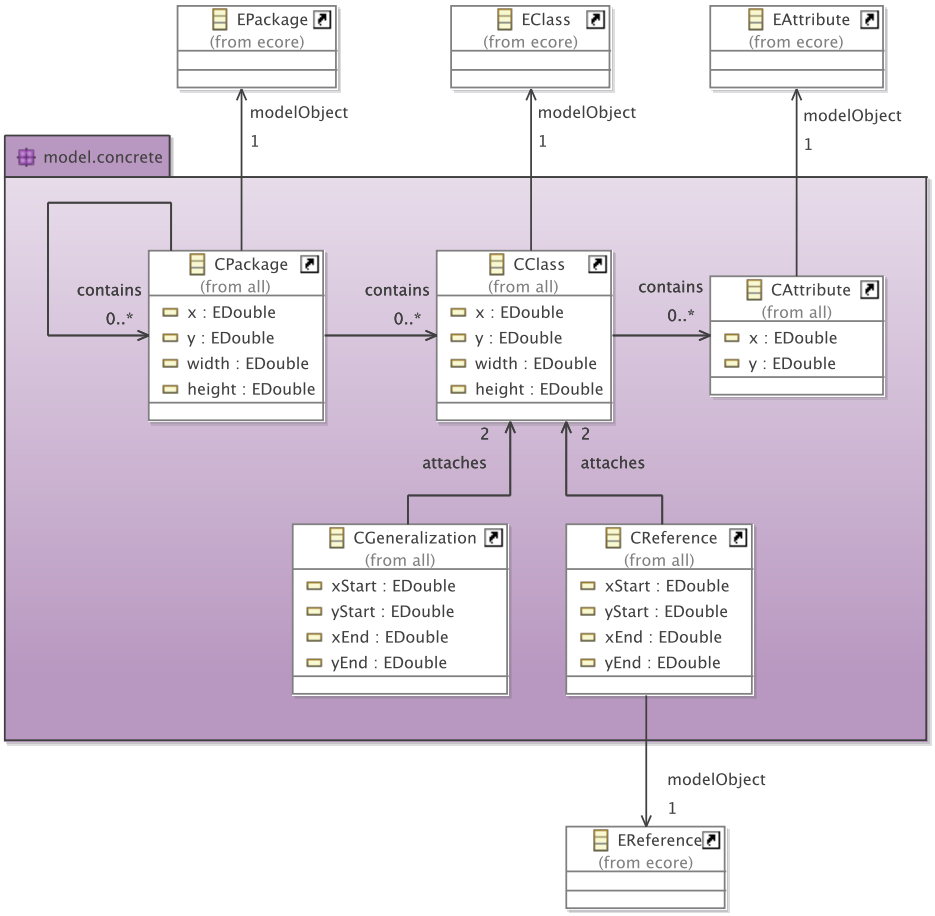


Figure 4.5: CSMM of Class Diagrams and its Connections to the ASMM

`EAttribute` and `EReference` and some associations. The ASMM is equal to the UML2 Ecore specification, as can be found in [102].

An excerpt of the CSMM for class diagrams and its connections to the corresponding ASMM is visualized in Figure 4.5. It consists of the classes `CPackage`, `CClass`, `CAttribute`, `CGeneralization` and `CReference`, which stand for the five different types of components, class diagrams may consist of. Packages, classes and attributes are defined via their `x`- and `y`-coordinates (top-left corner). In addition, packages and classes have a variable size, defined via `width` and `height`. Generalizations and references are defined via their start point (`xStart, yStart`) and end point (`xEnd, yEnd`). Besides, two different associations can be found in the CSMM, namely `attaches` and `contains`. These two associations stand for the two different types of correlations that may be present in class dia-

grams. The correlation `attaches` is used if two components attach each other. In class diagrams, a class may attach generalizations and references. The correlation `contains` is used if one component is inside another component. In class diagrams, a package may contain classes and packages. Furthermore, a class may contain attributes.

The ASMM and the CSMM are connected via `modelObject` links, as can be seen in Figure 4.5. Four `modelObject` links are present, namely between the classes `CPackage` and `EPackage`, the classes `CClass` and `EClass`, the classes `CAttribute` and `EAttribute`, and the classes `CReference` and `EReference`. There exists no `modelObject` link between the classes `CGeneralization` and `EGeneralization`, because the ASMM does not present a class `EGeneralization`. Instead, this concept is defined through the association `eSuperTypes`.

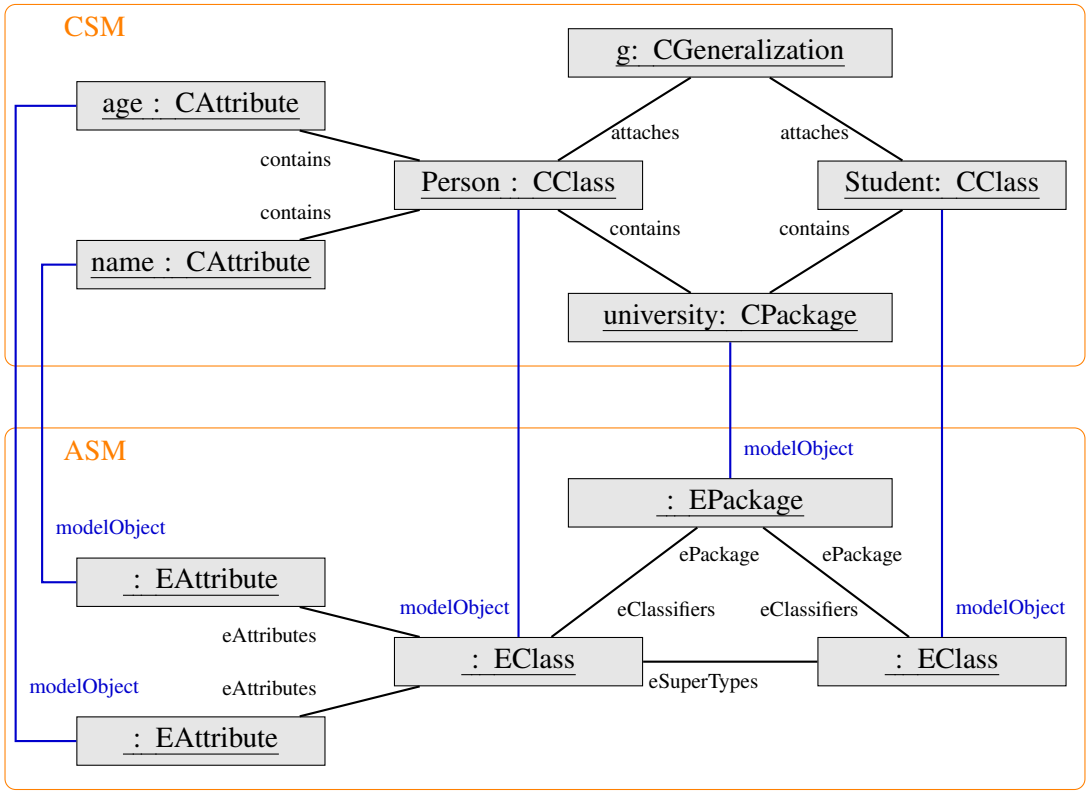


Figure 4.6: ASM and CSM of the Example

Instantiation of LMM

While the editor user interacts with a visual language editor, an instance of the LMM is automatically created and maintained. This instance conforms to the diagram created. E.g. the diagram shown in Figure 4.4 consists of the package `university`, the two classes `Person` and `Student`, the two attributes `name` and `age`, and one generalization. The CSM and the ASM of this diagram are shown in Figure 4.6. The two parts are connected via `modelObject` links, which are colored in blue.
As already mentioned, not every object in the CSM is "connected" to an object in the ASM via a `modelObject` link. In the example, this is the case for the object of type `CGeneralization`.

4.2.2 Pattern-Specific Meta-Models

As mentioned previously, language-independent layout patterns are defined on top of a pattern-specific meta-model, not on the language-specific meta-model. The most commonly used pattern-specific meta-models are shown in Figure 4.7.

Elem PMM

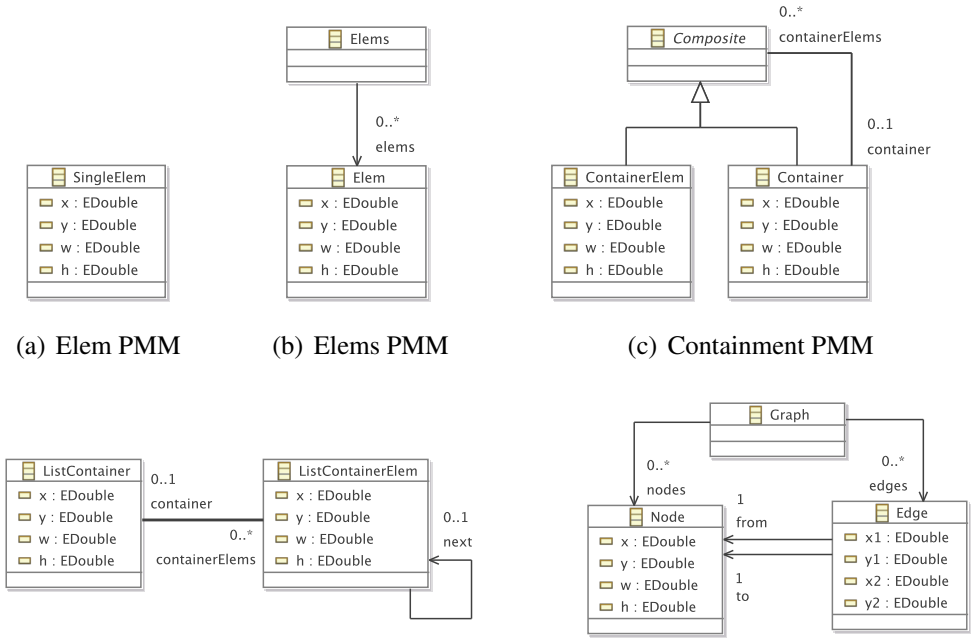
The Elem PMM represents one component, consisting of the class `SingleElem`. An instance of this meta-model consists of one `SingleElem` object.

Elems PMM

The Elems PMM represents a set of components, consisting of the classes `Elems` and `Elem` and the association `elems`. An instance of this meta-model consists of one `Elems` object and an unordered set of `Elem` objects. The `Elems` object is connected to all `Elem` objects via `elems` links.

Containment PMM

The Containment PMM defines a containment structure, consisting of the abstract class `Composite`, the subclasses `ContainerElem` and `Container` and an association with the two roles `containerElems` and `container`. An instance of this meta-model consists of a set of `Container` objects and a set of `ContainerElem` objects. The containment structure is established via `container$containerElems` links.



(a) Elem PMM (b) Elems PMM (c) Containment PMM

(d) List PMM (e) Graph PMM

Figure 4.7: Pattern-Specific Meta-Models (PMMs)

List PMM

The List PMM establishes a list structure, consisting of the classes `ListContainer` and `ListContainerElem`, and the associations `container$containerElems` and `next`. An instance of this meta-model consists of one `ListContainer` object and an ordered set of `ListContainerElem` objects. The `ListContainer` object is connected to all `ListElem` objects via `container$containerElems` links. The order of `ListContainerElem` objects is established via `next` links between them.

Graph PMM

The Graph PMM establishes a graph structure, consisting of the three classes `Graph`, `Node` and `Edge`, and four associations. An instance of this meta-model consists of one `Graph` object, a set of `Node` objects, and a set of `Edge` objects. The `Graph` object is connected to all `Node` objects via `nodes` links, and to all `Edge` objects via `edges` links. Each `Edge` object has one `from` link and one `to` link. Via these links between `Node` objects and `Edge` objects, the graph structure is established.

4.2.3 Correlation of LMM, PMMs and Layout Patterns

Each layout pattern is based on a pattern-specific meta-model. It may be the case that two different layout patterns are based on the same meta-model. Table 4.1 shows a list of the layout patterns that were introduced in Chapter 3, and the pattern-specific meta-model they are based on.

Layout Pattern	Pattern-Specific Meta-Model
Tree Layout Pattern	Graph PMM
Layered Layout Pattern	Graph PMM
Circular Layout Pattern	Graph PMM
Node Overlap Removal Pattern	Graph PMM
Edge Connector Pattern	Graph PMM
Equal Horizontal Distance Pattern	List PMM
Equal Vertical Distance Pattern	List PMM
Quadratic Component Pattern	Elem PMM
Minimal Size Component Pattern	Elem PMM
Equal Height Pattern	Elems PMM
Equal Width Pattern	Elems PMM
Align in a Row Pattern	List PMM
Align in a Column Pattern	List PMM
Horizontal Alignment Pattern	Elems PMM
Vertical Alignment Pattern	Elems PMM
List Pattern	List PMM
Rectangular Containment Pattern	Containment PMM
Circular Containment Pattern	–

Table 4.1: Pattern Meta-Models

Usually, patterns that incorporate only one component are based on the Elem PMM, ones that incorporate an unordered set of components are based on the Elems PMM, and ones that incorporate an ordered set of components are based on the List PMM. Patterns that encapsulate graph drawing algorithms are based on the Graph PMM, and ones that cope with nested components are based on the Containment PMM. The circular containment pattern is based on *no* PMM, as it is a language-dependent layout pattern.

Correspondence between LM and PMs

In order to be able to apply a certain layout pattern, a correspondence between the LM and the PM this layout pattern is defined on, needs to be defined. This correspondence is split into two parts:

- A correspondence between objects and links of the LM and the PM.
- A correspondence between the attributes of the LM and the PM.

Following these correspondences, an LM can be transformed into a PM, and a PM can be transformed into an LM. In the approach presented, this correspondence is used as follows:

- The first correspondence is used to transform objects and links of an LM into objects and links of a PM. Hence, the correspondence is used for a unidirectional transformation, only.
- The second correspondence is used to transform attributes of an LM into attributes of a PM. In addition, it is used to transform attributes of a PM into attributes of an LM. Hence, the correspondence is used for a bidirectional transformation.

This restriction turned out to be reasonable, because the layout engine usually does not change the "structure" of the LM. It does not add or remove objects or links. It usually changes attribute values only.
The transformation between objects and links of the LM and the PM is performed via a model transformation, and will be described in the following. The transformation between attributes of the LM and the PM is performed via constraints of a specific type (so-called mapping constraints), and will be described in detail in Chapter 5.
The first correspondence is defined via *triple graph grammars (TGGs)*[99, 54] in the following. TGGs allow us to define the relation between the LM and the PM and to transform one of these models into the other. The relation between the two models is defined in a declarative way. This definition can then be made operational, and the transformation can be performed.
The relation between objects and links is defined by so-called TGG-rules. E.g. in the class diagram editor example, the horizontal alignment pattern, which is based on the Elems PMM, may be applied to packages and classes. For that purpose, the following relations between objects of the LM and objects of the Elems PM are defined:
The first TGG-rule is shown in Figure 4.8. It defines the relation between a `CPackage` object and an `EPackage` object on one side, and an `Elem` object on the other side. On the left-hand side is the source domain (the LM), on the right-hand side is the target domain (the PM), and in the middle is the part defining the correspondences between the elements of the different models. These objects are called correspondence nodes. The meaning of this TGG-rule is as follows: The black parts, which are not marked with ++, represent

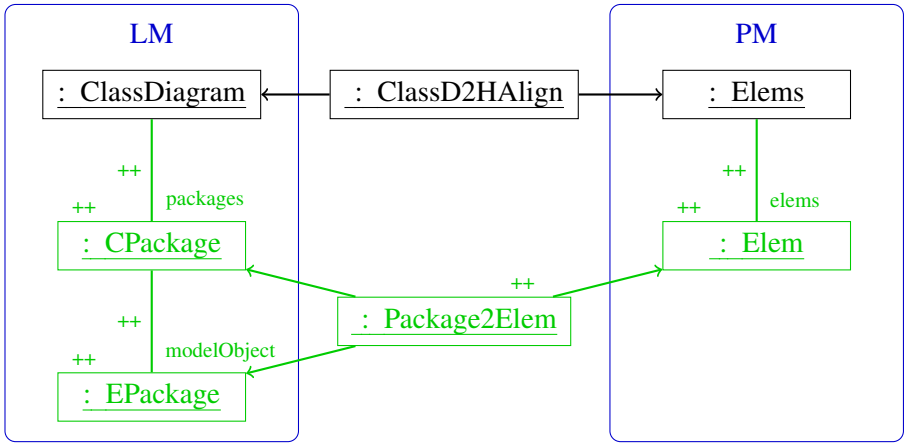


Figure 4.8: TGG-Rule 1

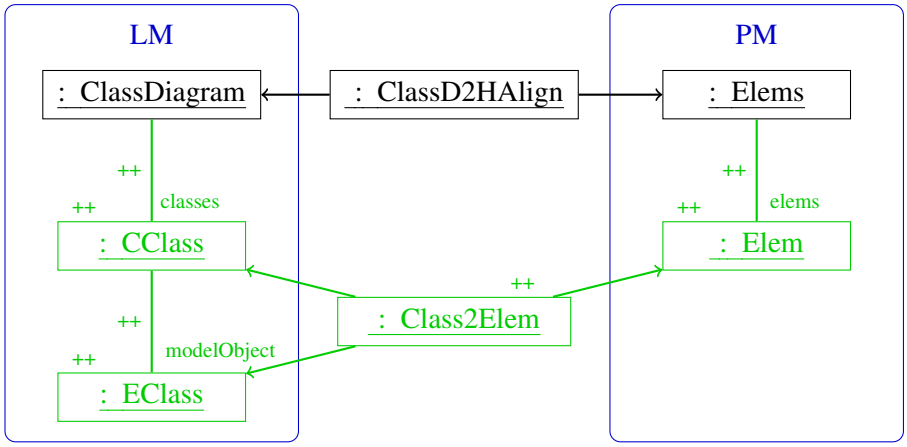


Figure 4.9: TGG-Rule 2

a LMM instance that corresponds to a PMM instance via the correspondence node `ClassD2HAlign`. The green parts, which are marked with ++, insert a `CPackage` object and an `EPackage` object into the LMM instance, as well as the corresponding `Elem` object into the PMM instance. The correspondence node `Package2Elem` keeps track of the correspondence.
The second TGG-rule is shown in Figure 4.9. Analogously to the first rule, it defines the relation between a `CClass` object and an `EClass` object on one side, and an `Elem` object on the other side.
In combination with the two meta-models - the LMM and the PMM - and the meta-model of the correspondence nodes, these TGG-rules define how to construct corresponding pairs of LMM instances and PMM instances. One starts from an "empty" LMM instance corresponding to an "empty" PMM

instance. This is called a TGG-axiom. The first meta-model for the example is shown in Figure 3.5 together with Figure 4.5, and the second meta-model is shown in Figure 4.7(b). The meta-model of the correspondence nodes is shown in Figure 4.11. Note that in the LMM, the class `ClassDiagram` was added, which serves as the "root node" of a class diagram. In the PMM, the class `Elems` serves as the "root node". The TGG-axiom for the example is shown in Figure 4.10.

Figure 4.10: TGG-Axiom

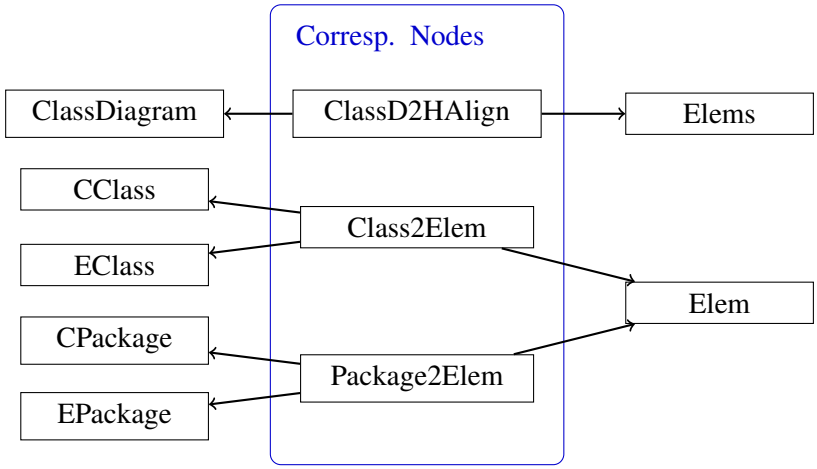


Figure 4.11: Meta-Model of the Correspondence Nodes

In certain situations, it may be the case that more than one correspondence needs to be defined between the LM and one PM, depending on the layout pattern defined on it. E.g. in the class diagram editor example, the layered layout pattern may be applied to classes together with generalizations, whereas the edge connector pattern may be applied to classes together with generalizations *and* associations. For that purpose, two correspondences are defined between the LM and the Graph PM.
To date, the correspondence between objects and links of the LM and the PM is hand-coded. Furthermore, the transformation that is guided by this correspondence is also hand-coded. The support of a transformation language would be reasonable.

Transformation between LM and PMs

When the editor user creates or modifies a diagram, an LMM instance that corresponds to this diagram is automatically created. Furthermore, if one or more pattern instances are present in the diagram, the corresponding PMM instances are automatically created as well. These PMs are created by applying the corresponding transformations. This means that if n pattern instances are present in a diagram, then n transformations are applied and n PMs are created (cf. Figure 4.12).

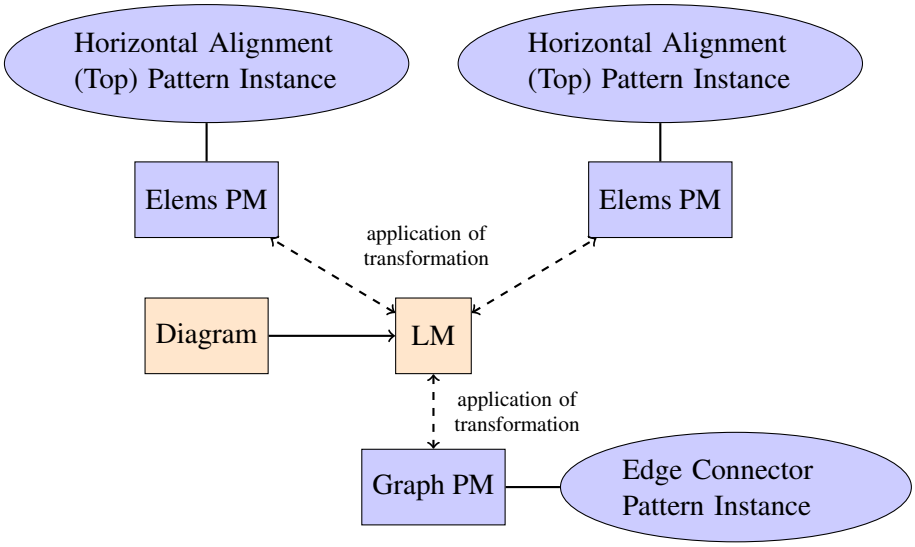


Figure 4.12: Correlation: Diagram, LM, PM and Layout Pattern Instances

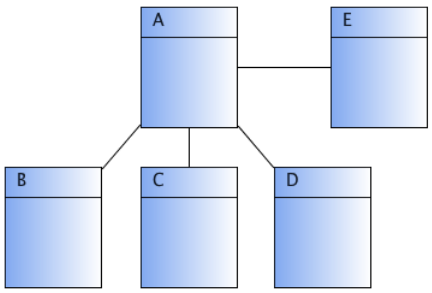
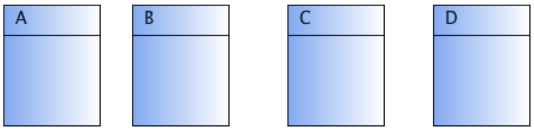


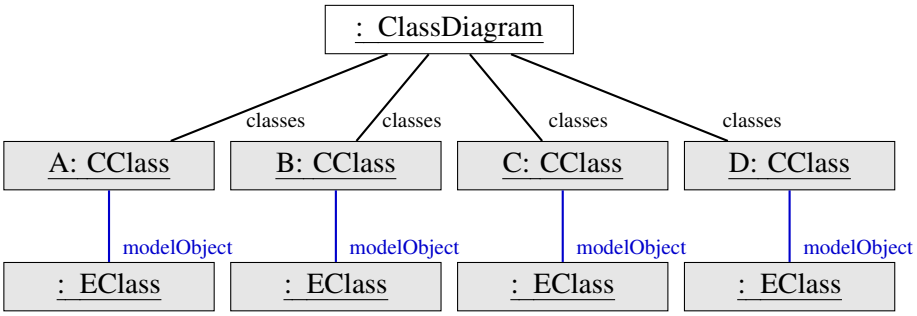
Figure 4.13: Example Class Diagram

In Figure 4.13, a diagram is shown, in which three pattern instances are present: One edge connector pattern instance is created for the five classes

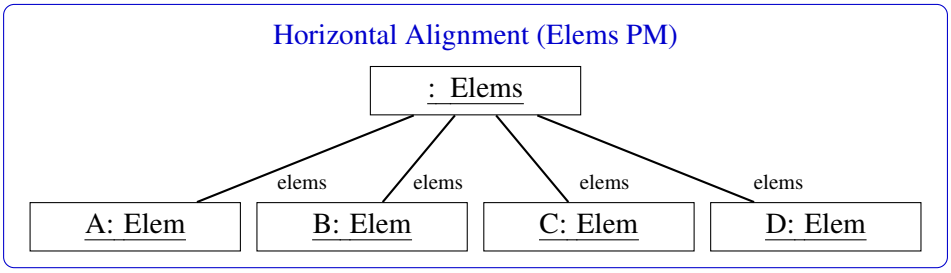
together with the four associations, one horizontal alignment (top) pattern instance is created for the classes A and E, and one horizontal alignment (top) pattern instance is created for the classes B, C and D. In Figure 4.12, the correlation between the diagram, the corresponding LM, the three PMs and the three pattern instances is sketched. The diagram and the LM are visualized as orange rectangles, the PMs as blue rectangles, and the layout pattern instances as blue ovals.



(a) Example Class Diagram



(b) Language-specific Meta-Model Instance



(c) Elems PMM Instance

Figure 4.14: Diagram and Meta-Model Instances

Another example is shown in Figure 4.14. Figure 4.14(a) shows a diagram, which consists of four classes that are aligned horizontally. One layout pattern instance is present in this diagram, namely a horizontal alignment pattern instance that aligns the four classes. The corresponding LM is visualized in Figure 4.14(b), and Figure 4.14(c) shows the corresponding Elems PM that is generated.

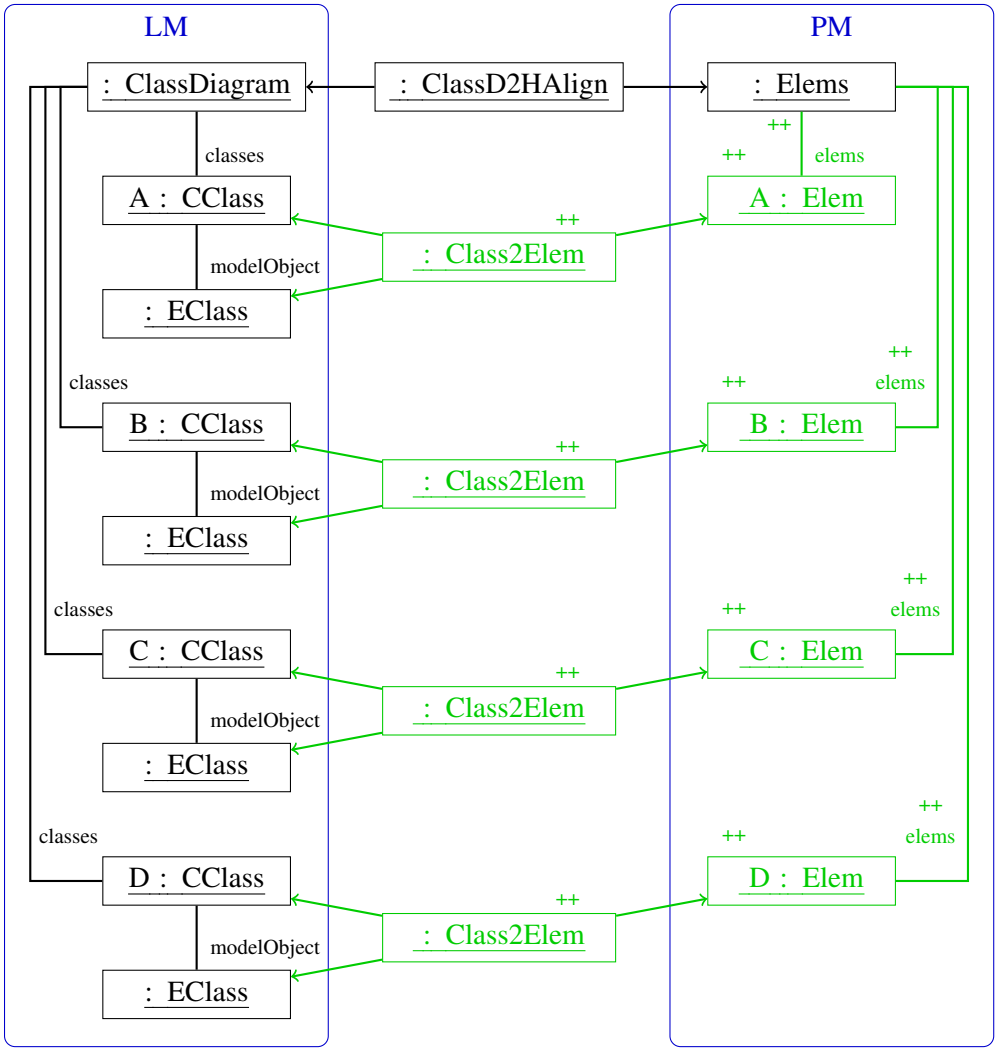
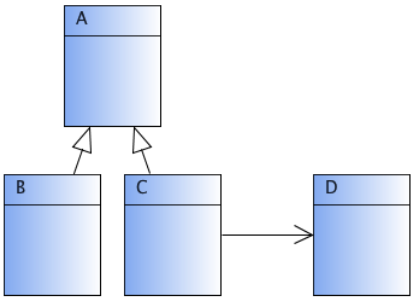


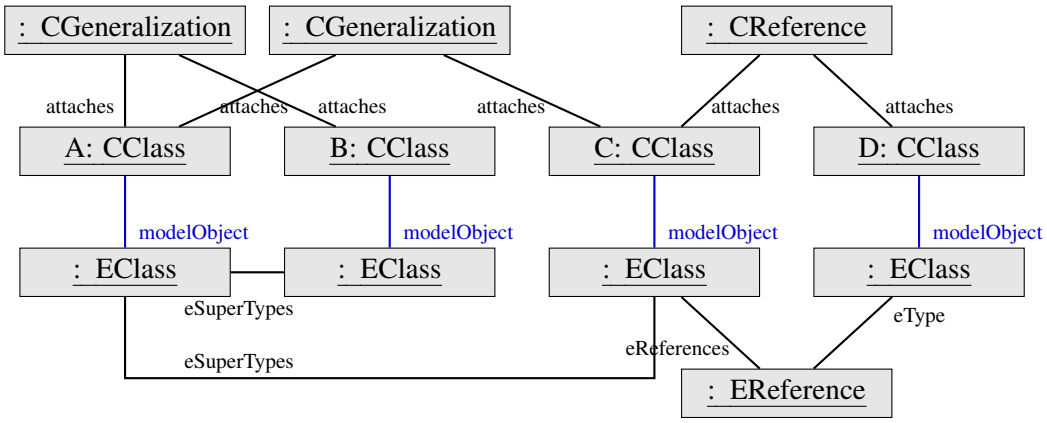
Figure 4.15: Example: After Transformation

As already mentioned, TGGs are used for the transformations between the LM and the PMs. In this scenario, the LM exists already, and the corresponding PMs are generated. This transformation is typically called *forward transformation*. In order to perform one transformation, one starts from the LM. Then, matching the source domain of the rules of the TGG to the existing LM is tried. Thereafter, one adds the missing correspondence objects and the missing objects and links in the PM. Once the LM has been fully matched, the correspondence objects and the PM have been fully generated.

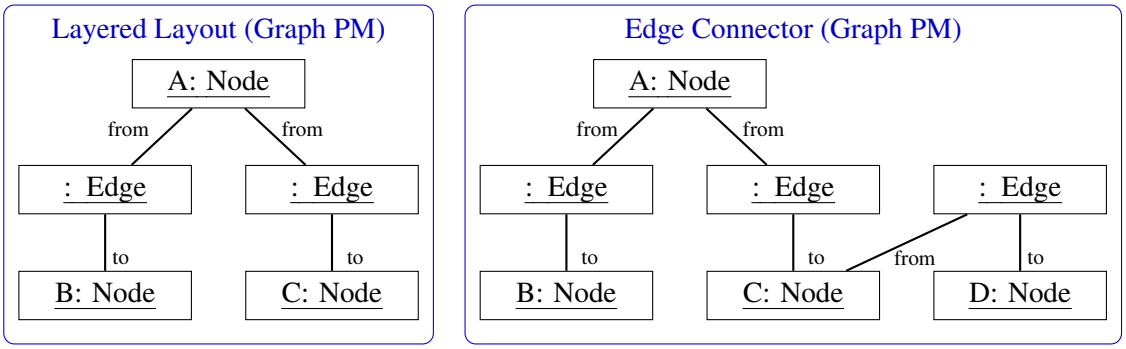
For the example (cf. Figure 4.14(a)), the transformation between the LM and the PM is guided by the TGG, which was described in the last section. One



(a) Example Class Diagram



(b) Language-specific Meta-Model Instance



(c) Different Graph PMM Instances

Figure 4.16: Diagram and Meta-Model Instances

starts with the black parts shown in Figure 4.15. After applying TGG-rule 2 (cf. Figure 4.9) four times, one ends up in the situation shown in Figure 4.15: For each triple (`CClass` object, `EClass` object, `modelObject` link), one `Class2Elem` object and one tuple (`Elem` object, `elems` link) are created.

Another example is shown in Figure 4.16. The diagram visualized in Figure 4.16(a) consists of four classes, two generalizations and one association. Two layout pattern instances are present in the diagram, one layered layout pattern instance that is applied to the classes together with the generalizations, and one edge connector pattern instance that is applied to the classes together with the generalizations *and* the association. The LM that corresponds to the diagram is visualized in Figure 4.16(b).[1] The two corresponding Graph PMs that are generated are shown in Figure 4.16(c).[2] In the first Graph PM, essentially, for each class in Figure 4.16(a), a `Node` object is created, and for each generalization, an `Edge` object is created. In the second Graph PM, in addition, for each association, an `Edge` object is created. Furthermore, several links are created in both cases.

4.3 Layout Patterns

In the following, layout patterns and their corresponding instances are defined. Language-dependent layout patterns and their corresponding instances are distinguished from language-independent layout patterns and their corresponding instances. Language-dependent layout patterns are defined on the LMM, whereas language-independent layout patterns are defined on a PMM.

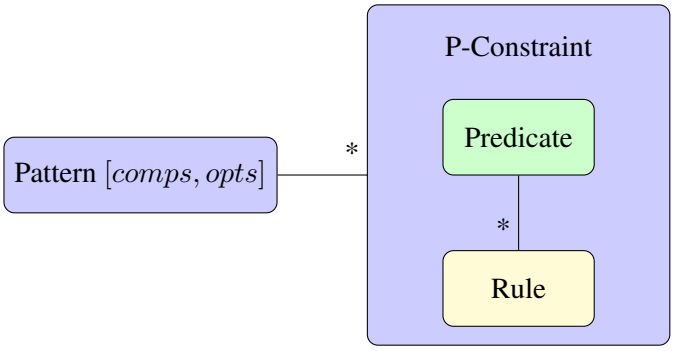


Figure 4.17: Pattern Structure

Figure 4.17 shows the general structure of a layout pattern. A pattern has one or more associated p-constraints. Each p-constraint consists of one predicate, and each predicate has one or more associated rules. A p-constraint is not satisfied if the predicate does not hold. Each pattern gets as input a set *comps* of components, the pattern is applied to. For some patterns, several

[1] The `ClassDiagram` object and the `classes` links were omitted.
[2] The `Graph` objects, `nodes` links, and `edges` links were omitted.

different variants exist. To choose a certain variant, each pattern has a set of options *opts*. This set may be empty. E.g. the minimal size pattern has an option *size* that specifies the minimal size allowed.
The term *constraint* used here should not be mistaken for the term "constraint" used in the context of constraint-based layout. Therefore, these different types of constraints are called *p-constraint* and *csp-constraint* in the following.

4.3.1 Instantiation of Layout Patterns

A layout pattern is instantiated for a set of components. In case of a language-dependent pattern, it gets as input a part of the LMM instance that corresponds to the set of components. In case of a language-independent pattern, it gets as input a PMM instance for the set of components. The layout pattern instance also gets as input values for all options.
Layout pattern instances are written in the form $\mathcal{I}(p, comps, opts)$ in the following. p denotes the instantiated layout pattern, *comps* is the set of components, the layout pattern is instantiated for, and *opts* is an (optional) set of options.

Example

The horizontal alignment (bottom) pattern is based on the Elems PMM. It has one associated p-constraint with one predicate $a.y + a.h = b.y + b.h$. a and b are two `Elem` objects. The predicate has four associated rules:

$a.y := b.y + b.h - a.h$
$a.h := b.y + b.h - a.y$
$b.y := a.y + a.h - b.h$
$b.h := a.y + a.h - b.y$

More details about predicates and rules will be given in the remainder of this chapter.
If the horizontal alignment (bottom) pattern is instantiated for the components A, B, C and D shown in Figure 4.18, this instance is written as $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C, D\}, \{b\})$. Here, p_{AlignH} stands for the horizontal alignment pattern. $\{b\}$ stands for the alignment at the bottom.

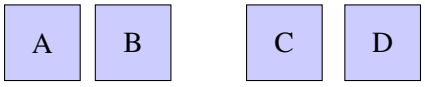


Figure 4.18: Example Diagram

4.3.2 Atomic Layout Patterns

In Chapter 3, several layout patterns were introduced. Each of these patterns defines some layout behavior for a set of components. For example, the horizontal alignment (bottom) pattern aligns n nodes. In order to decrease the complexity of the specification of a layout pattern, the pattern usually does not define layout behavior on top of an arbitrary number of components. Instead, it defines layout behavior on one, two or three components. This layout pattern is then instantiated multiple times.

Example

In case of the horizontal alignment (bottom) pattern, the corresponding pattern takes exactly two components into account. For n components, the pattern is instantiated $n - 1$ times.
For the set of nodes shown in Figure 4.18, the pattern, and hence the Elems PM, is instantiated three times, as can be seen in Figure 4.19: For the nodes A and B, for the nodes B and C, and for the nodes C and D. Hence, the instance $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C, D\}, \{b\})$ is equivalent to the three instances $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{b\})$, $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{b\})$, and $\mathcal{I}(p_{\text{AlignH}}, \{C, D\}, \{b\})$. An alternative would have been the instantiation for the following pairs of nodes: A and B, A and C, and A and D. Choices of this kind need to be made by the pattern creator and may have some implications regarding layout behavior and (or) performance.

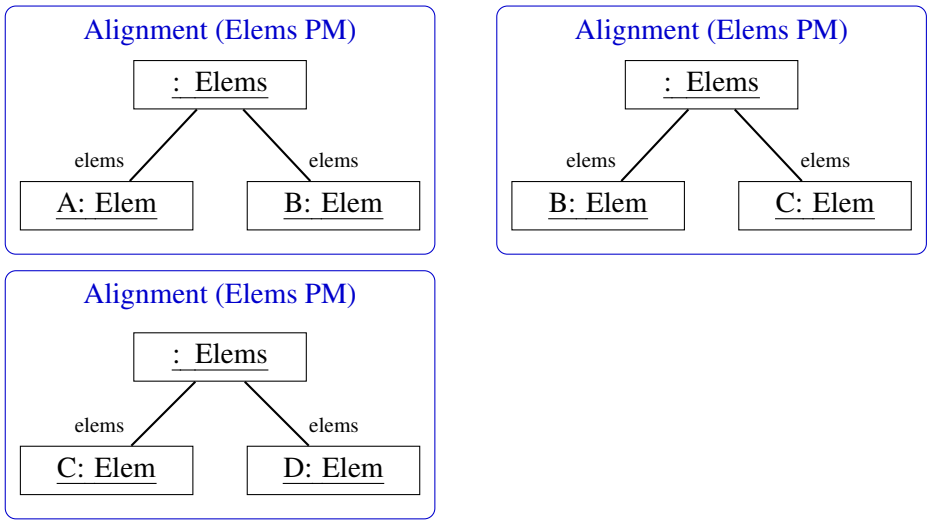


Figure 4.19: PMM Instances of Atomic Pattern Instances

4.4 Specification and Integration of Algorithms

Graph drawing algorithms, constraint-based algorithms, as well as rule-based algorithms may all be encapsulated in patterns. As already mentioned, rule-based algorithms are specialized constraint-based algorithms. When the control algorithm triggers the execution of one of these algorithms, one or more new attribute values are computed on the basis of the PMM instance, hence based on the "old" values. These new values are passed on to the control algorithm, and the layout engine "generates" the new layout of the diagram. In the current implementation, the layout patterns described in Chapter 3 are defined by the following algorithms:

- Graph Drawing Algorithms
  - Tree Layout
  - Layered Layout
  - Circular Layout
  - Node Overlap Removal
  - Edge Connector
- Constraint-based Algorithms
  - Equal Horizontal Distance
  - Equal Vertical Distance
- Rule-based Algorithms
  - Quadratic Component
  - Minimal Size Component
  - Equal Height
  - Equal Width
  - Align in a Row
  - Align in a Column
  - Horizontal Alignment
  - Vertical Alignment
  - List
  - Rectangular Containment
  - Circular Containment

As can be seen, most layout patterns were defined via rule-based algorithms. All these layout patterns show a "local" nature, meaning that only small numbers of components are required for the computation. In contrast, "global" changes are performed via graph drawing algorithms or via constraint-based algorithms. Graph drawing algorithms are used when the graph is affected by the changes, whereas constraint-based algorithms are used when other "global" changes are performed. Local layout patterns and global layout patterns are defined as follows:

- Local layout patterns require only a small and fixed number of components for layout computation.
- Global layout patterns require a large and varying number of components for layout computation.

Some of the layout patterns could have also been defined by another algorithm. Some more details will be given in Chapter 8. In terms of performance (cf. Chapter 9), rule-based algorithms are the first choice, graph drawing algorithms are the second choice and constraint-based algorithms are the third choice.

4.4.1 Specification of Graph Drawing Algorithms

Graph drawing algorithms tend to be quite complex. Hence, it is reasonable to implement them, and not to define them on an abstract level. Therefore the possibility to encapsulate self-implemented algorithms is offered. It is also possible to encapsulate existing implementations, i.e. graph drawing libraries. Examples for standard graph drawing algorithms are a layered layout algorithm, which is used in the layered layout pattern, or a force-directed layout algorithm, which is used in the node overlap removal pattern. Many of the existing graph drawing libraries focus on the visualization of graphs. For instance, they do not take the position or shape of nodes into account. Besides, they often lack of flexibility in terms of user choices.
Nevertheless, several situations exist where graph drawing algorithms are useful. Hence, it is sensible to build a library of graph drawing algorithms that fit into the context of visual language editors. In general, the diversity of the different visual languages and the interactive nature of visual language editors needs to be carefully taken into account when integrating standard graph drawing algorithms.
In the layout patterns described in Chapter 3, several third-party graph drawing algorithms were integrated. They were integrated into the tree layout

pattern, the layered layout pattern, the circular layout pattern and the node overlap removal pattern. The algorithms that were included are either part of the yFiles library [117] or of the Jung library [90]. In addition, one self-programmed graph drawing algorithm was integrated, namely the one used in the edge connector pattern.
Other graph drawing algorithms planned to be included are the edge router called Dokulil [23], which is included in Dunnart [118], and algorithms that are available in the ZEST framework [32], which is part of the Eclipse Plugin GEF [30]. The edge router is chosen because it is an algorithm that perfectly fits into an interactive environment. The ZEST framework is chosen, because it is part of GEF, and perfectly fits into the environment, the layout engine described in this thesis is designed for.

Example - Node Overlap Removal

An example of graph drawing algorithms is the node overlap removal pattern, which ensures that components do not overlap. In the node overlap removal pattern, a force-directed layout algorithm is used. More details about the node overlap removal pattern will be given in Chapter 8.
A force-directed layout algorithm is based on the concept of forces, meaning repulsion and attraction. The idea is that two nodes that are not connected via an edge push each other apart, whereas two nodes that are connected via an edge pull themselves tight. The graph is changed incrementally until a balance of forces is reached.
In the node overlap removal pattern, the force-directed layout algorithm is applied to nodes, only: The algorithm expands the diagram in each step, and is applied until no more overlaps are present in the diagram.

4.4.2 Specification of Constraint-based Algorithms

A constraint-based algorithm is defined by providing a set of csp-constraints. A general-purpose constraint solver then computes a solution to this constraint satisfaction problem (CSP). In the context of this work, no constraint solver was implemented. Instead, a third-party constraint solver is used. This decision was made because there already exist constraint solvers that proved to perform good in the context of layout computation. One of these solvers is the constraint solver QOCA [77]. It was chosen, as it was already integrated in DiaGen [85], the editor generation framework DiaMeta is based on. In the past, the constraint solver QOCA proved to be well suited for layout computation in DiaGen editors - both in terms of functionality and performance. The multi-way constraint solver QOCA supports linear con-

straints, such as $2x + 3y = 5$, but cannot handle constraints involving higher order polynomials, such as $x^2 + y^3 = 5$. Furthermore, QOCA can deal with both, equalities and inequalities, such as $x + y = 5$ or $x + y \leq 5$. The restrictions mentioned are acceptable: With the help of this constraint solver, it is possible to define the layout behavior chosen to be supported.
In [116], a comparison of several constraint solvers is provided. In the context of this work, the constraint solver CHOCO [107] was used to define the layout of two DiaMeta editors, namely a formula editor and an NSD editor. The solver proved to be suitable for our context, but our conclusion is that there is no need to exchange QOCA by another constraint solver.

Example - Equal Horizontal Distance

An example of constraint-based algorithms is the equal horizontal distance pattern, which ensures an equal distance between a set of components. Figure 4.20 shows a diagram after applying this pattern. More details about the equal horizontal distance pattern will be given in Chapter 8.

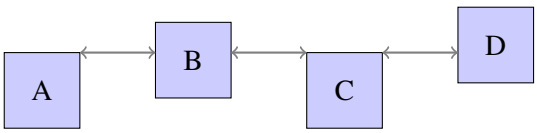


Figure 4.20: Equal Distance between Components

For a set of n components, $n-2$ csp-constraints are created: For every triple of components next to each other, one csp-constraint is created. For the example shown in Figure 4.20, the following two csp-constraints are created:

$$B.x - (A.x + A.w) = C.x - (B.x + B.w)$$
$$C.x - (B.x + B.w) = D.x - (C.x + C.w)$$

The $n - 2$ csp-constraints, together with a list of attributes that should not be modified, are the input for the constraint solver. The attributes that should not be modified "belong" to the component(s) that were changed by the user, and it is avoided that their values are changed by the constraint solver. Based on these requirements, the constraint solver then computes a solution, trying to minimize the attribute changes performed.

4.4.3 Specification of Rule-based Algorithms

Rule-based algorithms are a variation of constraint-based algorithms. Besides a set of csp-constraints, also a set of rules is provided to help satisfying the

CSP. This way, the solution computed is more in control and performance is improved. A performance comparison of constraint-based algorithms and rule-based algorithms can be found in Chapter 9.

Example - Horizontal Alignment (Bottom)

An example of rule-based algorithms is the horizontal alignment (bottom) pattern. Figure 4.22 shows four diagrams after applying this pattern to the diagram shown in Figure 4.21. More details about the horizontal alignment (bottom) pattern will be given in Chapter 8.

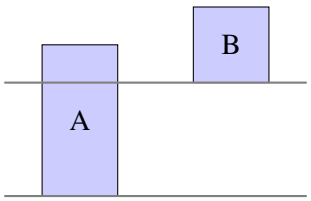


Figure 4.21: Diagram

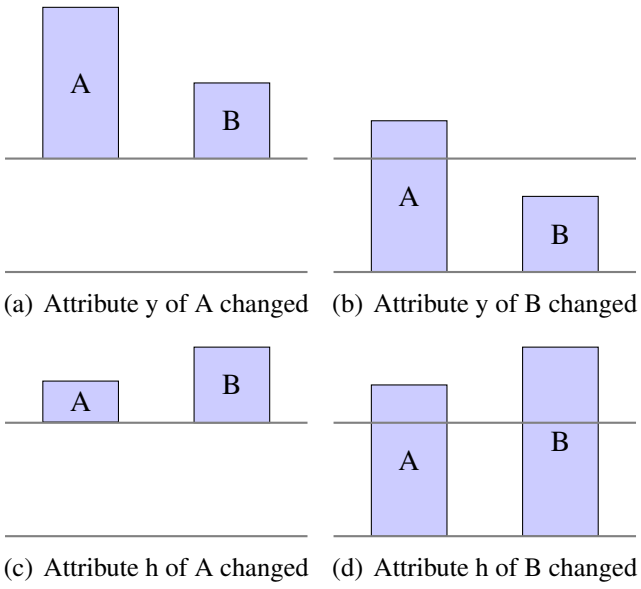


Figure 4.22: Horizontal Alignment – Four Solutions

For a set of n components, $n-1$ csp-constraints are created: For every tuple of components next to each other, one csp-constraint is created. For the example shown in Figure 4.21, the following csp-constraint is created:

$A.y + A.h = B.y + B.h$

Furthermore, the following four rules are created for these two components:

$A.y := B.y + B.h - A.h$
$A.h := B.y + B.h - A.y$
$B.y := A.y + A.h - B.h$
$B.h := A.y + A.h - B.y$

If the csp-constraint is not satisfied, e.g. as can be seen in Figure 4.21, one of these rules may be applied in order to fulfill the csp-constraint. In order to repair the example shown in Figure 4.21, either the y-coordinate of component A is changed, the y-coordinate of B is changed, the height of A is changed or the height of B is changed, as can be seen in Figure 4.22.
The nature of rule-based algorithms is that a small subset of the imaginable solutions is "allowed". This subset is defined by the pattern creator. In the example described above, for instance, it is not possible that the layout engine moves *and* resizes node B. At first sight, this seems to be a restriction, but some informal user experiments showed that rule-based algorithms mostly provide the behavior the user expects. Other (potentially correct) solutions might result in an unexpected or unwanted behavior of the layout engine.

4.4.4 Integration of Graph Drawing Algorithms

Each graph drawing algorithm is viewed as a certain layout pattern. A graph drawing algorithm operates on a pattern-specific meta-model, the Graph PMM, which represents the graph structure.

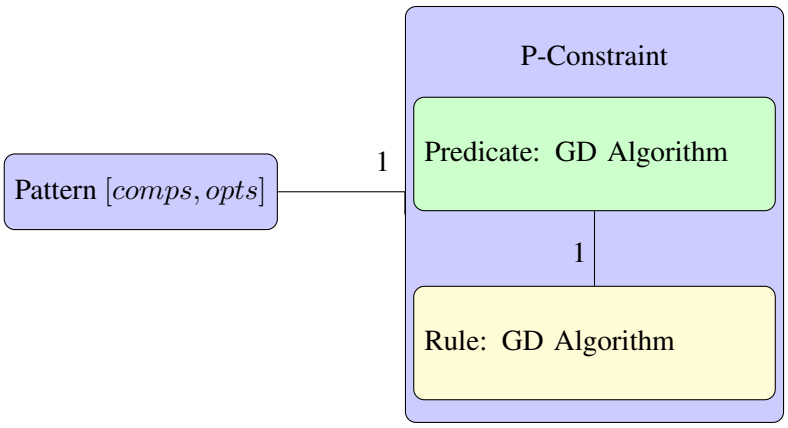


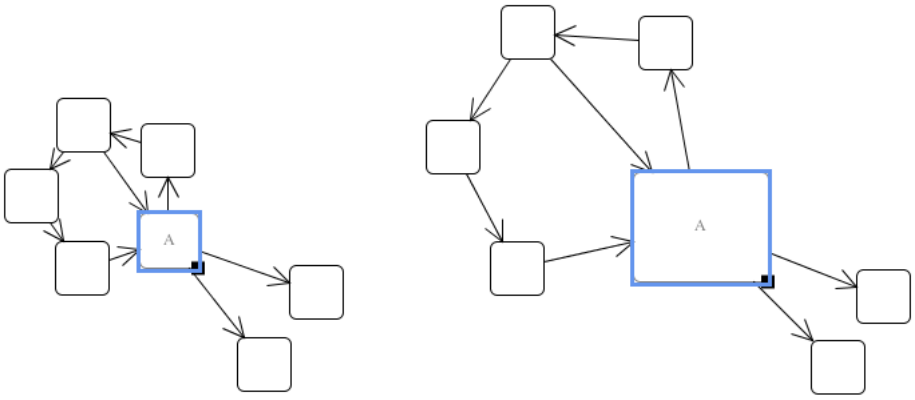
Figure 4.23: Pattern Structure: Graph Drawing Algorithms

As can be seen in Figure 4.23, a pattern that encapsulates a graph drawing algorithm consists of exactly one p-constraint, and the p-constraint has one

predicate and one rule. The predicate checks whether or not a certain algorithm needs to be applied. This "check" is performed by applying the graph drawing algorithm. If the algorithm changes one or more attribute values, the predicate is not satisfied and the changes are performed. Otherwise, the predicate is satisfied and nothing happens.

Discussion

Graph drawing algorithms usually require the graph in a certain form as input. This data structure needs to be created by the pattern. Therefore, a transformation between a Graph PMM instance and this data structure needs to be performed.
After the execution of the algorithm, a translation is applied to the result computed. This way, the diagram is moved to the "correct" position. E.g. if the user moves a node, the layout algorithm should not change this node, but update the rest of the graph.
One might think that the algorithm always needs to be executed after the modification of a component, but this is not the case. E.g. in case of the node overlap removal pattern, the algorithm only updates the diagram if two nodes overlap.



(a) Diagram (b) Diagram after Resizing Node A

Figure 4.24: Circular Layout

One might also think that the rest of the graph only has to be moved (i.e. the same translation is applied to all nodes) after the modification of a component, but this is not the case either. E.g. Figure 4.24 shows a graph before and after resizing node A. In this example, resizing the node causes the

layout algorithm to move each node separately, instead of applying the same translation to all nodes.
Generally, third-party graph drawing algorithms are viewed as a black box. As a consequence, a tight combination with other layout patterns is impossible. To improve this situation, two different approaches are imaginable:

1. If the implementation of the algorithm is known, but cannot be changed, one could design layout patterns that are specifically tailored to the graph drawing algorithm.

2. If the implementation of the algorithm is known and can be changed, one could adjust the implementation such that the algorithm is specifically tailored to the other layout patterns.

Example - Layered Layout Pattern

The layered layout pattern is applied to n components. For that purpose, one pattern instance is created. Checking the predicate of this instance means that the layered layout algorithm is applied to all n nodes. If the algorithm changes the diagram, the predicate is not satisfied and the changes are performed.

4.4.5 Integration of Constraint-based Algorithms

A constraint-based pattern operates on a certain pattern-specific meta-model and encapsulates a certain set of csp-constraints.
As can be seen in Figure 4.25, a pattern that encapsulates a constraint-based algorithm has one p-constraint, which consists of one predicate and one rule. The rule encapsulates the set of csp-constraints and the constraint solver. If the p-constraint is not satisfied, the rule invokes the constraint solver.

Discussion

A constraint solver usually tries to minimize the number of attribute values changed, which is a pleasing characteristic of constraint solvers in the context of mental map preservation.
In addition to the csp-constraints mentioned in the last paragraph, the system adds some additional csp-constraints: The system *freezes* the components that were changed by the user (or by the layout engine). For each component a that was changed by the user, the following csp-constraints are added:

$freeze(a.x)$
$freeze(a.y)$
$freeze(a.w)$
$freeze(a.h)$

These csp-constraints ensure that the layout engine does not change (i.e. *freezes*) the components that were changed by the user.

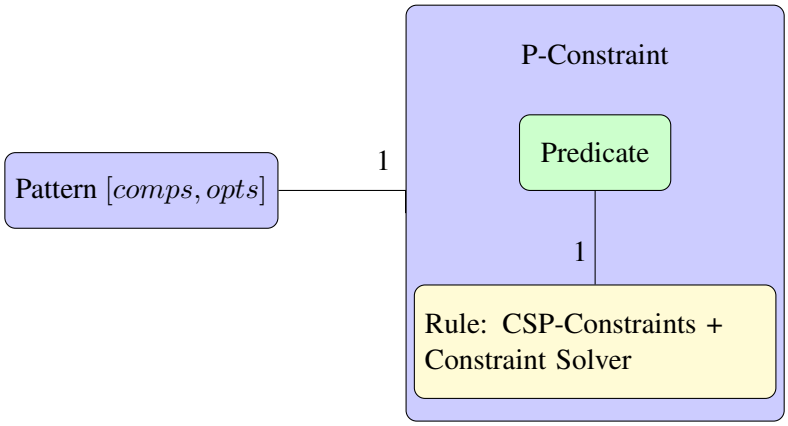


Figure 4.25: Pattern Structure: Constraint-based Algorithms

Example - Equal Horizontal Distance Pattern

The equal horizontal distance pattern is applied to n components. For that purpose, one pattern instance is created. The pattern instance has a predicate of the following form:

$b.x - (a.x + a.w) = c.x - (b.x + b.w)$
$\wedge\ c.x - (b.x + b.w) = d.x - (c.x + c.w)$
...

If this predicate is not satisfied, the following $n - 2$ csp-constraints are the input for the constraint solver:

$b.x - (a.x + a.w) = c.x - (b.x + b.w)$
$c.x - (b.x + b.w) = d.x - (c.x + c.w)$
...

Assuming that component a was changed, then the following csp-constraints are also added:

$freeze(a.x)$
$freeze(a.y)$
$freeze(a.w)$
$freeze(a.h)$

4.4.6 Integration of Rule-based Algorithms

As a third category, rule-based algorithms may be included. A rule-based algorithm operates on an arbitrary pattern-specific meta-model, e.g. the Elems PMM, which represents the required structure.
As can be seen in Figure 4.26, a pattern that encapsulates a rule-based algorithm has a set of p-constraints, and each p-constraint has one predicate and a set of rules. If a predicate is not satisfied, meaning that the predicate is not fulfilled, one corresponding rule is applied. The control algorithm, which will be described in Chapter 5, chooses which associated rule is applied.

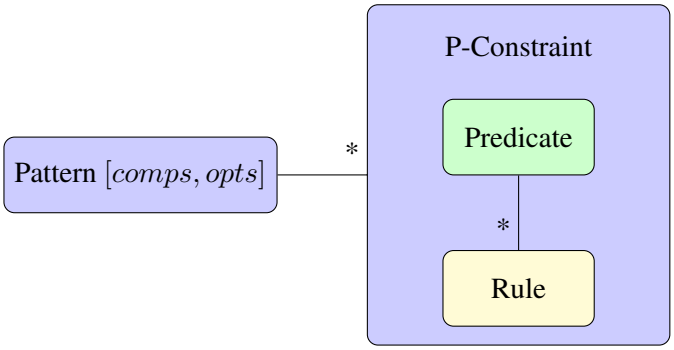


Figure 4.26: Pattern Structure: Rule-based Algorithms

Example - Horizontal Alignment (Bottom) Pattern

The horizontal alignment (bottom) pattern is applied to n components. For that purpose, one pattern instance is created. The pattern instance has a predicate of the following form:

$a.y + a.h = b.y + b.h$
$\wedge\ b.y + b.h = c.y + c.h$
...

The following $n - 1$ sets of rules are associated with this predicate:

$\{\ a.y := b.y + b.h - a.h,\ a.h := b.y + b.h - a.y,$
$\ b.y := a.y + a.h - b.h,\ b.h := a.y + a.h - b.y\ \}$

$\{\ b.y := c.y + c.h - b.h,\ b.h := c.y + c.h - b.y,$
$\ c.y := b.y + b.h - c.h,\ c.h := b.y + b.h - c.y\ \}$
...

If the predicate is not satisfied, one rule is chosen from each of these sets. For instance, the following rules are executed in order to repair the layout:

$b.y := a.y + a.h - b.h$
$c.y := b.y + b.h - c.h$
...

4.5 Atomic Layout Patterns

In order to construct atomic layout patterns, atomic predicates have to be defined. Therefore, each predicate is split into several modular predicates. This can be done as follows:

- Rule-based patterns: These layout patterns can usually be constructed in an atomic fashion. E.g. the horizontal alignment (bottom) pattern has a predicate of the form

  $a.y + a.h = b.y + b.h$
  $\wedge\ b.y + b.h = c.y + c.h$
  ...

  This predicate can be transformed into an atomic rule-based pattern with a predicate of the form

  $a.y + a.h = b.y + b.h$

  The following four rules are associated with this predicate:

  $a.y := b.y + b.h - a.h$
  $a.h := b.y + b.h - a.y$
  $b.y := a.y + a.h - b.h$
  $b.h := a.y + a.h - b.y$

  The first pattern needs to be instantiated once, whereas the second atomic pattern needs to be instantiated $n - 1$ times. E.g. the pattern instance $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C, D\}, \{b\})$ is equal to the atomic pattern instances $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{b\})$, $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{b\})$, and $\mathcal{I}(p_{\text{AlignH}}, \{C, D\}, \{b\})$.

- Constraint-based patterns: These layout patterns can also be constructed in an atomic fashion. E.g. the equal horizontal distance pattern has a predicate of the form

  $b.x - (a.x + a.w) = c.x - (b.x + b.w)$
  $\wedge\ c.x - (b.x + b.w) = d.x - (c.x + c.w)$
  ...

  This predicate can be transformed into an atomic constraint-based pattern with a predicate of the form

  $b.x - (a.x + a.w) = c.x - (b.x + b.w)$

  The first pattern needs to instantiated once, whereas the second atomic pattern needs to be instantiated $n-2$ times. Important is that if one of the pattern instances is violated, the constraint solver needs to compute a solution on the basis of all $n-2$ "atomic" predicates:

  $b.x - (a.x + a.w) = c.x - (b.x + b.w)$
  $c.x - (b.x + b.w) = d.x - (c.x + c.w)$
  ...

  Hence, these $n-2$ "atomic" predicates are the csp-constraints, which are the input for the constraint solver.

  E.g. the pattern instance $\mathcal{I}(p_{\text{EqDistH}}, \{A, B, C, D\})$ is equal to the atomic pattern instances $\mathcal{I}(p_{\text{EqDistH}}, \{A, B, C\})$, and $\mathcal{I}(p_{\text{EqDistH}}, \{B, C, D\})$.

- Graph-based patterns: These layout patterns are different, because an "explicit" predicate does not exist. Instead, the predicate is "checked" by applying the encapsulated graph drawing algorithm. An instance of this type of pattern usually gets a set of n components as input. In contrast, the atomic version gets only one component as input. The first pattern needs to be instantiated once, whereas the second atomic pattern needs to be instantiated n times. Important is that if one of the instances has to be checked, the graph drawing algorithm has to be executed for all n components. E.g. the layered layout pattern instance $\mathcal{I}(p_{\text{Layered}}, \{A, B, C, D\})$ is equal to the atomic pattern instances $\mathcal{I}(p_{\text{Layered}}, \{A\})$, $\mathcal{I}(p_{\text{Layered}}, \{B\})$, $\mathcal{I}(p_{\text{Layered}}, \{C\})$, and $\mathcal{I}(p_{\text{Layered}}, \{D\})$.

Atomization of rule-based patterns and constraint-based patterns has the benefit that the specification of predicates as well as rules is simplified. In case of graph-based patterns, there is no benefit. Hence, rule-based patterns and constraint-based patterns are usually defined in an atomic fashion, whereas graph-based patterns are not.

4.6 Summary

This chapter introduced layout patterns as means to encapsulate certain layout behavior. Based on this concept, an algorithm automatically computes the layout, as described in Chapter 5. The general concept of the pattern-based layout approach was outlined in Section 4.1. The different meta-models, namely the language-specific meta-model, the pattern-specific meta-models and their correlation were described in Section 4.2. In Section 4.3, the concept of layout patterns was discussed in detail. In Section 4.4, it was explained how the different types of algorithms, namely graph drawing algorithms, constraint-based algorithms and rule-based algorithms, are specified and integrated. In Section 4.5, the concept of atomic layout patterns was introduced.
The main benefits of the concept of layout patterns are the modularization of layout behavior and the separation of visual language and layout behavior. This way, reuse of layout behavior is enabled.

4.6.1 Integration into an Editor

Technical details about the integration of the layout engine into an editor is postponed to Appendix A. The integration mainly consists of three parts:

- An instance of the LMM needs to be created from the diagram.
- Based on this LMM instance, the layout engine computes a valid layout, and updates the LMM instance.
- The diagram needs to be updated in accordance to the updated LMM instance.

Additionally, the GUI needs to be extended to make use of the whole functionality of the approach and to improve usability. Some details are given in Chapter 6.

Chapter 5

Control Algorithm for Pattern Combination

In this chapter, details are given about the control algorithm whose purpose it is to control the combination of different layout patterns. In [70], the control algorithm is outlined. A more detailed description of the algorithm can be found in this chapter. The general idea of this control algorithm is sketched in Section 5.1. In Section 5.2, some definitions are given. The control algorithm is described in detail in Section 5.3, and two examples are given in Section 5.4. Some characteristics of the algorithm are discussed in Section 5.5. In Section 5.6, future work is outlined.

5.1 General Idea

The layout of a diagram needs to be updated if the user has changed one or more components. It also has to be updated if the user has triggered the creation of a layout pattern instance. Updating the layout essentially means that the control algorithm is executed.
While computing the layout, a certain set of layout pattern instances comes into play. Based on these pattern instances, a valid layout is computed. The algorithm for pattern control guides this computation. Essentially, it orders the layout pattern instances, and triggers the layout modifications, these pattern instances require.
More precisely, each pattern instance has one or more dedicated p-constraints, and each p-constraint has one dedicated predicate and one or more dedicated rules. All p-constraints *present* in a diagram form a constraint network. The layout of a diagram is *valid*, if all p-constraints are satisfied. The purpose of the control algorithm is to find a variable assignment for which all p-

constraints are satisfied. Therefore, the algorithm applies one rule for each unsatisfied p-constraint in order to "repair" this p-constraint. Cases exist, 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. As an alternative, the user could be asked how he or she wants to proceed, e.g. undoing the user changes, or breaking one or more pattern instances.
Some of the layout pattern instances present in the diagram are automatically created, while others are explicitly created by the editor user. This automatic and user-controlled instantiation of layout patterns is described in Chapter 6. Layout patterns may either be combined manually or automatically. The approach presented in this thesis supports both mechanisms. *Manual combination* of layout patterns means that a control program has to be provided for a visual language editor. This control program handles the combination of a set of layout patterns. Such control programs usually tend to be quite complex, already for very small toy-like examples. Hence, writing such a control program only makes sense in some *very* rare cases. *Automatic combination* of layout patterns means that manually written control programs are unnecessary. Instead, a generic control algorithm is used. As will be seen, in most cases, it is possible to compute the layout with the help of this generic control algorithm.

5.2 Definitions

In the following, definitions are given for the terms variables, p-constraints, predicates, (repair) rules, and constraint nets.

5.2.1 Variables

The control algorithm is based on a certain set of variables. This set consists of some language-specific variables, which are part of the LMM, and some pattern-specific variables, which are part of the PMMs. Based on these variables, p-constraints, predicates as well as rules are defined.

5.2.2 P-Constraints

Three different types of p-constraints are distinguished. The first type are layout p-constraints that "belong" to patterns that modify a set of components and that either operate on language-specific variables or on pattern-specific variables. The second type are component p-constraints that "belong" to patterns that modify a single component and that also either operate on

language-specific variables or on pattern-specific variables. The last type are mapping p-constraints. These p-constraints relate pattern-specific and language-specific variables. Hence, they form the second part of the correspondence between the language-specific model and the pattern-specific models, as was already mentioned in Chapter 4.

Layout P-Constraints

Layout p-constraints "belong" to patterns that *modify a set of components*. They either operate on language-specific variables or on pattern-specific ones. This depends on whether it is a language-dependent or language-independent layout pattern. Typically, these p-constraints are of the second type, and hence, the variables are part of the pattern-specific meta-models. An example is the horizontal alignment pattern, which is defined on top of the Elems PMM, and which makes sure that a set of components is horizontally aligned.

Component P-Constraints

Component p-constraints "belong" to patterns that *modify a single component*, which means that they take care of variables inside a single component. They also either operate on language-specific variables or on pattern-specific ones. Again, this depends on whether it is a language-dependent or language-independent layout pattern. Typically, these p-constraints are also of the second type, and hence, the variables are part of the pattern-specific meta-model. An example is the quadratic component pattern, which is defined on top of the Elem PMM, and which makes sure that a component is quadratic.

Mapping P-Constraints

Mapping p-constraints relate pattern-specific and language-specific variables. They are part of the definition of the correspondence between the LM and the PMs.

5.2.3 Predicates and (Repair) Rules

Each layout p-constraint, component p-constraint, and mapping p-constraint has one predicate and one or more rules assigned. A p-constraint is not satisfied if the predicate does not hold. If only one rule is assigned, a p-constraint may be "repaired" by applying this rule. If more than one rule is available, a choice can be made.

Encapsulated Algorithms

As already described, four types of layout patterns are distinguished, which encapsulate constraint-based layout algorithms, rule-based algorithms, and graph drawing algorithms.

- If a pattern encapsulates a constraint-based algorithm, predicates are defined straightforwardly, and a set of csp-constraints as well as the constraint solver is encapsulated in one associated rule.
- If a pattern encapsulates a rule-based algorithm, predicates as well as rules are defined straightforwardly.
- If 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, these changes are performed.

5.2.4 Constraint Net

Several pattern instances are present in a diagram. All p-constraints that "belong" to these pattern instances together with all other p-constraints present in the diagram form a constraint network. A constraint net is defined as follows:

Definition 1. *A* constraint net *consists of a set V of variables and a set C of p-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 p-constraint $c \in C$ consists of a predicate P_c and a set $R_c = \{r_1, ..., r_i\}$ of (repair) rules. An assignment α* satisfies *a p-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 p-constraint c is not satisfied, a rule may be applied, which changes the value and the status of some variables, thus "repairing" the p-constraint. Therefore, a rule operates on the variables V_c, and reads $\alpha(v_1), ..., \alpha(v_i)$ and $\sigma(v_1), ..., \sigma(v_i)$. It replaces α by α' and σ by σ', following these rules:

- The status of a variable $v \in V_c$ may be one of the following:

  - unchanged and unfixed: $\sigma(v) = u$
  - changed: $\sigma(v) = c$
  - fixed: $\sigma(v) = f$

  A variable has the status *changed* if its value was previously changed by the user or by the layout engine, and the status *fixed* if it was previously fixed by the user or by the layout engine. It has the status *unchanged* and *unfixed* if it neither has the status changed nor fixed.

- A variable may be changed or fixed only once. Hence, 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 $\forall v \in V_c : \sigma(v) \neq \sigma'(v) \Rightarrow \sigma(v) = u$.
- A repair rule can either change the value of a variable or it may fix the value of a variable:
  - $\sigma(v) = c$ and $\alpha(v) \neq \alpha'(v)$
  - $\sigma(v) = f$ and $\alpha(v) = \alpha'(v)$
- After rule application, all variables involved are either changed or fixed:
  - $\forall v \in V_c : (\sigma'(v) = c \vee \sigma'(v) = f)$

5.3 Control Algorithm

As already mentioned, the purpose of the control algorithm is to compute a variable assignment for which all p-constraints are satisfied. The algorithm is sketched in Listing 5.1. Its idea is that changes made are propagated in the diagram. First, two scenarios have to be distinguished:

- The user has changed one or more components. All variables that were changed are marked as changed. All other variables that "belong" to these components are marked as fixed.
- A layout pattern instance has been created for a certain set of components: All variables that "belong" to one of these components are marked as fixed. This component is usually the component that was marked first by the user.

```
function search (α, σ)
begin
  if P_c(α) holds for each p−constraint c ∈ C then the solution is α
    return success
  end if
  compute ordered list L_1 of p−constraints c ∈ C
    such that ¬P_c(α) for each c ∈ L_1 (∗)
  if each c ∈ L_1 is repairable then
    for each c ∈ L_1 do
      compute ordered list L_2 of rules in R_c
        such that A_r(α, σ) for each r ∈ L_2 (∗∗)
      for each r ∈ L_2 do
        apply r, obtaining α', σ'
        if search(α', σ') is successful then
          return success
        end if
      end for
    end for
  end if
  return failure
end
```

Listing 5.1: Control Algorithm for Pattern Combination

Starting with the variable(s) changed or fixed, all p-constraints are checked that involve these variables. For each violated p-constraint, one of the corresponding rules is applied. This rule changes one or more variable(s) and fixes the other variable(s) involved. Again, all p-constraints are checked that involve the variable(s) changed or fixed. This procedure is continued until all p-constraints that need to be checked are satisfied. The first result found is chosen and the algorithm stops immediately. If no solution can be found, the algorithm signals a failure.

In one step, it might be the case that more than one p-constraint needs to be repaired, and hence, more than one rule needs to be executed. In line (*), the algorithm creates an ordered list L_1 of all p-constraints that are not satisfied. The order is defined by the editor developer.

Furthermore, if a p-constraint is violated, there might be several rules available to "repair" this violation. Line (**) determines an ordered list L_2 of repair rules based on some prioritization scheme, which is defined by the layout pattern creator. The rules are tried out by *backtracking* until the algorithm terminates either by finding a new assignment satisfying the predicates of all p-constraints, or by signaling a failure.

5.3.1 Discussion

The control algorithm described in this chapter is essentially a rule-based constraint solver. As usual, it gets as input a set of p-constraints, and computes a solution to this constraint satisfaction problem. The algorithm is rather simple and uses local propagation. It uses backtracking in order to be able to find a solution in many cases. The two main characteristics of it are:

- The p-constraints are defined on some sort of meta-level, enabling the integration of graph drawing algorithms, constraint-based algorithms and rule-based algorithms.
- The computation of a solution is guided by rules.

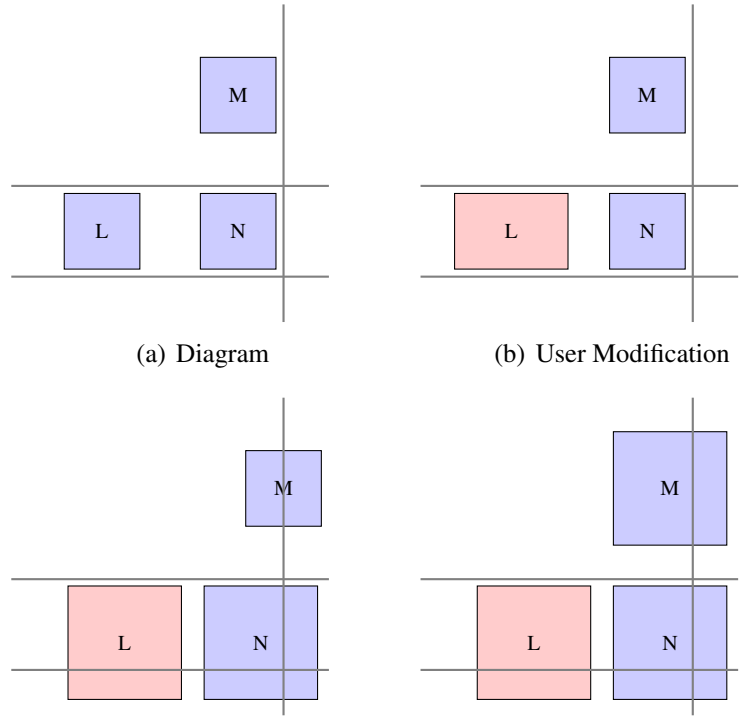
A huge variety of other constraint solving techniques exist. For instance, in [100], a first approach is described that aims at combining different layout modules. In [56], it is discussed how graph drawing algorithms can be combined and how they can be integrated into a visual language editor. Both approaches were quite complex, and suffered from their complexity. The charm of the technique described in this section is mainly its simplicity. One benefit is that the performance of the algorithm is quite convincing, as performance tests as well as user experiments showed. Another benefit is that the solutions that are computed are the ones the user expects. This obviously requires that the layout patterns are defined in a way that other solutions are not allowed. One drawback is that in some cases the algorithm is not able to compute a solution. As user experiments showed, the solutions that are computed are usually sufficient. Another drawback is that the layout patterns themselves are quite complex. This drawback is compensated by the fact that a layout pattern only needs to be defined once, and then can be reused afterwards. The drawback is further mitigated by the introduction of the concept of atomic layout patterns.

5.4 Example Execution of the Algorithm

In the following, two detailed examples of the execution of the control algorithm are given. First, the combination of different layout patterns in the graph editor is examined. This is a rather basic example for the control algorithm. As a second example, the combination of different layout patterns in the VEX editor is examined. Compared to the first one, this is a very demanding example for the control algorithm.

5.4.1 Graph Editor

For the graph editor, several layout patterns have been combined. More details about these layout patterns will be given in Chapter 8. The approach presented in the last sections is discussed via the example shown in Figure 5.1(a). The diagram consists of the components L, M and N.



(a) Diagram

(b) User Modification

(c) L and N are resized, M is moved to the right

(d) L, N and M are resized

Figure 5.1: Running Example: Two Layout Alternatives

The layout pattern instances present in the diagram require that all three components are quadratic, that L and N are aligned at the top and at the bottom, and that M and N are aligned at the right. Hence, the following pattern instances are present in the diagram:

- (1) Horizontal Alignment (Top): $\mathcal{I}(p_{\text{AlignH}}, \{L, N\}, \{t\})$
- (2) Horizontal Alignment (Bottom): $\mathcal{I}(p_{\text{AlignH}}, \{L, N\}, \{b\})$
- (3) Vertical Alignment (Right): $\mathcal{I}(p_{\text{AlignV}}, \{M, N\}, \{r\})$
- (4) - (6) Quadratic Component: $\mathcal{I}(p_{\text{Quad}}, \{L\})$, $\mathcal{I}(p_{\text{Quad}}, \{M\})$, $\mathcal{I}(p_{\text{Quad}}, \{N\})$

The user changes the width of component L (see Figure 5.1(b)). Now the layout engine has to update the diagram accordingly. The two solutions, the layout engine potentially computes, are shown in Figures 5.1(c) and 5.1(d). Figure 5.2 shows the LM of the diagram shown in Figure 5.1(a).

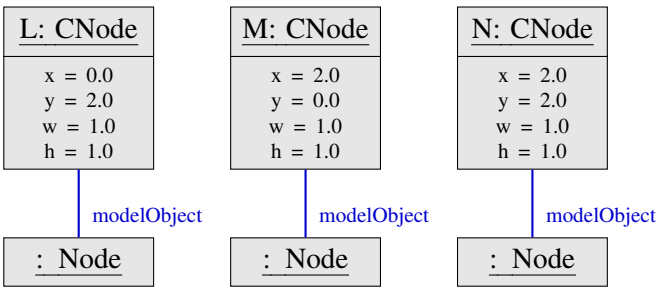


Figure 5.2: Instance of LMM

The horizontal and the vertical alignment pattern both operate on the Elems PMM. In the following, two variants of the Elems PMM are considered, as shown in Figures 5.3(a) and 5.3(b): Both variants consist of the classes `Elems` and `Elem`, which are connected by an association. In the first variant, the class `Elem` has the attributes x, y, w and h, which stand for the (x, y)-position of the top-left corner, the width and the height. In the second variant, the class `Elem` has the attributes t, b, l and r, which denotes the y-coordinate of the top, the y-coordinate of the bottom, the x-coordinate of the left and the x-coordinate of the right border.
The quadratic component pattern operates on the Elem PMM. Here one variant is considered, only, as shown in Figure 5.4(a): the class `SingleElem` has the attributes x, y, w and h.

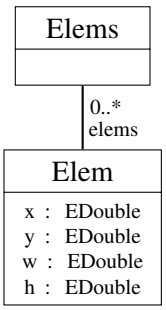
Variables

The control algorithm operates on the following set of variables:[1]

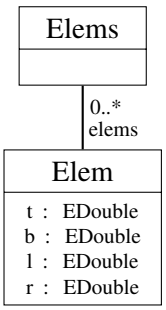
- Language-specific variables: As shown in Figure 5.2, the shape of each node c is defined via its language-specific variables $c.x$, $c.y$, $c.w$ and $c.h$, which denote its top-left corner (x-position, y-position), width and height (w and h).

- Pattern-specific variables (variant 1): As shown in Figure 5.3(c), the alignment pattern refers to the pattern-specific variables $c_i.x$, $c_i.y$, $c_i.w$ and $c_i.h$, which stand for the (x, y)-position of the top-left corner, the width and the height of a component.

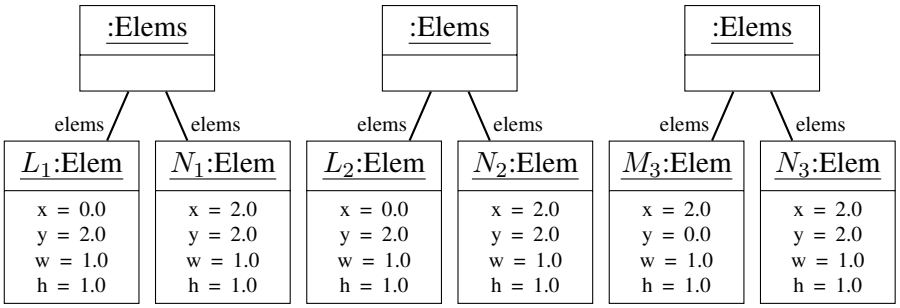
[1] Note that A is the object in the LM, whereas c_i is the object in the PM.



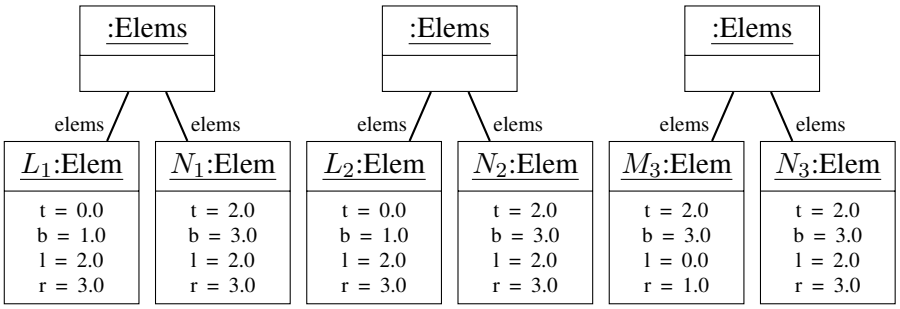
(a) Variant 1



(b) Variant 2

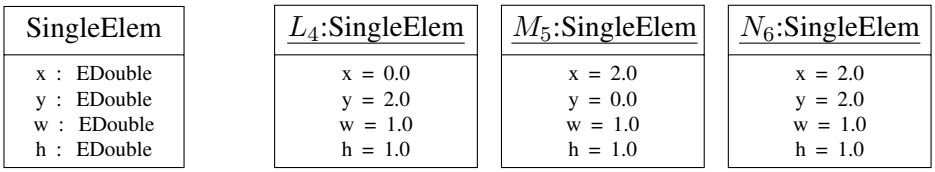


(c) Instances (1) - (3): Variant 1



(d) Instances (1) - (3): Variant 2

Figure 5.3: Elems PMM



(a) Elem PMM

(b) Instances (4) - (6)

Figure 5.4: Elem PMM

- Pattern-specific variables (variant 2): As shown in Figure 5.3(d), the alignment pattern refers to the pattern-specific variables $c_i.t$, $c_i.b$, $c_i.l$ and $c_i.r$, which stand for the y-coordinate of the top, the y-coordinate of the bottom, the x-coordinate of the left, and the x-coordinate of the right border of a component c_i.
- Pattern-specific variables: As shown in Figure 5.4(b), the quadratic component pattern refers to the pattern-specific variables $c_i.x$, $c_i.y$, $c_i.w$ and $c_i.h$, which stand for the (x, y)-position of the top-left corner, the width and the height of a component c_i.

P-Constraints Variant 1

The following p-constraints are present in variant 1 of the example:

- Three *layout p-constraints* were added:
  - Nodes L and N are aligned at the top (layout p-constraint $[L_1N_1 : align_t]$): $L.y = N.y$
  - Nodes L and N are aligned at the bottom (layout p-constraint $[L_2N_2 : align_b]$): $L.y + L.h = N.y + N.h$
  - Nodes M and N are aligned at the right (layout p-constraint $[M_3N_3 : align_r]$): $M.x + M.w = N.x + N.w$
- Three *component p-constraints* were added:
  - Node L is quadratic (component p-constraint $[L_4 : quad]$): $L.h = L.w$
  - Node M is quadratic (component p-constraint $[M_5 : quad]$): $M.h = M.w$
  - Node N is quadratic (component p-constraint $[N_6 : quad]$): $N.h = N.w$

The following holds for all components c, and hence rather trivial mapping p-constraints are required for all pattern instances:

$c.x = c_i.x$
$c.y = c_i.y$
$c.w = c_i.w$
$c.h = c_i.h$

These mapping p-constraints are omitted in the following, in order to reduce the complexity of the diagrams as well as the explanations.
Figure 5.5 shows the complete constraint net. Variables are displayed as ovals, p-constraints as rectangles, and dependencies as lines.

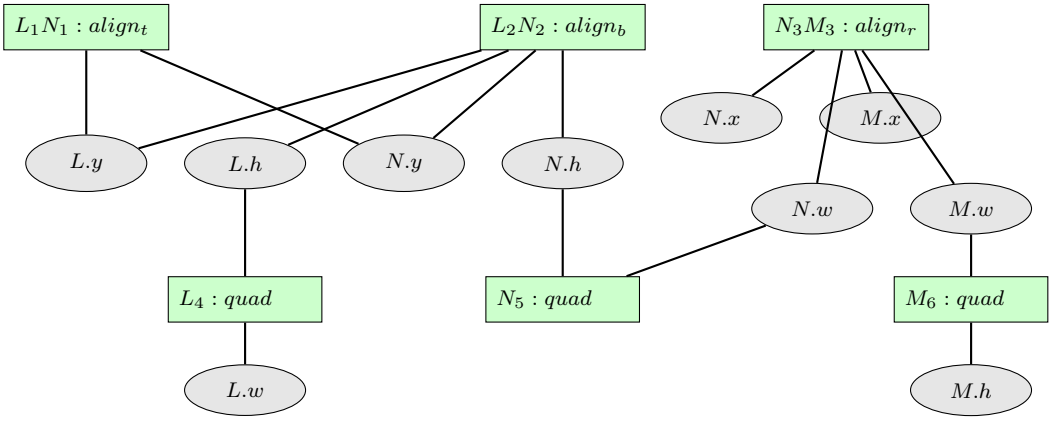


Figure 5.5: Constraint Network of the Example (Variant 1)

P-Constraints Variant 2

The following p-constraints are present in variant 2 of the example:

- Three *layout p-constraints* were added:
  - Nodes L and N are aligned at the top (layout p-constraint $[L_1N_1 : align_t]$): $L_1.t = N_1.t$
  - Nodes L and N are aligned at the bottom (layout p-constraint $[L_2N_2 : align_b]$): $L_2.b = N_2.b$
  - Nodes M and N are aligned at the right (layout p-constraint $[M_3N_3 : align_r]$): $M_3.r = N_3.r$
- Six *mapping p-constraints* were added to relate pattern-specific and language-specific variables:
  - Mapping p-constraint $[L_1L : map_t]$: $L_1.t = L.y$
  - Mapping p-constraint $[L_2L : map_b]$: $L_2.b = L.y + L.h$
  - Mapping p-constraint $[N_1N : map_t]$: $N_1.t = N.y$
  - Mapping p-constraint $[N_2N : map_b]$: $N_2.b = N.y + N.h$
  - Mapping p-constraint $[N_3N : map_r]$: $N_3.r = N.x + N.w$
  - Mapping p-constraint $[M_3M : map_r]$: $M_3.r = M.x + M.w$
- Three *component p-constraints* were added:
  - Node L is quadratic (component p-constraint $[L_4 : quad]$): $L.h = L.w$

- Node M is quadratic
  (component p-constraint $[M_5 : quad]$): $M.h = M.w$
- Node N is quadratic
  (component p-constraint $[N_6 : quad]$): $N.h = N.w$

Once again, the following holds for all components c, and hence rather trivial mapping p-constraints are required for the quadratic component pattern instances:

$c.x = c_i.x$
$c.y = c_i.y$
$c.w = c_i.w$
$c.h = c_i.h$

These mapping p-constraints are omitted in the following, in order to reduce the complexity of the diagrams as well as the explanations.
Figure 5.6 shows the complete constraint net. Variables are displayed as ovals, p-constraints as rectangles and dependencies as lines.

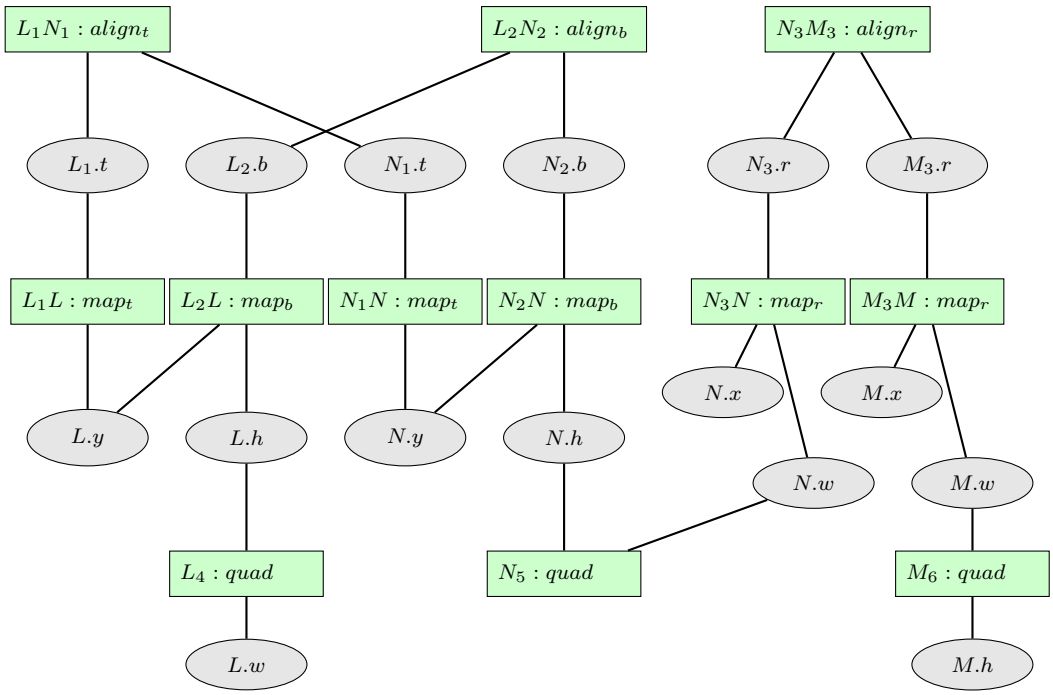


Figure 5.6: Constraint Network of the Example (Variant 2)

Rules

In this example, every p-constraint has one or more associated rules. E.g. the component p-constraint $[L_4 : quad]$ has the associated rules $L.w := L.h$ and $L.h := L.w$. A full list of rules is omitted here. All rules "needed" during the execution of the control algorithm will be introduced on the fly.

Control Algorithm

For the example, the control algorithm may determine two valid solutions: The first one is that L and N are resized, and M is moved to the right, as can be seen in Figure 5.1(c). The second one is that L, N and M are resized, as can be seen in Figure 5.1(d). The example run of the control algorithm for variant 1 is shown in Figure 5.7, the example run for variant 2 is visualized in Figure 5.8. Each arrow denotes a call of the recursive function *search*. Only the two branches that lead to the two valid solutions are visualized. In the end, the control algorithm chooses the first valid solution as the result.

Variant 1 As visualized in Figure 5.7, the starting point is the variable $L.w$, which was changed by the user. As a consequence, the p-constraint $[L_4 : quad]$ is violated and one corresponding rule is applied. The rule associated with $[L_4 : quad]$ changes the variable $L.h$. Afterwards, the p-constraint $[L_2N_2 : align_b]$ is violated, and the corresponding rule updates the variable $N.h$. Then, the p-constraint $[N_5 : quad]$ is violated and one corresponding rule changes the variable $N.w$. Next, the p-constraint $[N_3M_3 : align_r]$ is violated, and now two rules lead to a valid solution. The component M is either moved to the right (rule 1) or it is resized (rule 2). In the first case, rule 1 changes the variable $M.x$ and *Solution 1* is found. In the second case, rule 2 changes the variable $M.w$. As a consequence, the p-constraint $[M_6 : quad]$ is violated. One associated rule updates the variable $M.h$, and *Solution 2* is found. The backtracking algorithm stops when the first variable assignment is found, which satisfies all p-constraints. In the example, this is *Solution 1*. Hence, *Solution 2* is not computed.

Variant 2 As visualized in Figure 5.8, starting point is again the variable $L.w$, which was changed by the user. As a consequence, the p-constraint $[L_4 : quad]$ is violated and the corresponding rule is applied. One rule associated with $[L_4 : quad]$ updates the variable $L.h$. Afterwards, the p-constraint $[L_2L : map_b]$ is violated, and one corresponding rule updates the variable $L_2.b$. Then, $[L_2N_2 : align_b]$ is violated, and one corresponding rule updates the variable $N_2.b$. Next, the p-constraint $[N_2N : map_b]$ is violated, and

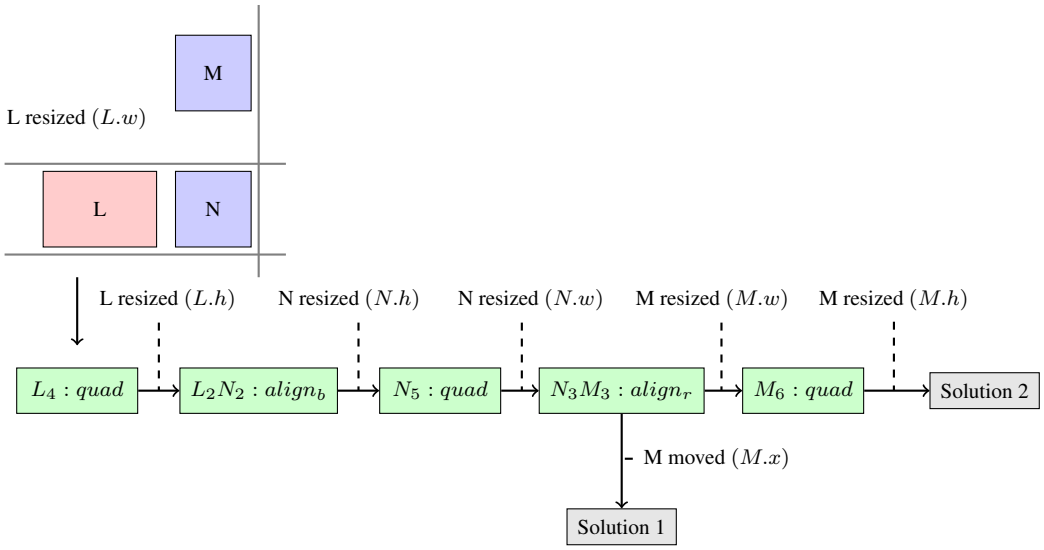


Figure 5.7: Example Run of the Algorithm (Variant 1)

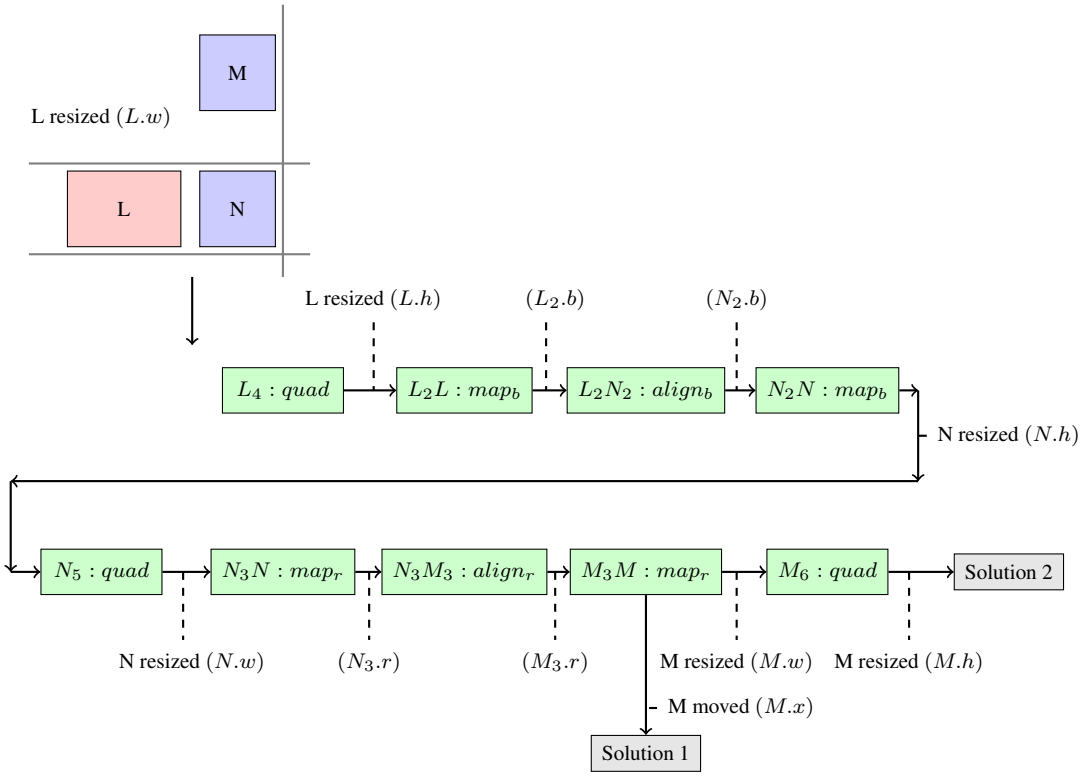


Figure 5.8: Example Run of the Algorithm (Variant 2)

the variable $N.h$ is updated via one corresponding rule. This procedure is continued until the first variable assignment is found, which satisfies all p-constraints. Here, this is *Solution 1*.

Discussion

The main difference between variant 1 and variant 2 is that variant 2 requires mapping p-constraints whereas variant 1 requires quite simple mapping p-constraints, only. Variant 1 has the benefit that the constraint net is usually less complex. One consequence is often a better performance. In contrast, variant 2 has the benefit that the defined p-constraints and rules are potentially simpler. Which variant to choose has to be decided from case to case. In most cases, variant 1 is the better choice. Variant 2 should only be chosen in cases where the layout behavior being defined is quite complex.

5.4.2 VEX Editor

Within a VEX diagram editor, several layout patterns - the containment pattern, the minimal size component pattern and the edge connector pattern - need to heavily interact with each other. More details about these layout patterns will be given in Chapter 8. Instances of all these layout patterns are automatically created. None of them is a layout pattern, whose instantiation is triggered by the user.
The diagram shown in Figure 5.9 and in Figure 5.10(a) represents the λ-expression $\lambda x.(xy)$. The circle A is colored in red, because this is the component that is changed by the user in the example described in the next section. In the diagram shown, nine pattern instances are automatically created. The first two pattern instances (1) - (2) ensure the correct nesting of the circles. The next five pattern instances (3) - (7) ensure that the circles have a minimal size. The last pattern instances (8) - (9) ensure the correct connection of the two lines and the arrow:

- (1) Containment (Abstraction): $\mathcal{I}(p_{\text{CAbs}}, \{A, B, C\})$
- (2) Containment (Application): $\mathcal{I}(p_{\text{CAppl}}, \{C, D, E\})$
- (3) - (7) Minimal Size Component: $\mathcal{I}(p_{\text{MS}}, \{A\})$, $\mathcal{I}(p_{\text{MS}}, \{B\})$, $\mathcal{I}(p_{\text{MS}}, \{C\})$, $\mathcal{I}(p_{\text{MS}}, \{D\})$, $\mathcal{I}(p_{\text{MS}}, \{E\})$
- (8) Edge Connector: $\mathcal{I}(p_{\text{EC}}, \{B, D, E, F, e1, e2\})$, $\mathcal{I}(p_{\text{EC}}, \{D, E, a1\})$; $e1$ is the edge between F and D, $e2$ is the edge between B and E, and $a1$ is the arrow between D and E.

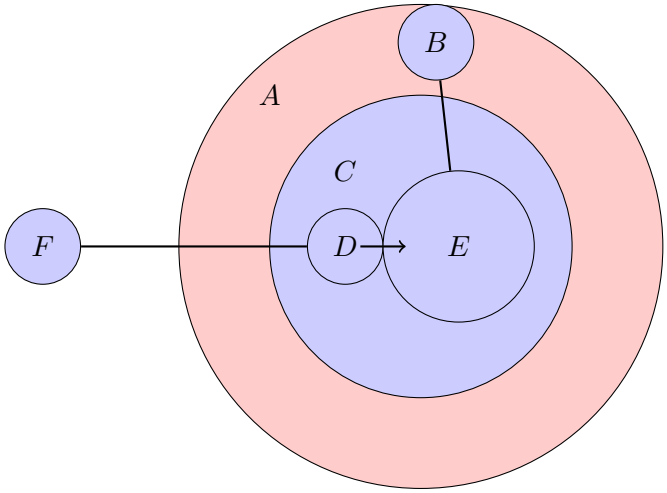
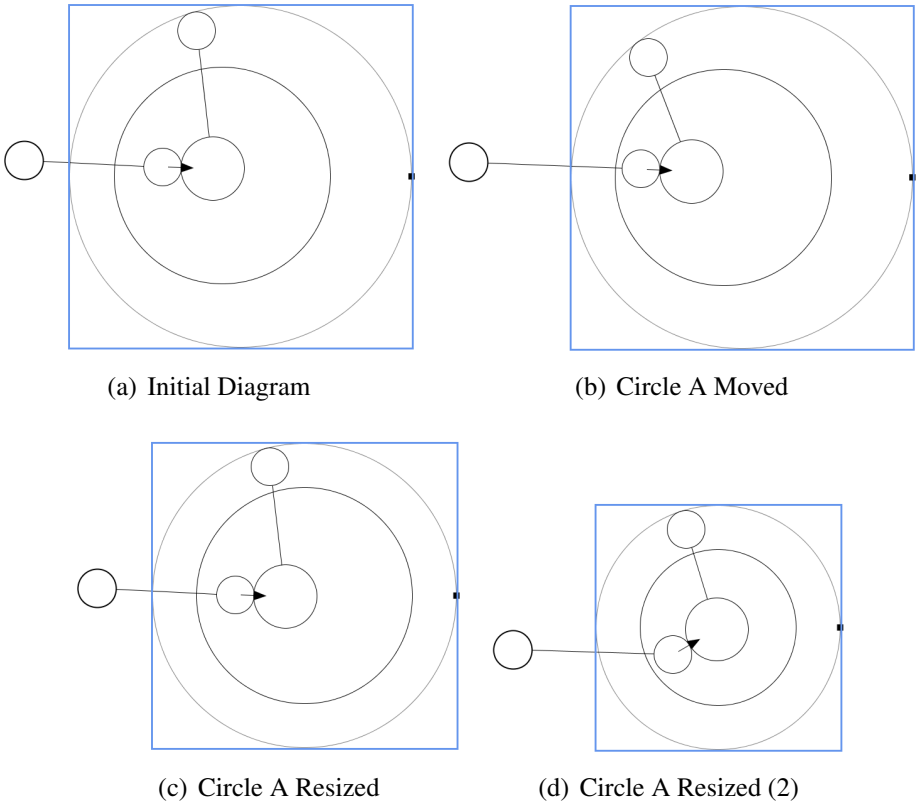


Figure 5.9: VEX Diagram Example



(a) Initial Diagram (b) Circle A Moved

(c) Circle A Resized (d) Circle A Resized (2)

Figure 5.10: VEX Diagram Example

Moving a Component

If the user moves the circle A to the right, the four contained circles B, C, D and E are moved, too. In addition, the lines and the arrow are updated. This example is shown in Figure 5.10(b). The changes performed by the different pattern instances are visualized in Figure 5.11. The user moves component A. As a consequence, components B and C are moved by the circular containment pattern instance (1), components D and E are moved by the circular containment pattern instance (2), and the three edges are updated by the edge connector pattern instances (8) - (9).

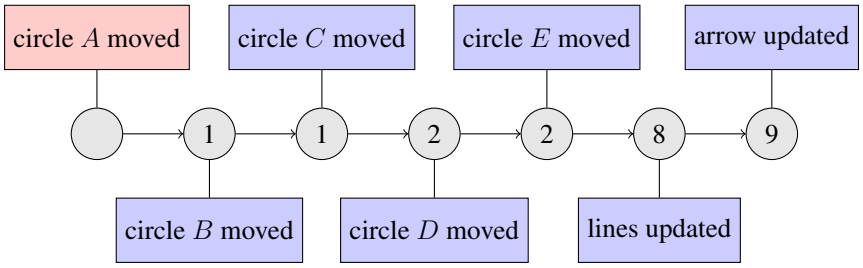


Figure 5.11: Moving a Component

Resizing a Component

If the user reduces the size of the circle A, the four contained circles are moved, and the lines and the arrow are updated. This example is shown in Figure 5.10(c). The changes performed by the different pattern instances are shown in Figure 5.12. The user resizes component A. As a consequence, components B and C are moved by the circular containment pattern instance (1), components D and E are moved by the circular containment pattern instance (2), and the three edges are updated by the edge connector pattern instances (8) - (9).

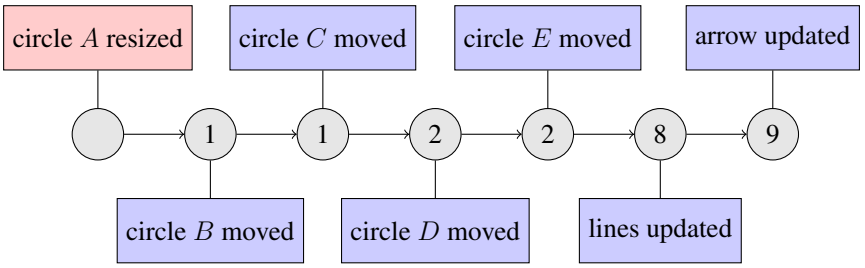


Figure 5.12: Resizing a Component

If the size of the circle A drops below a certain threshold, the size of circle C is decreased, the circles B, D and E are moved, and the lines and the arrow are updated. This example is shown in Figure 5.10(d). The changes performed by the different pattern instances are shown in Figure 5.13. The user resizes component A. As a consequence, component B is moved and C is resized by the circular containment pattern instance (1), components D and E are moved by the circular containment pattern instance (2), and the three edges are updated by the edge connector pattern instances (8) - (9). If the editor user reduces the size of the circle A even more, the size of the circles B, D and (or) E is also decreased. The user can reduce the size of the circle until one or more circles "reach" their minimal size. Then, the minimal size component pattern instances (3) - (7) prevent the user from reducing the size of the circle even more.

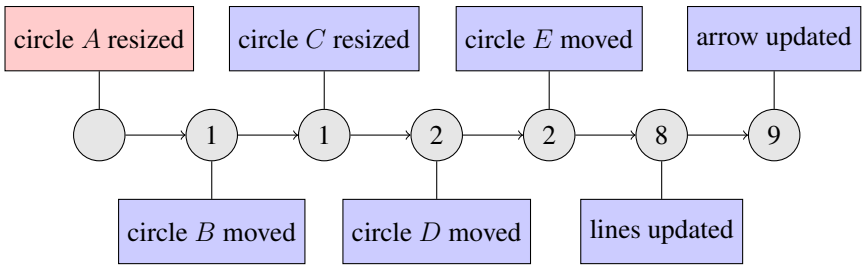


Figure 5.13: Resizing a Component (2)

5.5 Characteristics of the Algorithm

In the following, some characteristics of the control algorithm are discussed.

5.5.1 Correctness

First, the following definition is given:

Definition 2. *The layout of a diagram is correct iff all p-constraints are satisfied.*

The layout of a diagram is correct at any time during user interaction, i.e. all p-constraints that are present in the diagram are satisfied. (User interaction also comprises the computation of the layout.) The correctness of the layout is ensured as follows:

- It is assumed that the layout is correct before the user modifies the diagram. Afterwards, only p-constraints that involve changed variables

are possibly incorrect. All these p-constraints are checked by the algorithm. If one or more of those p-constraints are not satisfied, some variables are changed. Again, all p-constraints are checked that involve these changed variables. This procedure is continued until either all of these p-constraints are satisfied or the algorithm signals that no valid solution can be found. If no valid solution can be found, the changes performed by the user are undone.

5.5.2 Runtime Complexity

Performance tests showed that the most time consuming part of layout computation is usually the execution of the control algorithm presented in this chapter. Either the control algorithm itself or the encapsulated layout algorithm is responsible for this. The following factors are important in the context of performance:

- The size of the diagram.
- The number of p-constraints "present" in the diagram.
- The degree of connectivity of the corresponding constraint network.

The approach is designed for a context in which diagrams are medium sized ($0 - 200$ components), a small number of p-constraints is present in the diagram ($0 - 200$ p-constraints), and the constraint network has a small degree of connectivity. In such practical scenarios, runtime complexity is usually quite good. In Chapter 9, several performance tests are presented that further support the claim that the layout approach can be used in an interactive environment.

5.5.3 Discussion

Several details concerning the control algorithm are discussed in the following.

Component P-Constraints and Layout P-Constraints

The difference between component p-constraints and layout p-constraints is that component p-constraints are responsible for the layout of a single component, whereas layout p-constraints are responsible for the layout of a set of components. Therefore, component p-constraints are allowed to change

variables of a component the user changed, whereas layout p-constraints are not:
In case of layout p-constraints, if one variable of a component is changed, all other variables of this component are fixed. The reason for that is that it would be unnatural if, for instance, the layout engine resizes a component after the user moves this component.
But there are some cases where the layout pattern has to change variables of the same component. One example is the quadratic component pattern. It ensures that a component is always quadratic. If the user changes the width of a component, the layout engine has to update the height of the same component. Such layout behavior is defined via component p-constraints.

Intermediate Results

In rule-based layout patterns, only *extreme solutions* are usually computed. As an alternative, also *intermediate solutions* could be computed. E.g. if the components A (with $y = 10$) and B (with $y = 20$) are aligned at the top, then the extreme solutions are that either $A.y = 10$ and $B.y = 10$ or $A.y = 20$ and $B.y = 20$. An intermediate solution would be, for instance, that $A.y = 12$ and $B.y = 12$. One argument for the first variant is that the resulting layout is often more predictable. One argument for the second variant is that there might be cases where the first variant does not find a valid solution, whereas the second variant does.

5.6 Future Work

In the following, future work concerning the control algorithm is discussed.

5.6.1 Initial Layout

In the examples shown in the last sections, only an incremental layout was computed. With the help of the approach presented, it is also possible to compute an initial layout. For that purpose, all variables are initialized with a certain value, and certain pattern instances are created. Afterwards, not only p-constraints that involve variables that were changed by the user need to be checked, but all p-constraints. This way, a valid layout can be computed. More details about the initialization of variables and about the creation of pattern instances in the context of initial layout computation will be given in Chapter 10.

5.6.2 Partial Results

A partial result is a result in which one or more p-constraints are broken. If the algorithm does not succeed, such a partial result could be computed, and the diagram could be updated accordingly. Considering partial results might have the benefit that the layout engine is more flexible. But it might have the downside that changes performed in the diagram are irreproducible for the user. Moreover, it might have the disadvantage that these kind of layout changes result in semantic changes of the diagram.

5.6.3 Combination with Constraint Solver

It may be the case that the algorithm does not find a solution. In such situations, one could collect all (or a subset of) p-constraints and determine a result with the help of a constraint solver. Especially if a rule-based algorithm does not succeed, one could provide a fallback solution via a constraint solver. This "feature" could be added easily. The feature was not focused on, because it could produce unexpected changes.

5.6.4 Snap to Grid and Hypersnapping

With the approach at hand, it would be straightforward to define snap-dragging [6] and hypersnapping [78] as follows:

- After a user has modified a component, the layout engine updates this component in a sense that it is moved to the "correct" position, e.g. the top-left corner of the component is "snapped" to the grid. This functionality contradicts the rule: "A layout engine should never modify a diagram component the user is currently modifying." Obedient to this rule, this functionality was not included.

- Alternatively, the layout engine could prevent the user from moving a component to a position that is not "correct", e.g. the left side of the component is not aligned to the grid. As this kind of functionality would restrict user interaction, it is not supported by this approach either.

5.7 Summary

In this chapter, details were given about the control algorithm whose purpose it is to control the combination of different layout patterns. The general

idea of this control algorithm was sketched in Section 5.1. In Section 5.2, some definitions were given. The control algorithm was described in detail in Section 5.3, and two examples were given in Section 5.4. Some characteristics of the algorithm were discussed in Section 5.5. In Section 5.6, future work was outlined.

The most important property of the control algorithm is performance. In general, the worst-case time complexity of the algorithm is exponential. But as will be shown in Chapter 9, the performance in practical scenarios is quite convincing.

Chapter 6

User-Controlled Layout Behavior

In the last chapters, the general concept of the layout approach was described. In this chapter, another concept, namely user-controlled layout behavior, which is based on the pattern concept, will be described. The purpose of user-controlled layout behavior is that the user has the possibility to influence the layout pattern instantiation at runtime. Due to the fact that user interaction is required, the graphical user interface of the system is a critical factor. Therefore, the description of user-controlled layout behavior will be based on the user interface of the DiaMeta graph editor, as shown in Figure 6.1.
In this chapter, the user-controlled instantiation of layout patterns and some special features that are useful in the context of user-controlled instantiation of layout patterns are described. The automatic and user-controlled instantiation of layout patterns is discussed in Section 6.1. In Section 6.2, two examples of user-controlled instantiation are given. Some special features that are useful in the context of user-controlled instantiation of layout patterns are described in Section 6.3. In Section 6.4, future work is outlined.

6.1 Instantiation of Layout Patterns

There are two mechanisms of instantiation. The first one is the automatic instantiation of layout patterns, the second one is the user-controlled instantiation of layout patterns.

6.1.1 Automatic Instantiation

In case of automatic instantiation, pattern instances are created automatically. The user may only alter the behavior by completely turning a pattern on or off. The options frame of the graph editor is shown in Figure 6.2. In

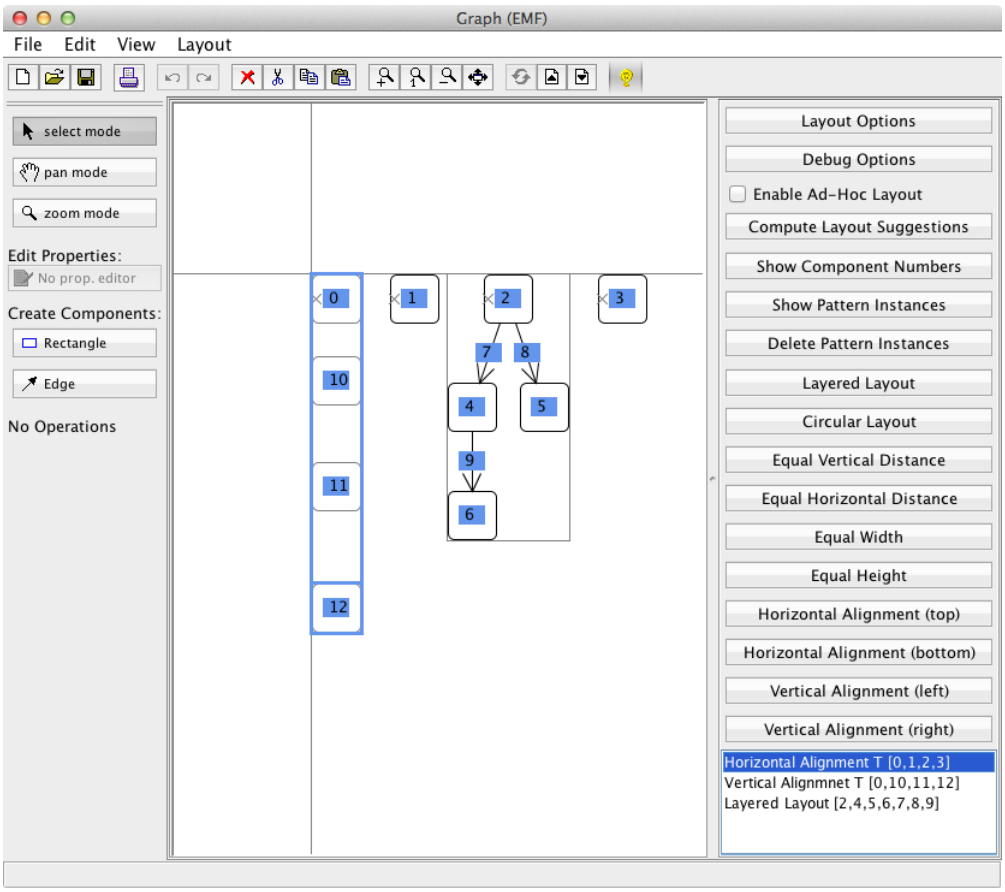


Figure 6.1: DiaMeta Graph Editor

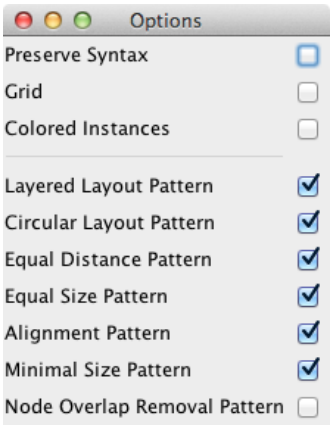


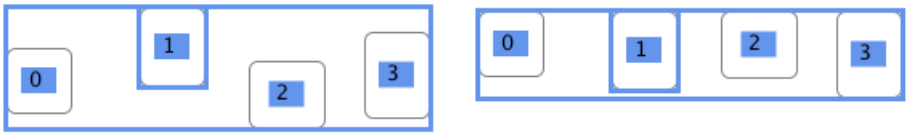
Figure 6.2: GUI for Turning Layout Patterns On and Off

the example, the user has turned off the node overlap removal pattern. Turning a pattern off is, for instance, useful in the following situations:

- It is often pleasing to deactivate one or more layout patterns during the layout improvement - e.g. deactivating the node overlap removal pattern, if the user wants to position components right next to each other, or if he or she wants to move a component across the diagram.
- The deactivation is also sometimes useful during the modification of a diagram - e.g. turning off the containment pattern. Afterwards, the user can change the nesting of components.

6.1.2 User-Controlled Instantiation

In case of user-controlled instantiation, the user may define a part of a diagram to which a certain layout pattern is applied. To do so, the user selects a set of components, and afterwards chooses the layout pattern he or she wants to apply to this part of the diagram. From this point on, the layout pattern is applied to this part of the diagram each time the layout engine is called until the user removes this instantiation request. During the first application of the layout pattern, one component is fixed. This means that the component may not be changed by the layout engine. By selecting the desired component first, the user has the possibility to choose the component to be fixed. In the diagram, the set of selected components is surrounded by a light blue rectangle. In addition, the component that was selected first is also surrounded by a light blue rectangle.



(a) Diagram (b) Diagram: Horizontal Alignment

(c) Pattern Instance List

Figure 6.3: User-Controlled Application

In the example shown in Figure 6.3(a), the user has selected the components 0, 1, 2 and 3. The component 1 was selected first. Afterwards, the user

has selected the layout pattern *Horizontal Alignment (top)* in the list on the right side of the editor. As a consequence, the four components are aligned at the top, as can be seen in Figure 6.3(b). In addition, the entry *Horizontal Alignment T [0, 1, 2, 3]* is added to the list at the bottom right of the editor, as can be seen in Figure 6.3(c). The user can remove this pattern instance again by double-clicking this list entry.

6.1.3 Behind the Scenes

For each pattern instance, an instance of the PMM that corresponds to this layout pattern needs to be created. Therefore, the LM is transformed into a PM. In case of automatic instantiation, the input of the transformation is the complete LM, whereas in case of user-controlled instantiation, the input of the transformation is only an excerpt of the LM. This excerpt corresponds to the user-selected components.

6.2 Examples of User-Controlled Instantiation

Some examples of automatic instantiation as well as of user-controlled instantiation were already provided in Chapter 4. In the following, two examples of user-controlled instantiation of layout patterns will be described in detail. Here, the layered layout pattern, which encapsulates a graph drawing algorithm, and the horizontal alignment pattern, which encapsulates a rule-based algorithm, are depicted.

6.2.1 Layered Layout Pattern

The instantiation of the layered layout pattern can be controlled by the user: The user selects a set of components and applies the layout pattern desired. In the example shown in Figure 6.4, the user added two instantiation requests. As a consequence, two pattern instances are created, and hence, two Graph PMM instances are created. The first Graph PMM instance consists of three `Node` instances for the components A, B and C, and two `Edge` instances. The second Graph PMM instance consists of four `Node` instances for the components D, E, F and G, and three `Edge` instances. These two Graph PMs are visualized in Figure 6.5.[1]

[1] The `Graph` objects, `nodes` links, and `edges` links were omitted.

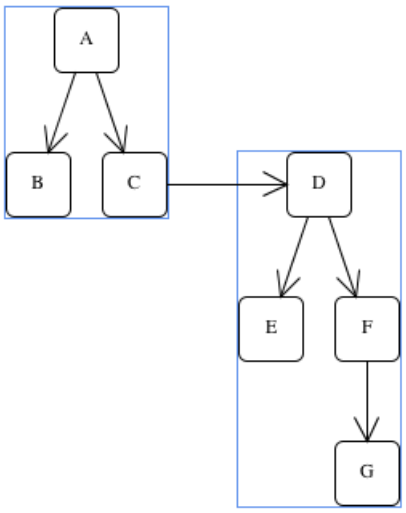


Figure 6.4: Two Layered Layout Pattern Instances

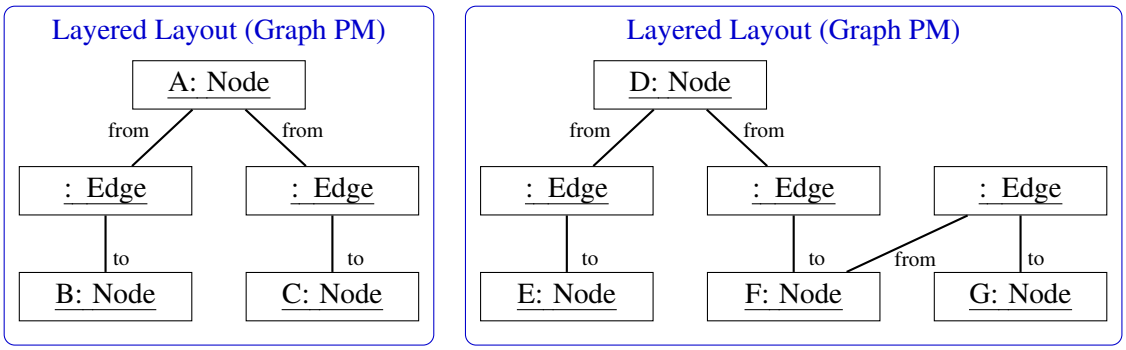


Figure 6.5: Two Layered Layout Pattern Instances

6.2.2 Horizontal and Vertical Alignment Pattern

The instantiation of the horizontal and the vertical alignment pattern is also performed manually. Again, the user selects a set of components and applies the layout pattern desired. In the example shown in Figure 6.6, the user added three instantiation requests. The Elems PM of the first instance contains three `Elem` instances for the components A, B and C, the second one three instances for A, D and E, and the third one two instances for F and C. The three Elems PMs are visualized in Figures 6.7 and 6.8. The first instance, a vertical alignment pattern instance, holds the option *left*, the second one, a horizontal alignment pattern instance, the option *top* and the third one, a horizontal alignment pattern instance, the option *bottom*.

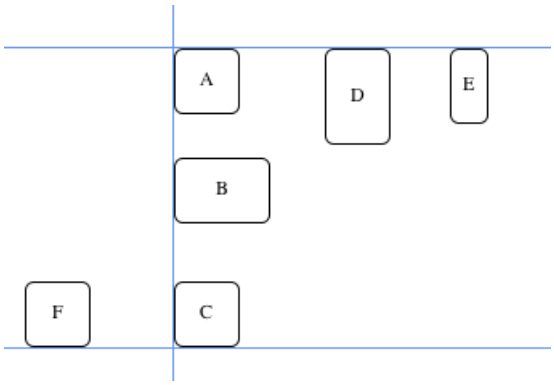


Figure 6.6: Three Alignment Pattern Instances

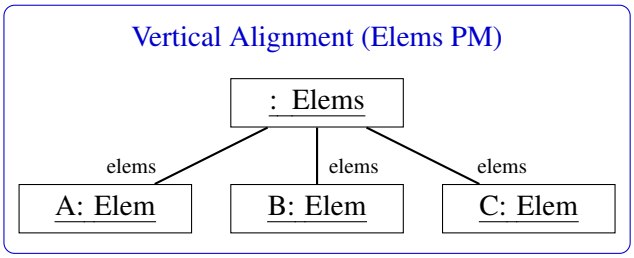
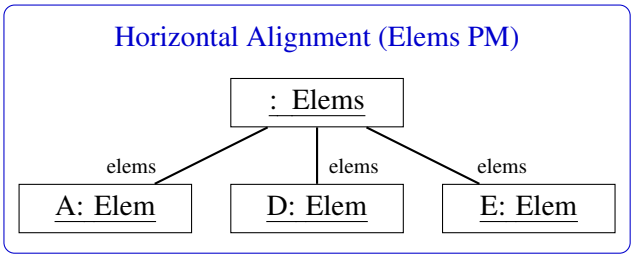


Figure 6.7: One Vertical Alignment Pattern Instance



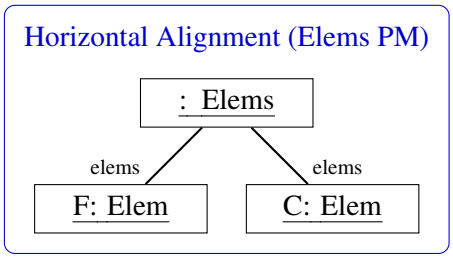


Figure 6.8: Two Horizontal Alignment Pattern Instances

6.3 Useful Features

In the context of the approach, some useful features worth mentioning were added. These features enable a more comfortable usage of the layout approach.

6.3.1 Pattern Instance Aggregation

In a diagram, it might be the case that there is more than one pattern instance present. It is even possible that two or more pattern instances are of the same type. For instance, the following instances of the horizontal alignment pattern p_{AlignH} could be present in the diagram:

- $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{t\})$
- $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{t\})$
- $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{t\})$
- $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{t\})$
- $\mathcal{I}(p_{\text{AlignH}}, \{B, C, D\}, \{t\})$

The layout engine is able to automatically aggregate instances where possible. This essentially means that two or more pattern instances are replaced by one pattern instance that describes the same layout behavior. For the example shown above, this means that the five pattern instances are replaced by the pattern instance $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C, D\}, \{t\})$.
The procedure for pattern instance aggregation is usually as follows:

- For any pattern p, duplicate pattern instances are removed. This means that two pattern instances $\mathcal{I}(p, comps_1)$ and $\mathcal{I}(p, comps_2)$ are replaced by the pattern instance $\mathcal{I}(p, comps_1)$ if $comps_1 = comps_2$. For the example, this means that the second and the fourth pattern instance are removed:
  - $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{t\})$
  - $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{t\})$
  - $\mathcal{I}(p_{\text{AlignH}}, \{B, C, D\}, \{t\})$

  Note that duplicate pattern instances can be created by the editor user, who creates the same pattern instance twice. They might also be the result of the computation of automatic ad-hoc layout (cf. Chapter 7).

- For a pattern p that realizes a transitive relation on components, e.g., the vertical alignment (left) pattern, two pattern instances $\mathcal{I}(p, C_1)$ and $\mathcal{I}(p, C_2)$ are replaced by the pattern instance $\mathcal{I}(p, C_1 \cup C_2)$ if $C_1 \cap C_2 \neq \emptyset$. For the example, this means that the remaining instances are aggregated as follows: First, $\mathcal{I}(p_{\text{AlignH}}, \{A, B\}, \{t\})$ and $\mathcal{I}(p_{\text{AlignH}}, \{B, C\}, \{t\})$ are combined to $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C\}, \{t\})$, and then $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C\}, \{t\})$ and $\mathcal{I}(p_{\text{AlignH}}, \{B, C, D\}, \{t\})$ are combined to $\mathcal{I}(p_{\text{AlignH}}, \{A, B, C, D\}, \{t\})$.

  As can be seen, aggregation is performed incrementally. In each step, two pattern instances are replaced by one pattern instance. This procedure is continued as long as possible.

In general, the aggregation of pattern instances is pattern-dependent. This means that the pattern creator has to define how pattern instances are aggregated: Duplicate pattern instances are always automatically removed. Besides, the pattern creator has to state whether or not the layout pattern realizes a transitive relation on components. If it realizes a transitive relation on components, then the replacement of pattern instances is automatically performed, following the strategy described above.
If the aggregation strategy described is not sufficient, the pattern creator also has the possibility to implement his own aggregation strategy.

6.3.2 Pattern Instance Visualization: Diagram

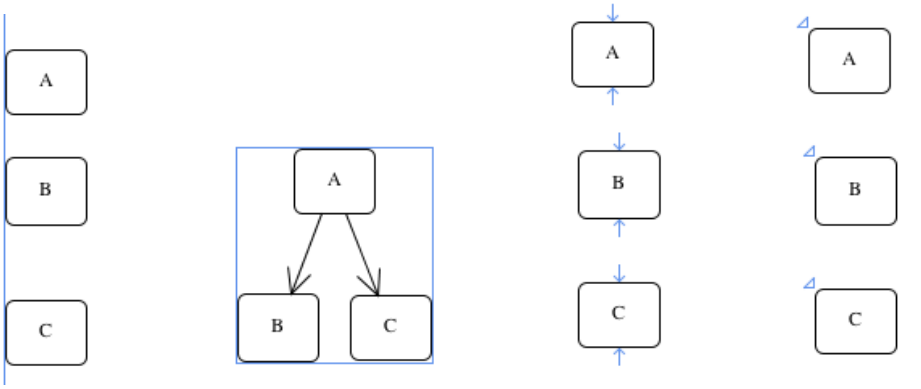
Interaction with the diagram proved to be quite complicated if user-controlled pattern instances are not visualized in the diagram. For that purpose, a *pattern instance visualization mode* is included, which allows to recognize layout pattern instances in the diagram. The existence of a user-controlled instantiation request is visualized in a certain way in the diagram. Some possibilities are shown in Figure 6.9:

- A vertical or horizontal line at the top, bottom, left, or right side of each component. (cf. Figure 6.9(a))
- A rectangle that surrounds all components involved. (cf. Figure 6.9(b))
- Vertical or horizontal arrows between every pair of components involved. (cf. Figure 6.9(c))
- A triangle in one corner of each component. (cf. Figure 6.9(d))

User experiments showed that the following pattern instance visualization is meaningful:

Pattern	Visualization
Tree Layout	Rectangle
Layered Layout	Rectangle
Circular Layout	Rectangle
Node Overlap Removal	–
Edge Connector	–
Equal Horizontal Distance	Arrows
Equal Vertical Distance	Arrows
Quadratic Component	Triangle
Minimal Size Component	–
Equal Height	Triangle
Equal Width	Triangle
Align in a Row	Horizontal Line
Align in a Column	Vertical Line
Horizontal Alignment	Horizontal Line
Vertical Alignment	Vertical Line
List	–
Rectangular Containment	–
Circular Containment	–

Table 6.1: Pattern Visualization



(a) Line (b) Rectangle (c) Arrows (d) Triangle

Figure 6.9: Visualization of Pattern Instances

- User-controlled pattern instances are highlighted in a certain way, whereas automatically created pattern instances are not highlighted.
- The different types of user-controlled pattern instances are visualized as shown in Table 6.1.

Different pattern instances of the same type may be visualized in different colors. This feature can be turned on and off via the option *Colored Instances* in the options frame of the editor, which is shown in Figure 6.2. In Figure 6.10, as an example, four colored instances of the layered layout pattern are shown.

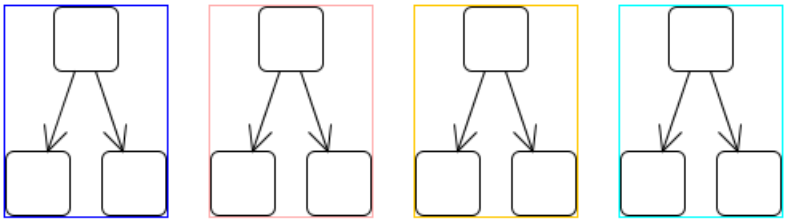
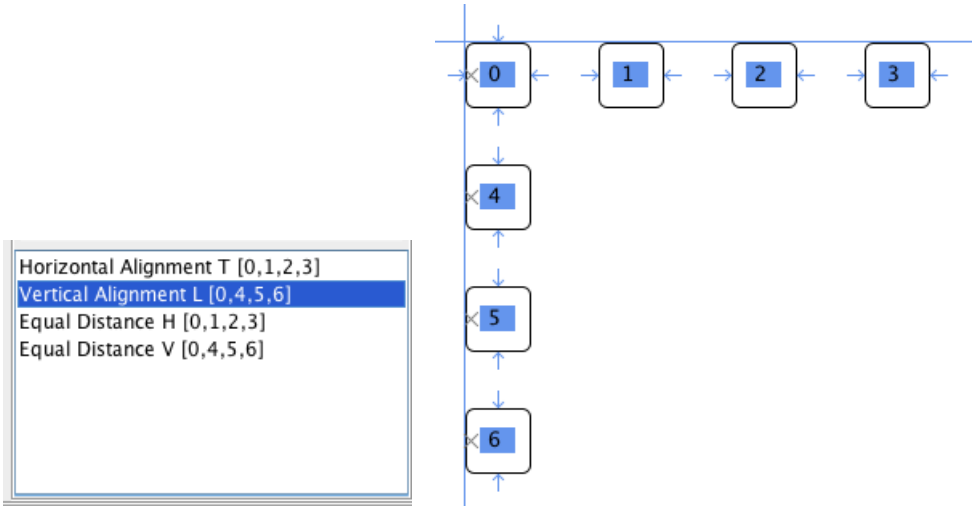


Figure 6.10: Coloring of Pattern Instances

6.3.3 Pattern Instance Visualization: List

User-controlled pattern instances are not only visualized in the diagram itself, but also in the list at the bottom right of the editor. For each pattern instance, one list entry is created. If the user selects one of those list entries, a gray cross is added to each component belonging to the corresponding pattern instance.



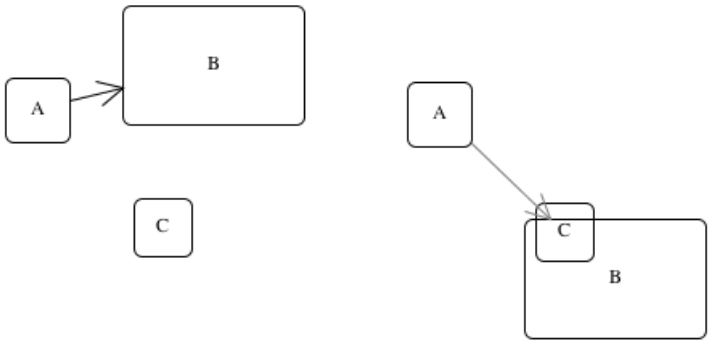
(a) List of Pattern Instances (b) Selected Pattern Instance

Figure 6.11: Selection of Pattern Instances

E.g. in the diagram shown in Figure 6.11(b), four pattern instances were created by the user. As a consequence, the list, which is shown in Figure 6.11(a), has four list entries. E.g. the list entry *Vertical Alignment L [0,4,5,6]* denotes that a vertical alignment (left) pattern instance is present in the diagram, and that this instance makes sure that the components 0, 4, 5 and 6 are aligned vertically. The list entry *Vertical Alignment L [0,4,5,6]* was selected by the user, and hence a gray cross was added to the components 0, 4, 5 and 6 in the diagram. These components are the ones that are affected by the corresponding vertical alignment (left) pattern instance.

6.3.4 Syntax Preservation Mode

DiaMeta supports freehand editing, enabling the editor user to freely position diagram components on the screen. The framework then automatically examines this drawing and creates the corresponding LM. As a consequence, the editor user as well as the layout engine may potentially "destroy" the diagram. As shown in Figure 6.12, the user or the layout engine could move node B in a sense that it no longer connects the nodes A and B, but now potentially connects the nodes A and C. To restrain the user and the layout engine from doing something like this, a so-called *syntax preservation mode* was invented.



(a) Diagram

(b) Diagram after Layout Computation

Figure 6.12: Syntax Preservation

The syntax preservation mode may be turned on and off via the option *Preserve Syntax*, which is available in the options frame shown in Figure 6.2. In syntax preservation mode, it is impossible that the user or the layout engine "destroys" the diagram.

Syntax preservation is achieved by a comparison of the diagram before and after user changes (including the execution of the layout engine). If the syntax of the diagram would be changed, all attribute changes that were performed by the user and the layout engine are undone.
In DiaMeta, the internal graph representation of the diagram is used for the comparison. This graph representation is called graph model [86]. Arrangements of diagram components are described by spatial relationships between them: Each diagram component has several attachment areas at which it can be connected to other diagram components. In graphs, a circle, for instance, has its border as attachment area and an arrow has its start point and its end point as attachment areas. Connections can be established by spatially related attachment areas. In graphs, an arrow has to start or end at the border of a circle in order to be connected to this circle.
In the graph model, each component is modeled by a node (called component node). Each attachment area is also modeled by a node (called attachment node). Edges (called attachment edges) connect component nodes with all attachment nodes that belong to this component node. Furthermore, edges (called relationship edges) connect attachment nodes that are in relationship with each other.
Figure 6.14 shows the graph model of the diagram shown in Figure 6.13. Attachment nodes are drawn as black circles, component nodes as blue rectangles, attachment edges as thin black arrows and relationship edges as thick blue arrows.

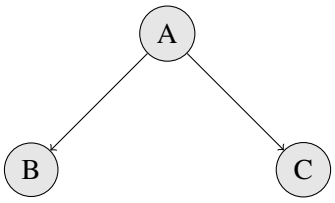


Figure 6.13: Graphs: Example

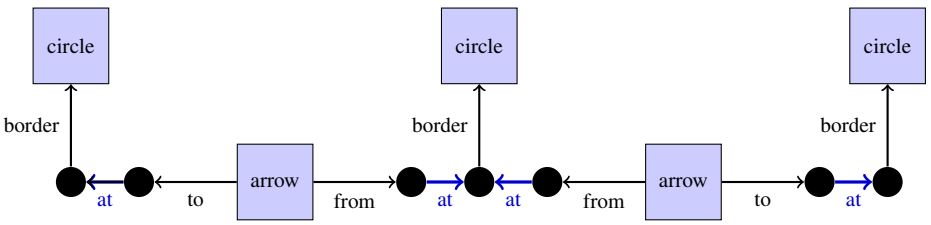


Figure 6.14: Graph Model: Example

The graph model (GM) (named g_1 in the following) that represents the diagram before the user changes took place, and the graph model (GM) (named g_2 in the following) that represents the diagram afterwards are compared. The algorithm that is used for the comparison is shown in Listing 6.1. First, it is checked if g_1 and g_2 have the same set of attachment nodes, namely if the sets AN_1 and AN_2 are equal. Afterwards, for each attachment node $an \in AN_1$, all relationship edges that are connected to it in g_1 are determined. These sets of relationship edges are stored in the set RE_1. Analogously, all relationship edges that are connected to it in g_2 are determined. These sets of relationship edges are stored in the set RE_2. (RE_1 and RE_2 are sets of sets of relationship edges.) Finally, it is checked whether the sets RE_1 and RE_2 are equal.
As an alternative, if the user changes would result in syntax changes, instead of undoing all attribute changes, a subset of the changes performed could be undone. This procedure was perceived being too complex, and the result being more or less irreproducible for the user. Hence, this idea was not developed further.

```
function compareGMs (g1: GM, g2 : GM)
begin
  AN1 := g1.attachmentNodes
  AN2 := g2.attachmentNodes
  if AN1 ≠ AN2 do
    return false
  end if
  for each an ∈ AN1 do
    RE1 := g1.relationshipEdges(an)
    RE2 := g2.relationshipEdges(an)
    if RE1 ≠ RE2 do
      return false
    end if
 end do
 return true
end
```

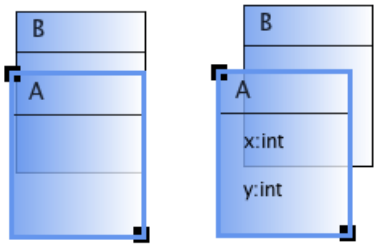
Listing 6.1: Algorithm for Graph Model Comparison

Syntax Changes

In syntax preservation mode, it is impossible to change the syntax of the diagram. If the user wants to perform this kind of changes, he or she has to leave this mode. Usually, the user only enters the syntax preservation mode if he or she solely wants to update the layout of the diagram.

Correctness and Completeness of a Specification

The syntax preservation mode turned out to be a useful tool for the editor developer. In this mode, he or she can easily examine the correctness and completeness of a specification. Actually, some specification errors were discovered during the experimentation in syntax preservation mode.
Figure 6.15(a) shows an example where the user places two classes (more or less) on top of each other, such that they (more or less) completely overlap. This diagram is correct, and hence is allowed in syntax preservation mode. Figure 6.15(b) shows an example where the user tries to place two classes on top of each other. This time, this modification would result in an ambiguity, and hence is not allowed in syntax preservation mode: After further moving class A, the attribute x of class A could either be an attribute of class A or of class B.



(a) Example 1 (b) Example 2

Figure 6.15: Examination of Diagram Correctness

6.4 Future Work

In the following, future work concerning user-controlled layout behavior is discussed.

6.4.1 Conflict Highlighting

As mentioned, it might be the case that the layout engine fails at computing a valid layout.
The existence of unsatisfiable constraints is always possible and does not automatically mean that there is a mistake in the layout specification. E.g. it is impossible to apply the layered layout pattern as well as the alignment pattern to the same set of nodes. Nevertheless, it is reasonable to provide

the user both layout patterns. In such situations, a set of conflicting pattern instances could automatically be identified and highlighted in the diagram. The challenge here is to identify the set of conflicting pattern instances, as this set is ambiguous.
Algorithmically, one could proceed as follows: Add one pattern instance after the other and execute the layout engine. Automatically applied patterns are added first, user-controlled layout patterns are added thereafter. If the layout is no longer computable, the system outputs (highlights) the last pattern instance added together with the pattern instances that were not added, yet. These highlighted pattern instances form the set of conflicting pattern instances. Via this procedure, an optimal solution is usually not found. The procedure rather tries to be fast and to find a comprehensible solution. (For instance, an optimization criterion could be the minimization of the number of conflicting pattern instances.)

6.5 Summary

In this chapter, the user-controlled instantiation of layout patterns and some special features that are useful in the context of user-controlled instantiation of layout patterns were described. The automatic and user-controlled instantiation of layout patterns was discussed in Section 6.1. In Section 6.2, two examples of user-controlled instantiation were given. Some special features that are useful in the context of user-controlled instantiation of layout patterns were described in Section 6.3. In Section 6.4, future work was outlined. Obviously, the seamless integration of the layout engine into an editor is a crucial factor. Besides user-controlled layout behavior, many other useful features are imaginable. Two of them will be described in Chapter 7, namely layout suggestions and ad-hoc layout.

Chapter 7

Layout Suggestions and Ad-hoc Layout

In this chapter, the concept of layout suggestions and the concept of ad-hoc layout are described, two features that are based on the idea of user-controlled layout behavior. In [73], layout suggestions and ad-hoc layout have been sketched already. A more detailed description can be found in this chapter. The concept of layout suggestions is discussed in Section 7.1. In Section 7.2, the concept of ad-hoc layout is described. In Section 7.3, future work is outlined.

7.1 Layout Suggestions

When the editor user selects a set of diagram components and, therefore, defines a sub-diagram, the layout engine can identify those patterns from the set of all available patterns that may be applied to the sub-diagram. To help the user decide what layout pattern he or she should apply, a quality criterion is introduced. This quality criterion identifies those layout patterns whose application would result in a small layout modification. Based on this quality criterion, the layout suggestions are computed.

7.1.1 Highlighting of Layout Suggestions

In a diagram editor, layout suggestions can be used as follows: The user selects diagram components and then pushes the button *Compute Layout Suggestions*. As a consequence, the layout engine computes the layout suggestions. They are displayed by highlighting the corresponding buttons on the right side of the editor in a certain color. Gray indicates that the cor-

responding pattern cannot be applied to the selected components because it either does not fit the chosen diagram part or is inconsistent with the currently active pattern instances. The other buttons are colored blue. Stars are added to the button labels of the patterns within the set of layout suggestions, i.e. whose application would result in a small layout modification.

Figure 7.1 shows a diagram that consists of several nodes and edges. The user already added two user-defined pattern instances: A horizontal alignment (top) pattern instance was created for the nodes 0, 1 and 2. A layered layout pattern instance was created for the nodes 1, 3, 4 and 5 and the edges 8, 9 and 10. Now the user selects the nodes 0, 6 and 7, and asks the system what layout patterns could be applied. The system indicates that all patterns can be applied. The equal height pattern and the vertical alignment (left) pattern are suggested because they only require small attribute changes.

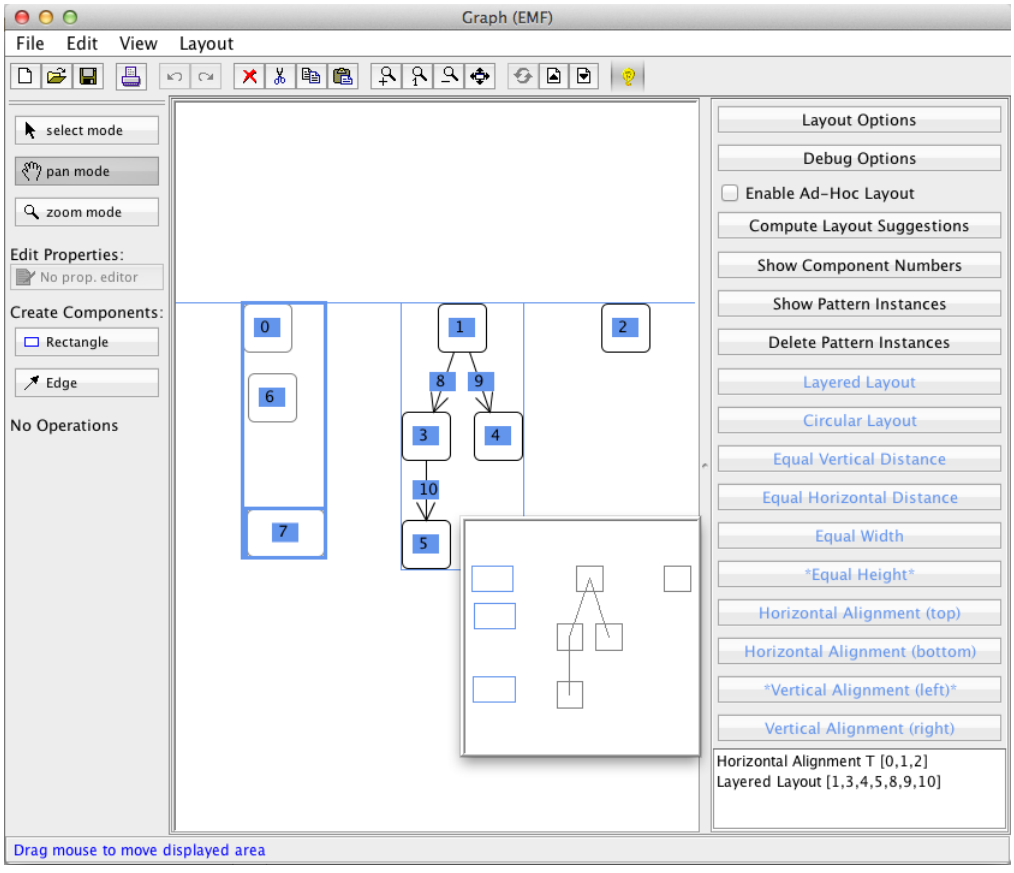
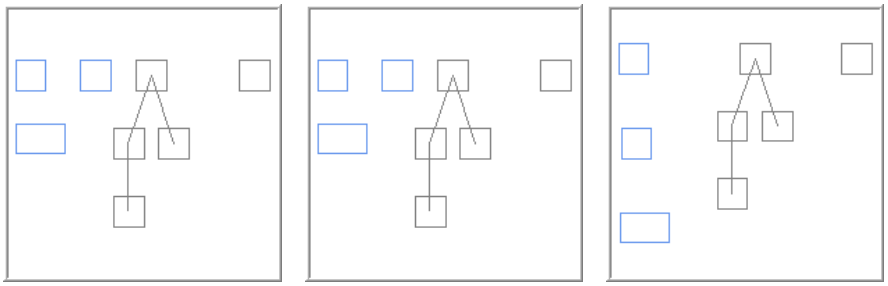
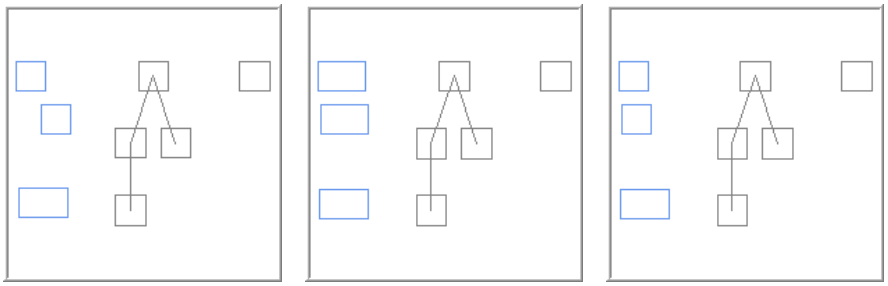


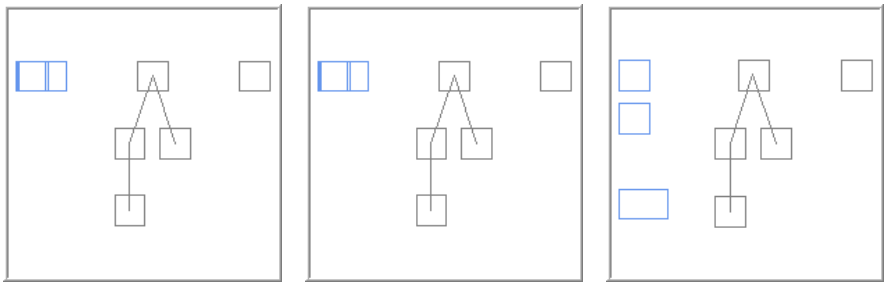
Figure 7.1: Highlighting of Layout Suggestions



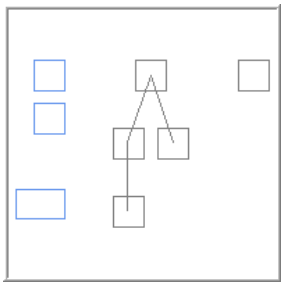
(a) Layered Layout (b) Circular Layout (c) Equal Vert. Dist.



(d) Equal Horiz. Dist. (e) Equal Width (f) Equal Height



(g) Horiz. Align. (t) (h) Horiz. Align. (b) (i) Vert. Align. (l)



(j) Vert. Align. (r)

Figure 7.2: Preview of Layout Suggestions

7.1.2 Preview of Layout Suggestions

To further support the user, a preview of the updated diagram after the application of a layout pattern is provided. Here, three different variants are imaginable:

- As the first alternative, a preview of the whole diagram could be shown.
- As the second alternative, only the selected components could be presented. This way, only a small part of the diagram is shown.
- As the third alternative, the preview could show the selected components together with the components that are affected by the layout changes. This way, only the "important" part of the diagram is shown.

Currently, the first alternative is integrated in the system: When the editor user moves his or her mouse over one of the buttons, a preview of the whole diagram after the corresponding layout pattern is applied, is shown. E.g. in Figure 7.1, a preview of the diagram after the equal width pattern was applied to the nodes 0, 6 and 7 is shown. In Figure 7.2, all previews for the example shown in Figure 7.1 are visualized.
The first variant was chosen, because it gives the best overview of the changes that are implied by the application of the layout pattern. This variant should only be chosen if the diagrams that are drawn are quite small.

7.1.3 Concept

First, two questions need to be answered:

- Which layout patterns should be suggested?
- How can these layout patterns be identified?

It turned out that computing the set of layout suggestions is straightforward:

- The set of layout suggestions initially contains all available layout patterns.
- First, all patterns are removed from the set that cannot be applied to the selected sub-diagram because the pattern either does not fit to the chosen sub-diagram or is inconsistent with the currently active pattern instances. The pattern does not fit the chosen sub-diagram if no match can be found in the LM. Inconsistency is checked by applying the layout pattern temporarily. If the layout engine is not able to compute a valid layout afterwards, the layout pattern is not applicable, and hence, inconsistent with the currently active pattern instances.

- For each of the remaining patterns, the layout engine computes the layout modifications that would be necessary if the pattern was applied. Only those patterns are kept in the set that would require a small layout modification, the others are removed from the set.

Quality Criterion

A rather simple metrics is used for deciding whether a layout modification is small, namely the mean square of all attribute changes. It is computed as follows: Let C_s be the set of components selected by the user and $A(c)$ the set of all attributes of a component $c \in C_\mathrm{s}$. Furthermore, for any component $c \in C_\mathrm{s}$ and any attribute $a \in A(c)$, let $\mathrm{val}_\mathrm{prev}(a)$ and $\mathrm{val}(a)$ be the attribute value of a before and after computing the layout modification. Then the size d of the layout modification is computed by

$$d = \frac{\sum_{c \in C_\mathrm{s}} \sum_{a \in A(c)} (\mathrm{val}(a) - \mathrm{val}_\mathrm{prev}(a))^2}{\sum_{c \in C_\mathrm{s}} |A(c)|}.$$

A layout modification is considered *small* if $d < m$ for some threshold m.
Figure 7.1 shows the results when computing the layout suggestions for the nodes 0, 6, and 7. All buttons are drawn in light blue since each pattern can be applied. But not all of them would result in small layout modifications, e.g. aligning these three nodes vertically on the right side. *Vertical Alignment (right)*, therefore, is not a layout suggestion and not marked with stars. In contrast, *Vertical Alignment (left)* is a layout suggestion since the three nodes are almost aligned vertically on the left side. This button, therefore, is marked with stars.
Instead of the mean square of all attribute changes, a more sophisticated strategy might become necessary. The quality criterion currently used leads to the following problem: Suppose you have two diagrams, one with one component named A, and one with one component named B. Let component A have the attributes x and y, and component B the attributes x, y, w and h. Suppose that the attribute x is increased by 10. Then d is computed as follows: $d_A = 100/2 = 50$ and $d_B = 100/4 = 25$. Hence, this difference potentially could lead to differing suggestions in the two scenarios.

7.2 Ad-hoc Layout

So far, layout suggestions have been computed based on some user-selected diagram parts. Automatic ad-hoc layout as a more powerful mode of operation is made possible by allowing the layout engine to autonomously extend

the set of selected components. The following sections restrict these extensions in different ways, enabling different kinds of ad-hoc layout.

7.2.1 Global Ad-hoc Layout

Global Ad-hoc Layout (GAL) means that layout suggestions are computed and automatically applied for all elementary extensions C of the set C_s of user-selected components, i.e., for each set $C \subseteq C_{all}$ such that $|C \setminus C_s| = 1$ where C_{all} indicates the set of all diagram components.
The algorithm is outlined in Listing 7.1. It gets as input the set P of all layout patterns available in the diagram editor, the set C_s of selected components, and the set C_{all} of all components (formal parameter C_{max}). It returns *true* iff it has applied a pattern. The algorithm may influence the diagram layout globally because it considers every diagram component.

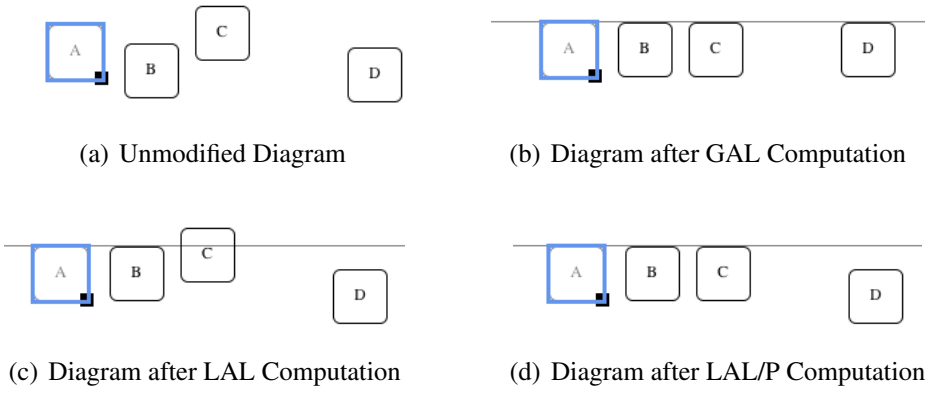
```
proc computeAdHocLayout(P, C_s, C_max)
  candidates := P × (C_max \ C_s)
  do
    instances := ∅
    for each (p, c) ∈ candidates do
      if p applied to C_s ∪ {c} results in small layout modifications then
        apply p to C_s ∪ {c}
        add (p, c) to instances
      end if
    end do
    candidates := candidates \ instances
  while instances ≠ ∅
  return candidates ≠ P × (C_max \ C_s)
end
```

Listing 7.1: Algorithm for Computing GAL and LAL

E.g. in the example shown in Figure 7.3(a), the user has moved component A. As a consequence, layout suggestions are computed and applied for the sets $\{A, B\}$, $\{A, C\}$, and $\{A, D\}$ of components. The horizontal alignment (top) pattern is instantiated for these sets because the attribute changes after application are quite small, as can be seen in Figure 7.3(b).

7.2.2 Local Ad-hoc Layout

GAL must try layout patterns for all diagram components, even those far apart from the user-selected ones. Therefore, performance is an issue, and



(a) Unmodified Diagram

(b) Diagram after GAL Computation

(c) Diagram after LAL Computation

(d) Diagram after LAL/P Computation

Figure 7.3: Automatic Ad-hoc Layout

it may lead to some surprising layout modifications at distant diagram locations. *Local ad-hoc layout (LAL)* enhances on these aspects. In contrast to GAL, LAL computes layout suggestions only for components that are in a close neighborhood of the selected components. These components form the set $C_\mathrm{n} \subseteq C_\mathrm{all}$ of components. More precisely, only components $c \in C_\mathrm{all}$ that are near the set C_s of selected components are considered. For that purpose, the minimal distance between the bounding box of the component c and the bounding box of each of the selected components $c_\mathrm{s} \in C_\mathrm{s}$ is computed. A component is considered being near the selected components if this distance is less than a certain threshold t:

$$C_\mathrm{n} = \{c \in C_\mathrm{all} \mid \exists c_\mathrm{s} \in C_\mathrm{s} : \mathrm{dist}(c, c_\mathrm{s}) < t\}$$

LAL can again use the algorithm in Listing 7.1. However, it is called with the neighborhood C_n instead of the set C_all of all components.
E.g., in the example shown in Figure 7.3(a), the user has moved the component A. The neighborhood of C_s is just $C_\mathrm{n} = \{A, B\}$ since the components C and D are too far apart. LAL then suggests and applies the horizontal alignment (top) pattern for the set $\{A, B\}$ of components. The result is shown in Figure 7.3(c).

7.2.3 Local Ad-hoc Layout with Propagation

As the example shows, LAL has only small benefits. *Local ad-hoc layout with propagation (LAL/P)* enhances on this aspect. It starts with the set $C = C_\mathrm{s}$ of user-selected components and computes the local ad-hoc layout for C. The set C is then extended by all components that have just been modified, and local ad-hoc layout is computed again for this extended set C of components.

This iteration is continued until no further pattern instance has been created. This way, layout improvement is "propagated" through the diagram, as long as new layout suggestions can be computed. The propagation algorithm is outlined in Listing 7.2.

```
proc computeAdHocLayoutWithPropagation(P, C_s, C_all)
  C = C_s
  do
    compute neighborhood C_n of C
    changed := computeAdHocLayout(p, C, C_n)
    C_c := components changed by the layout engine
    C := C ∪ C_c
  while changed
end
```

Listing 7.2: Algorithm for Computing LAL/P

E.g., in the example shown in Figure 7.3(a), the user has moved the component A. As described before, local ad-hoc layout suggests and applies a layout modification horizontally aligning A and B whereas components C and D are too far apart. Since component B has just been changed, local ad-hoc layout is applied again for the set $C = \{A, B\}$ of components with a neighborhood $C_\mathrm{n} = \{A, B, C\}$. This time, C is horizontally aligned to A and B by local ad-hoc layout. The next iteration with the set $C = \{A, B, C\}$ has the neighborhood $C_\mathrm{n} = \{A, B, C\}$ again since D is located too far apart. No suggestions are computed, and the algorithm stops. The result is shown in Figure 7.3(d).
The example above shows that the layout engine cannot just add new pattern instances to the set of active pattern instances: The first iteration added a pattern instance $i_1 = \mathcal{I}(p_\mathrm{AlignH}, \{A, B\}, \{t\})$, and the second iteration $i_2 = \mathcal{I}(p_\mathrm{AlignH}, \{A, B, C\}, \{t\})$ where p_AlignH indicates the horizontal alignment pattern. Pattern instance i_2 apparently includes i_1. The layout engine, therefore, automatically combines pattern instances where possible, as already described in Chapter 6.

7.2.4 Discussion

GAL is quite powerful, but produces unpredictable layout "improvements". LAL improves on this aspect, but only enables quite small layout improvements. LAL/P turned out to be a good compromise between predictability and power. At the moment, LAL/P is integrated in the system.
A comparison of GAL, LAL and LAL/P in terms of performance will be given in Chapter 9.

At the moment, components that are in a "small neighborhood" of the selected components are chosen as input for the computation of the LAL and the LAL/P. As an alternative, components that are (completely) visible from one of the selected components, or components that are near a certain area, such as a horizontal line, in case of the horizontal alignment (top) pattern, could be chosen.

7.3 Future Work

In the following, future work concerning layout suggestions and ad-hoc layout is discussed.

7.3.1 Distinction of Pattern Instances

At the moment, all pattern instances are visualized in the diagram as well as in the list at the bottom right of the editor. The current visualization does not distinguish pattern instances that were created by the user himself or herself and the ones that were created in the course of automatic ad-hoc layout. A more sophisticated visualization could allow the user to distinguish these different types of pattern instances. E.g. the two types could be distinguished by using a different text color in the list at the bottom right of the editor.

7.3.2 Partial Ad-hoc Layout

Currently, ad-hoc layout is enabled for all patterns at once. To improve flexibility, one could also allow the user to choose the layout pattern(s), for which ad-hoc layout is automatically computed.

7.3.3 Preview of Ad-hoc Layout

Instead of applying layout patterns and updating the diagram, the layout improvement could be visualized in the diagram first. Afterwards, the user could decide whether or not he or she wants to carry out these modifications. For the visualization, two variants came to mind: Firstly, the updated diagram could be shown in light gray on top of the diagram. Secondly, the new pattern instances could be visualized in the diagram, as it is done for user-controlled layout patterns, without updating the diagram itself. The second variant is similar to the way it is done in tools like POWERPOINT or VISIO.

7.4 Summary

In this chapter, the concept of layout suggestions and the concept of ad-hoc layout were described, two features that are based on the idea of user-controlled layout behavior. The concept of layout suggestions was discussed in Section 7.1. In Section 7.2, the concept of ad-hoc layout was described. In Section 7.3, future work was outlined.
The features presented so far are only a small subset of imaginable features that could be created based on the layout approach. Some more ideas will be sketched in Chapter 10.

Chapter 8

Examples of Layout Patterns

In this chapter, several layout patterns are described, and their integration in diagram editors is discussed. In Section 8.1, several layout patterns are described in detail. In Section 8.2, it is discussed how these layout patterns are integrated in the editors described in Chapter 3.

8.1 Examples of Layout Patterns

The functionality of several layout patterns was already described in Chapter 3. Now, details of the specification of these layout patterns will be given. As shown in Figure 8.1, a pattern gets a set (*comps*) of components as input. In addition, a pattern may be available in several variants, and for that purpose, a set (*opts*) of options is added as input. A pattern has a set of associated p-constraints, and each p-constraint has a predicate and a set of rules. Predicates as well as rules are defined on the basis of a set ($vars$) of variables. Patterns and p-constraints are colored blue, predicates are colored green, and rules are colored yellow. In the following visualizations, the blue rounded rectangle on the right side is omitted. Predicates as well as rules are sorted from left to right. This sorting defines an order on the predicates and on the rules, and potentially influences the layout that is computed, as already described in Chapter 5.
Language-independent patterns are distinguished from language-dependent patterns. The main difference is that the layout pattern is specified on top of the PM in the first case, and on top of the LM in the second case. The "structure" of the set (*comps*) of components is defined by the associated model, hence either the PM or the LM. Patterns that belong to the first category are described next, and patterns that belong to the second category are described thereafter.

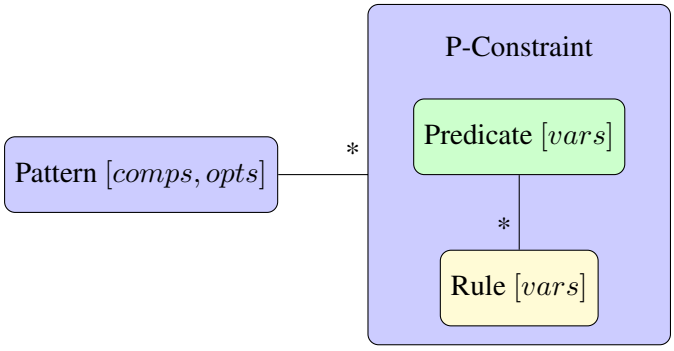


Figure 8.1: Pattern Structure

8.1.1 Introductory Example

In Figure 8.2, a diagram is shown that consists of the three classes A, B and C. Internally, this diagram is represented by an LMM instance. The equal height pattern is applied to the three components A, B and C. As a consequence, two (atomic) instances of the Elems PMM are created, one for the components A and B, and one for the components B and C. Furthermore, two (atomic) pattern instances are created, namely $\mathcal{I}(p_{\text{EqH}}, \{A, B\})$ and $\mathcal{I}(p_{\text{EqH}}, \{B, C\})$. The equal height pattern is a language-independent pattern, and hence, the "structure" of the set (*comps*) of components of each pattern instance is defined by the associated Elems PM.

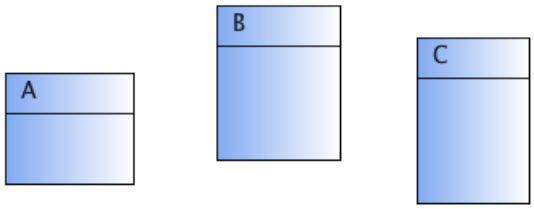


Figure 8.2: Sample Diagram

The diagram, the LM, the two Elem PMs and the two equal height pattern instances are shown in Figure 8.3. The orange arrows indicate the correlation between these artifacts.

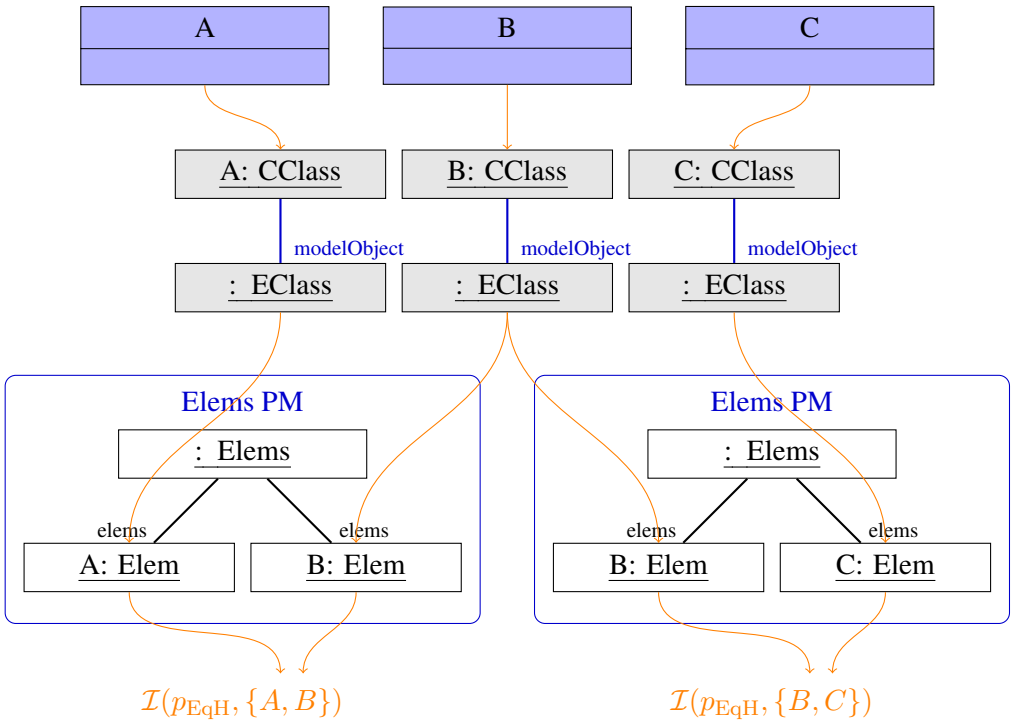
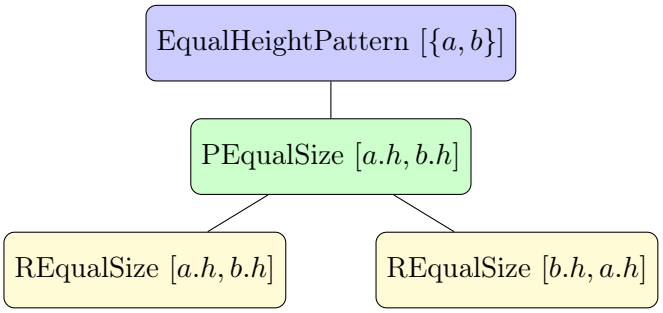


Figure 8.3: Introductory Example

The equal height pattern ensures that each component in a set of components has the same height. It gets two components as input, which are named A and B.

The pattern has one associated predicate and one associated rule that is instantiated twice, which either updates the height of the first or of the second component. The predicate *PEqualSize* $[a.h, b.h]$ is not fulfilled if $a.h = b.h$ does not hold. If applied, the rule *REqualSize* $[a.h, b.h]$ performs $a.h := b.h$, and the rule *REqualSize* $[b.h, a.h]$ performs $b.h := a.h$.



8.1.2 Running Example

If not mentioned otherwise, the example shown in Figure 8.4 serves as the running example in the following sections. The diagram consists of the nodes A, B, C and D.

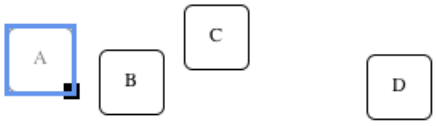


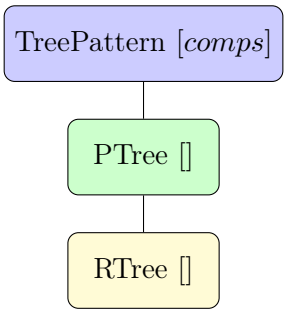
Figure 8.4: Sample Diagram

8.1.3 Language-independent Patterns

In the following, several language-independent layout patterns are described. First, graph-based patterns are described, then constraint-based patterns are depicted, and finally rule-based patterns are described. As already mentioned in Chapter 4, rule-based patterns and constraint-based patterns are usually defined in an atomic fashion, whereas graph-based patterns are not.

Tree Layout Pattern

The tree layout pattern applies a tree layout algorithm to a set of components. For that purpose, an external graph drawing algorithm is integrated.
The pattern gets the set of components *comps*, the pattern is applied to, as input. It has one associated predicate and one associated rule that applies the graph drawing algorithm.



For all components, one pattern instance is created. For the example shown in Figure 8.4, an instance is created for the following set of components:

- $\{A, B, C, D\}$

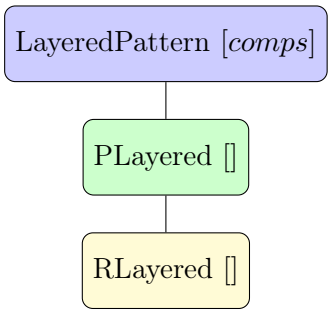
The predicate *PTree* is not fulfilled if the graph drawing algorithm would modify the diagram. In order to decide this, the graph drawing algorithm is executed.
The rule *RTree* performs the following (cf. Listing 8.1): First, a graph is created for the set of components *comps*. Based on this graph, the external graph drawing algorithm computes the layout. The resulting layout is then moved to the "correct" position via translation. The "correct" position is determined by the changed component *A* that triggered the execution of the layout algorithm: This component remains at the position the user has moved it to.

```
proc apply(A, comps)
    graph = createGraph(comps)
    layout = doTreeLayout(graph)
    translate(layout, A)
end
```

Listing 8.1: Graph Drawing Algorithm

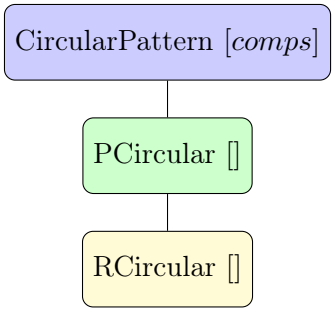
Layered Layout Pattern

The layered layout pattern applies a layered layout algorithm to a set of components. For that purpose, an external graph drawing algorithm is integrated. This pattern is defined analogously to the tree layout pattern.



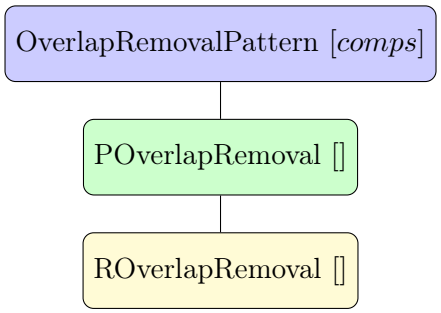
Circular Layout Pattern

The circular layout pattern applies a circular layout algorithm to a set of components. For that purpose, an external graph drawing algorithm is integrated. This pattern is also defined analogously to the tree layout pattern.



Node Overlap Removal Pattern

The node overlap removal pattern avoids the overlap of components. It may either be implemented by a force-directed layout strategy, or by a constraint-based approach. In the following, the first variant will be described. Later on, the second variant will be described.



For all components, one pattern instance is created. Hence, for the example shown in Figure 8.4, an instance is created for the following set of components:

- $\{A, B, C, D\}$

The predicate *POverlapRemoval* is not fulfilled if component A (the component that was modified by the user) overlaps with one of the components of the set *comps* of components.
The rule*ROverlapRemoval* performs the following (cf. Listing 8.2): First, all overlapping components of the set *comps* of components are determined. For this set of overlapping components, a graph is created. Based on this graph, the external graph drawing algorithm computes the layout. This procedure is continued until no more components overlap. As a last step, the resulting layout is moved to the "correct" position via translation.

```
proc apply(A, comps)
    list = identifyOverlappingComps(comps)
    while list ≠ ∅ do
      graph = createGraph(list)
      layout += doForceDirectedLayout(graph)
      overlapComps = identifyOverlappingComps(comps)
  end do
  translate(layout, A)
end
```

Listing 8.2: Graph Drawing Algorithm for Node Overlap Removal

The force-directed layout algorithm is solely applied to nodes instead of applying it to nodes as well as edges. This procedure was chosen, as user experiments showed that this variant provided a more natural "feeling" while interacting with an editor.
The force-directed layout algorithm is applied to nodes that overlap instead of to all nodes. This has the benefit that performance is improved. Furthermore, an unnecessary expansion of the diagram is avoided. The drawback is that there might be cases where it is not possible to compute a valid layout via this procedure. But as user experiments showed, this almost never happens.
A force-directed layout algorithm can be used as described above if it fulfills the following criterion: The force-directed layout algorithm allows for an incremental (step-by-step) application. This way, it is possible to execute the force-directed layout algorithm step-wise, as described above.
There is a huge variety of implementations available that could potentially be used. The following third-party force-directed layout algorithms were considered being integrated, because they are widely used and because they are implemented in Java:

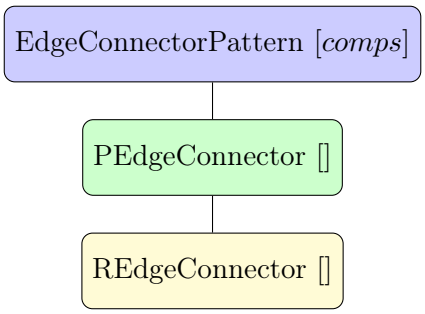
- A force-directed layout algorithm contained in the Jung library [90]. The algorithm provides a special mode for node overlap removal, which allows for an incremental application of the algorithm.

- A force-directed layout algorithm contained in the yFiles library [117]. The algorithm does not provide a special mode for node overlap removal.

The first force-directed layout algorithm was chosen, as the "special mode for node overlap removal" fulfills the criterion mentioned earlier.

Edge Connector Pattern

The edge connector pattern ensures that edges are correctly connected to their corresponding nodes. For that purpose, an external graph drawing algorithm is integrated. This pattern is also defined analogously to the tree layout pattern.

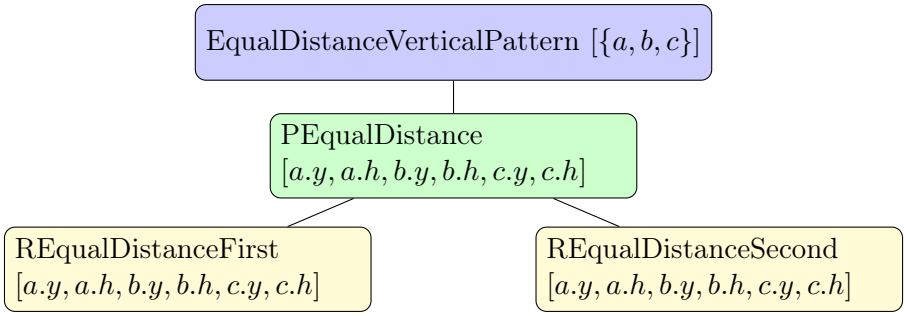


Up to now, only one pattern was created that focuses on edges, namely the edge connector pattern. Of course it is possible to create other patterns that take care of edges.

Equal Distance Pattern

The equal vertical (horizontal) distance pattern preserves an equal vertical (horizontal) distance between a set of components. Two variants of this pattern exist - a rule-based version and a constraint-based version. Both variants of the equal vertical distance pattern will be described in the following.

Rule-based Version The first variant has one associated predicate and two associated rules.



The components are sorted by their y-position, as an equal vertical distance is established. For three components that are next to each other, one pattern instance is created. Hence, for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A, B, C\}$, $\{B, C, D\}$

The predicate *PEqualDistance* is not fulfilled if the following does not hold:

$$b.y - a.y - a.h = c.y - b.y - b.h$$

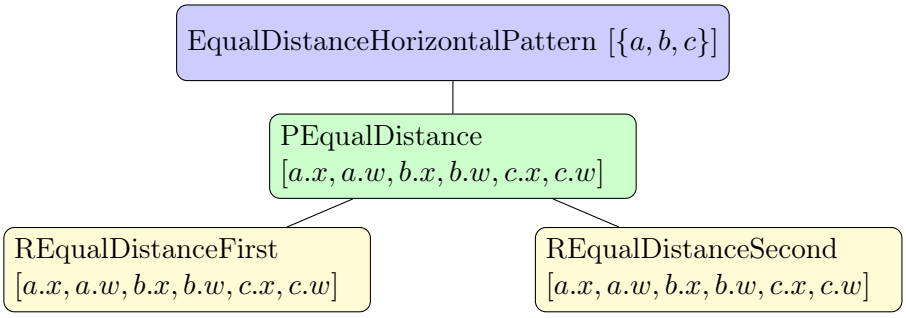
The rule *REqualDistanceFirst* performs the following:

$$dist := b.y - a.y - a.h$$
$$c.y := b.y + b.h + dist$$

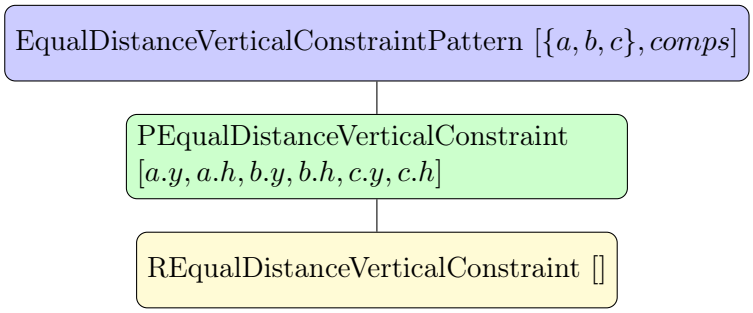
The rule *REqualDistanceSecond* performs the following:

$$dist := c.y - b.y - b.h$$
$$a.y := b.y - dist - a.h$$

The equal horizontal distance pattern is defined analogously to the equal vertical distance pattern. It reuses the predicate *PEqualDistance* and the rules *REqualDistanceFirst* and *REqualDistanceSecond* for the attributes $a.x$, $a.w$, $b.x$, $b.w$, $c.x$ and $c.w$.



Constraint-based Version The second variant has one associated predicate and one associated rule that calls the constraint solver.



The predicate *PEqualDistanceVerticalConstraint* is not fulfilled if the following does not hold:

$$b.y - a.y - a.h = c.y - b.y - b.h$$

The rule *REqualDistanceVerticalConstraint* performs the following: For each component $c_i \in comps$, the variables $c_i.y$ and $c_i.h$ are created.
In addition, the following constraints are created:
The first constraint fixes the ordering of the components. This predicate is created for each tuple of components c_i and c_{i+1}, where $c_i, c_{i+1} \in comps$ and $c_i.y < c_{i+1}.y$ and $\neg\exists c_k \in comps : c_i.y < c_k < c_{i+1}.y$:

$$c_{i+1}.y - c_i.y > 20$$

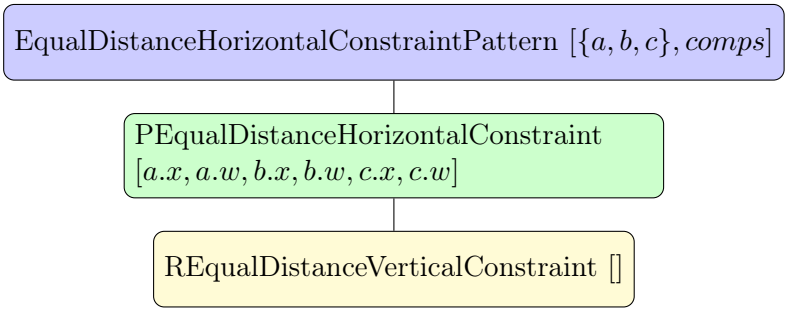
The second constraint ensures an equal distance. This constraint is created for each triple of components c_i, c_{i+1} and c_{i+2}, where $c_i, c_{i+1}, c_{i+2} \in comps$ and $c_i.y < c_{i+1}.y < c_{i+2}.y$ and $\neg\exists c_k \in comps : c_i.y < c_k < c_{i+1}.y$ and $\neg\exists c_k \in comps : c_{i+1}.y < c_k < c_{i+2}.y$:

$$c_{i+1}.y - c_i.y - c_i.h = c_{i+2}.y - c_{i+1}.y - c_{i+1}.h$$

This constraint is reformulated as follows:

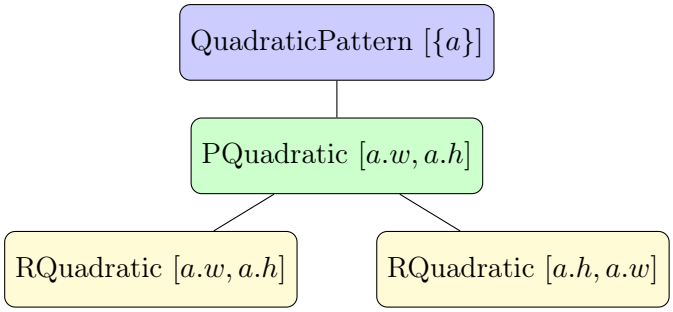
$$2c_{i+1}.y - c_{i+2}.y - c_i.y = c_i.h - c_{i+1}.h$$

The constraint solver only updates the variables on the left side of the equation, namely $c_i.y$, $c_{i+1}.y$ and $c_{i+2}.y$. It does not update the variables on the right side, namely $c_i.h$, $c_{i+1}.h$ and $c_{i+2}.h$.
The constraint-based version of the equal horizontal distance pattern is defined analogously to the constraint-based version of the equal vertical distance pattern. For that purpose, a new predicate *PEqualDistanceHorizontalConstraint* and a new rule *REqualDistanceHorizontalConstraint* are defined for the attributes $a.x$, $a.w$, $b.x$, $b.w$, $c.x$ and $c.w$. The predicate *PEqualDistanceVerticalConstraint* and the rule *REqualDistanceVerticalConstraint* cannot be reused, because components have to be ordered differently: Instead of ordering them by their y-coordinate, they now have to be ordered by their x-coordintate.



Quadratic Component Pattern

The quadratic component pattern ensures that a component has the same height and width all the time.
It has one associated predicate and one associated rule, which is instantiated twice. The rule *RQuadratic* tries to update the width or the height of the component.



For each component, one pattern instance is created. Hence, for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$

The predicate *PQuadratic* is not fulfilled if the following does not hold:

$$a.w = a.h$$

The first instance of the rule *RQuadratic* performs the following:

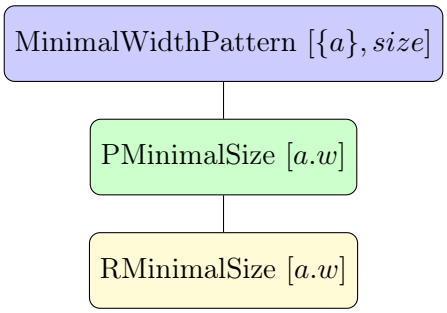
$$a.h := a.w$$

Analogously, the second instance performs the following:

$$a.w := a.h$$

Minimal Size Component Pattern

The minimal size component pattern makes sure that a component is not smaller than a minimal size. In case of the *MinimalWidthPattern*, the parameter *size* defines the minimal width of a component, whereas in case of the *MinimalHeightPattern*, the parameter *size* defines the minimal height of a component.
The minimal width pattern has one associated predicate and one associated rule.



For each component, one pattern instance is created. Hence, for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$

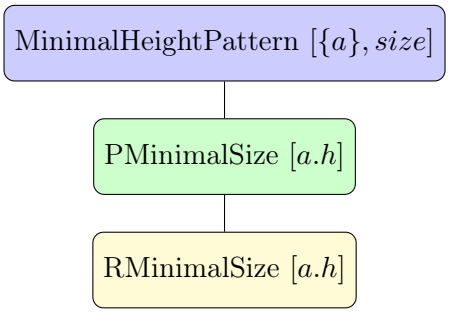
The predicate *PMinimalSize* is not fulfilled if the following does not hold:

$$a.w > size$$

The rule *RMinimalSize* performs the following:

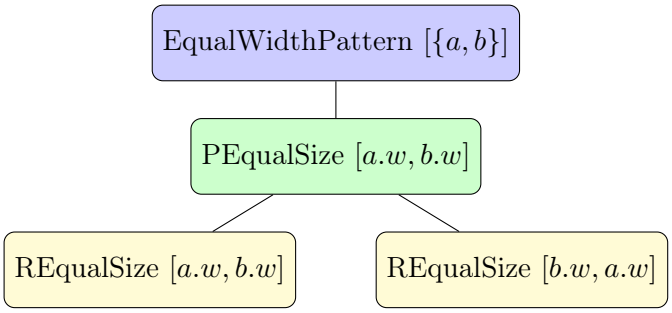
$$a.w := size$$

The minimal height pattern is defined analogously to the minimal width pattern. It reuses the predicate *PMinimalSize* and the rule *RMinimalSize* for the attribute $a.h$.



Equal Size Pattern

The equal size pattern makes sure that each component in a set of components has the same height (equal height pattern) or width (equal width pattern).
The equal width pattern has one associated predicate and one associated rule that is instantiated twice, which either updates the width of the first or of the second component.



For each pair of components, one pattern instance is created. Hence for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A, B\}$, $\{B, C\}$, $\{C, D\}$

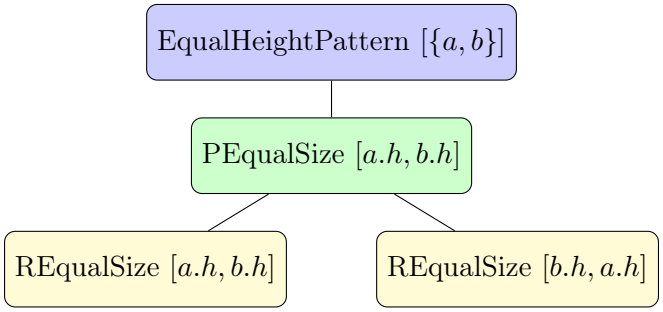
The predicate *PEqualSize* is not fulfilled if the following does not hold:

$$a.w = b.w$$

The two instances of the rule *REqualSize* perform the following:

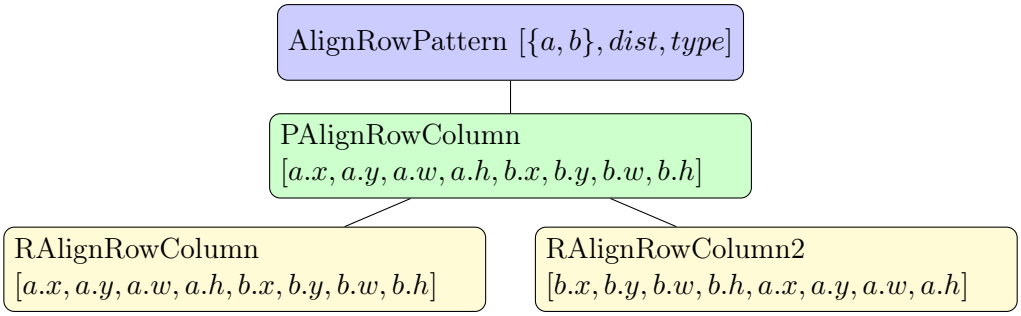
$$a.w := b.w$$
$$b.w := a.w$$

This rule is instantiated twice, and hence either the width of the first component is updated, or the width of the second component is updated.
The equal height pattern is defined analogously to the equal width pattern. It reuses the predicate *PEqualSize* and the rule *REqualSize* for the attributes $a.h$ and $b.h$.



Align in a Row / Column Pattern

The align in a row/column pattern orders a set of components in a row or in a column. The parameter *dist* defines the horizontal or vertical distance between components. In case of the *AlignRowPattern*, the parameter *type* defines, whether the components are aligned at the top, at the center or at the bottom. In case of the *AlignColumnPattern*, the parameter *type* defines, whether the components are aligned at the left side, at the center or at the right side. In the following, the *AlignRowPattern* is described, where $type = center$.
The pattern has one associated predicate and two associated rules, which either update the first component or the second component.



The components are sorted by their x-position, as they are aligned in a row. For each pair of components, one pattern instance is created. Hence for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A, B\}$, $\{B, C\}$, $\{C, D\}$

The predicate *PAlignRowColumn* is not fulfilled if the following does not hold:

$$(a.y + a.h/2 = b.y + b.h/2) \wedge (b.x = a.x + a.w + dist)$$

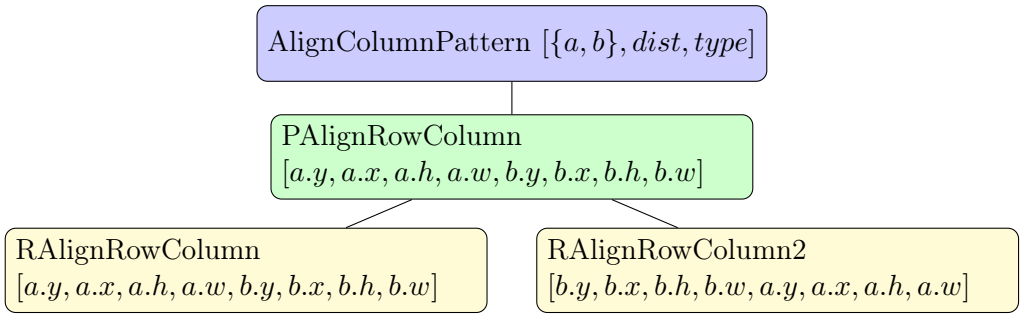
The rule *RAlignRowColumn* performs the following:

$$b.y := a.y + a.h/2 - b.h/2$$
$$b.x := a.x + a.w + dist$$

The rule *RAlignRowColumn2* performs the following:

$$a.y := b.y - a.h/2 + b.h/2$$
$$a.x := b.x - a.w - dist$$

The align in a column pattern is defined analogously to the align in a row pattern. It reuses the predicate *PAlignRowColumn* and the rules *RAlignRowColumn* and *RAlignRowColumn2*.

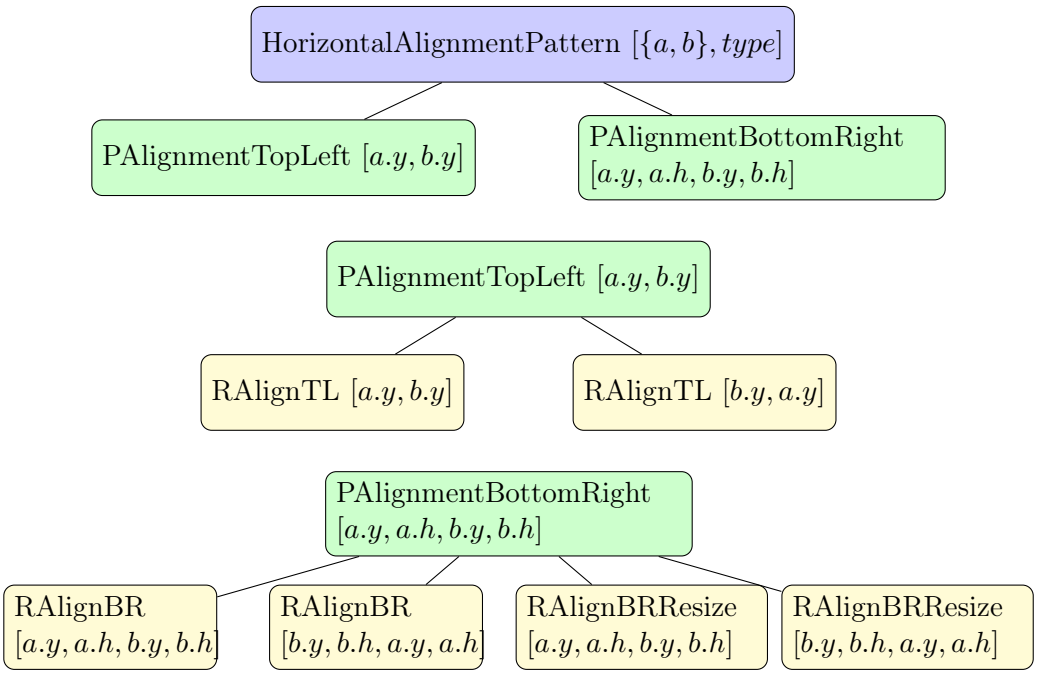


The align in a row pattern and the align in a column pattern are defined in such a way that the components are centered. In case of the align in a row pattern, the components could also be aligned at the left or at the right side. In case of the align in a column pattern, the components could also be aligned at the top or at the bottom. These variants can be defined analogously.

Alignment Pattern

The alignment pattern aligns components either vertically or horizontally. In case of the *HorizontalAlignmentPattern*, the parameter *type* defines whether the components are aligned at the top, in the center or at the bottom. In case of the *VerticalAlignmentPattern*, the parameter *type* defines, whether the components are aligned at the left side, at the center or at the right side. In the following, the *HorizontalAlignmentPattern* is described, where $type = top$ and also where $type = bottom$.
The horizontal alignment pattern has two associated predicates. The predicate *PAlignmentTopLeft* aligns the components at the top, and the predicate *PAlignmentBottomRight* aligns the components at the bottom.

The predicate *PAlignmentTopLeft* has one associated rule, which is instantiated twice. It updates the y-position of component a or b. The predicate *PAlignmentBottomRight* has two associated rules, both instantiated twice. The rule *RAlignBR* updates the y-position of component a or b, and the rule *RAlignBRResize* updates the height of component a or b.



For each pair of components, one pattern instance is created. Hence, for the example shown in Figure 8.4, instances are created for the following sets of components:

- $\{A, B\}$, $\{B, C\}$, $\{C, D\}$

The predicate *PAlignmentTopLeft* is not fulfilled if the following does not hold:

$$a.y = b.y$$

The predicate *PAlignmentBottomRight* is not fulfilled if the following does not hold:

$$a.y + a.h = b.y + b.h$$

The two instances of the rule *RAlignTL* perform the following:

$$b.y := a.y$$
$$a.y := b.y$$

The two instances of the rule *RAlignBR* perform the following:

$$b.y := a.y + a.h - b.h$$
$$a.y := b.y + b.h - a.h$$

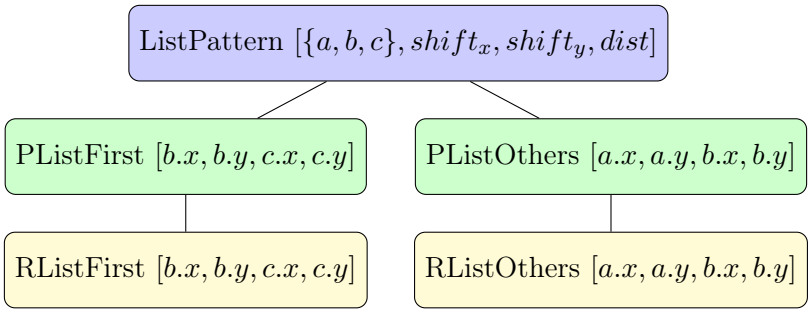
The two instances of the rule *RAlignBRResize* perform the following:

$$b.h := a.y + a.h - b.y$$
$$a.h := b.y + b.h - a.y$$

The vertical alignment pattern is defined analogously to the horizontal alignment pattern. It reuses the predicates *PAlignmentTopLeft* and *PAlignmentBottomRight* for the attributes $a.x$, $a.w$, $b.x$ and $b.w$.
The horizontal alignment pattern and the vertical alignment pattern are defined in such a way that the components are aligned at the top-bottom or at the left-right. As an alternative, the components could also be centered. This variant can be defined analogously.

List Pattern

The list pattern arranges components as a vertical list. The parameters $shift_x$ and $shift_y$ define a spacing between list container and list elements. The parameter *dist* defines the vertical distance between list elements. In its current form, the list pattern is defined in a way that the list elements are aligned vertically at the left side. Here, once again, other variants could be defined.
The pattern has two associated predicates and two associated rules.



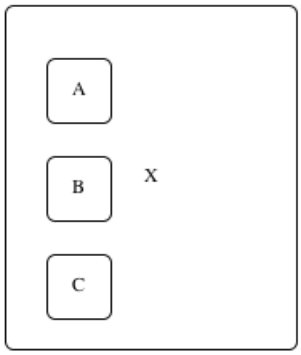


Figure 8.5: Sample Diagram

The list elements are sorted by their y-position, as they are aligned in a column. For each pair of list elements together with the list container, one pattern instance is created. Hence, for the example shown in Figure 8.5, instances are created for the following sets of components:

- $\{-, A, X\}$, $\{A, B, X\}$, $\{B, C, X\}$

The predicate *PListFirst* is initialized only for the first element of the list ($\{A, X\}$). The predicate is not fulfilled if the following does not hold:

$$(a.x = c.x + shift_x) \wedge (a.y = c.y + shift_y)$$

The rule $RListFirst$ performs the following:

$$a.x := c.x + shift_x$$
$$a.y := c.y + shift_y$$

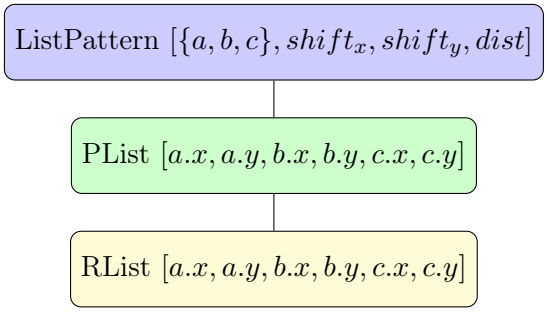
The predicate *PListOthers* is initialized for all pairs of elements in the list ($\{A, B\}$, $\{B, C\}$, $\{C, D\}$). The predicate is not fulfilled if the following does not hold:

$$(b.x = a.x) \wedge (b.y = a.y + dist)$$

The rule *RListOthers* performs the following:

$$b.x := a.x$$
$$b.y := a.y + dist$$

Alternative The list pattern could also be defined via one predicate and one rule. This predicate is (again) initialized for all sets of components ($\{-, A, X\}$, $\{A, B, X\}$, $\{B, C, X\}$). Internally the same actions are performed, depending on whether or not the first component in the list is $-$.



Performance Comparing these two variants, the constraint network of the first variant is less connected than the one of the second variant, as can be seen in Figures 8.6 and 8.7. C_1 is the predicate for the set of components $\{-, A, X\}$, C_2 is the predicate for the set of components $\{A, B, X\}$, and C_3 is the predicate for the set of components $\{B, C, X\}$.
E.g. the user moves component X. For the first variant, this means that component A needs to be updated in the first iteration, whereas for the second variant, the components A, B, and C need to be updated in the first iteration. As a consequence, the number of cases the propagation algorithm has to take into account in the first iteration for the second variant is much higher than for the first variant. The cases that have to be taken into account in each iteration are visualized in Figures 8.8 and 8.9.

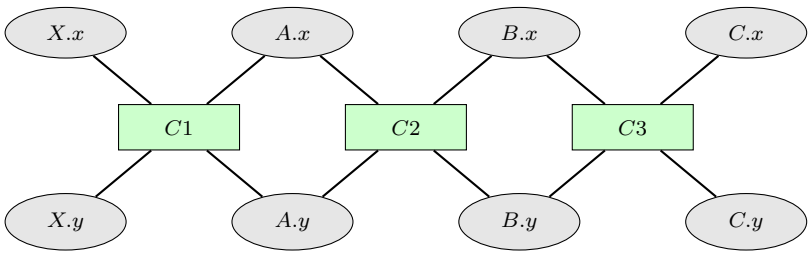


Figure 8.6: Constraint Network of the Example (Variant 1)

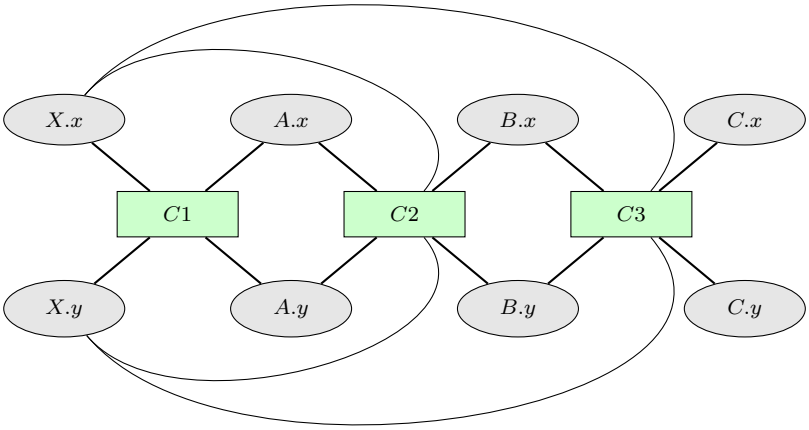


Figure 8.7: Constraint Network of the Example (Variant 2)

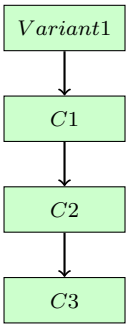


Figure 8.8: Propagation (Variant 1)

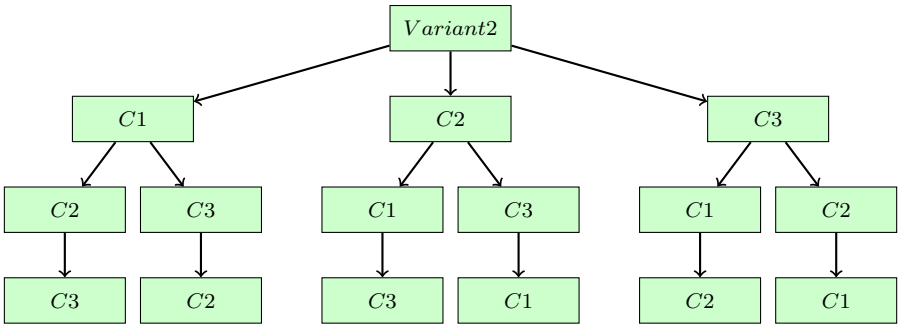
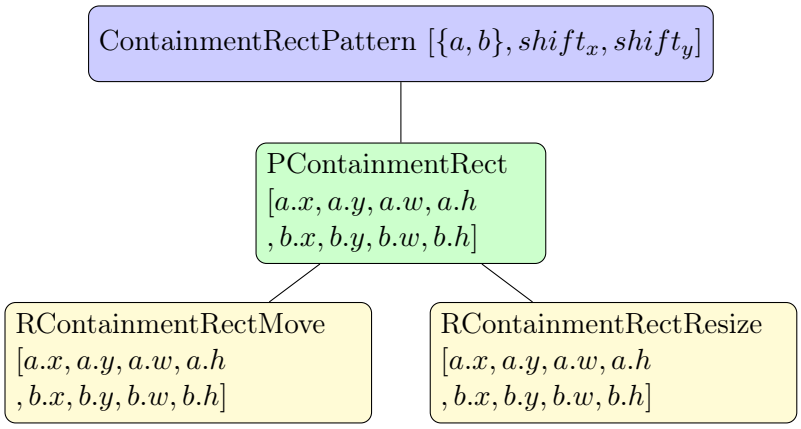


Figure 8.9: Propagation (Variant 2)

Rectangular Containment Pattern

The rectangular containment pattern is responsible for the correct nesting of components. In case the nesting is not correct, either the container or the contained components may be changed. Each component may either be moved or resized. The parameters $shift_x$ and $shift_y$ define the spacing between the container and the contained element.
The pattern has one associated predicate and two associated rules. The rule *RContainmentMove* updates the position of the component (x-position and (or) y-position), and the rule *RContainmentResize* updates the size of the component (width and (or) height).



For each pair of components (contained component & container), one pattern instance is created. Hence for the example shown in Figure 8.10, instances are created for the following sets of components:

- $\{A, X\}$, $\{B, X\}$, $\{C, X\}$, $\{D, X\}$

The predicate *PContainmentRect* is not fulfilled if the following does not hold:

$$(a.x > b.x) \wedge (a.x + a.w < b.x + b.w)$$
$$\wedge (a.y > b.y) \wedge (a.y + a.h < b.y + b.h)$$

The rule *RContainmentRectMove* moves the inner component. It performs the following:

if $(a.x + a.w > b.x + b.w)$
  $a.x := b.x + b.w - a.w$
else if $(a.x < b.x)$
  $a.x := b.x$
end if
if $(a.y + a.h > b.y + b.h)$
  $a.y := b.y + b.h - a.h$
else if $(a.y < b.y)$
  $a.y := b.y$
end if

The rule *RContainmentRectResize* resizes the outer component. It performs the following:

if $(a.x + a.w > b.x + b.w)$
  $b.w := a.x + a.w - b.x$
else if $(a.x < b.x)$
  $b.x := a.x$
  $b.w := b.w + (a.x - b.x)$
end if

Note that $shift_x$ and $shift_y$ were omitted in the predicates and rules in order to improve readability.

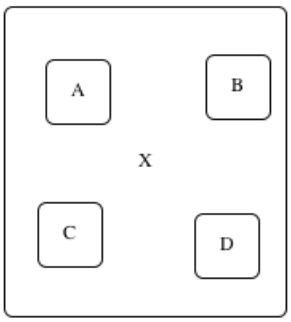


Figure 8.10: Sample Diagram

8.1.4 Language-dependent Patterns

In the following, one language-dependent layout pattern is described, namely the circular containment pattern, which is specifically designed for the VEX editor.

Circular Containment Pattern

The circular containment pattern was specifically designed for VEX diagrams. It is responsible for the correct nesting of components that have a circular shape. In case the nesting is incorrect, either the container or the contained components may be changed, by moving or resizing them.

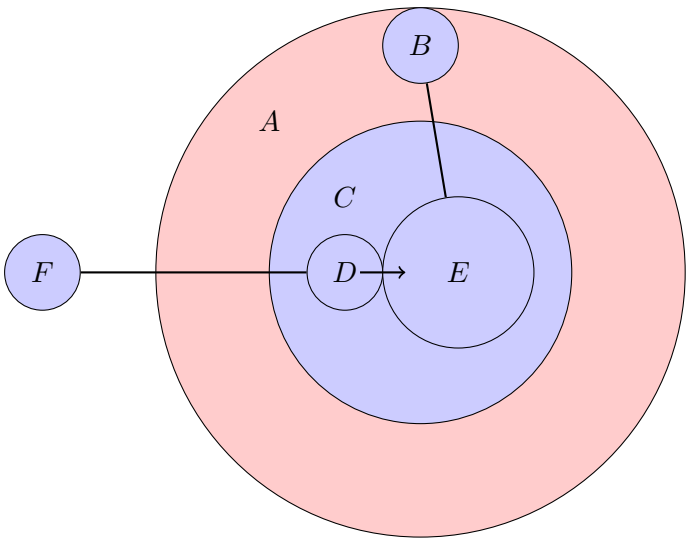
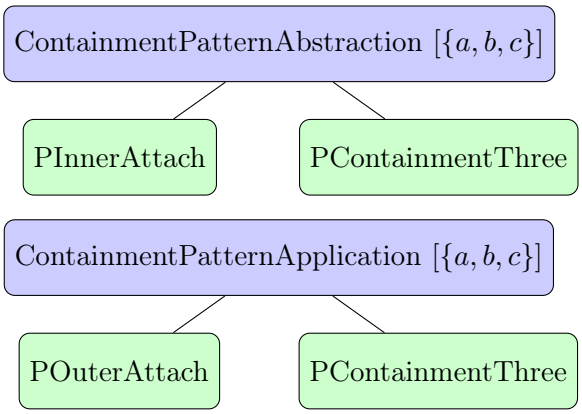


Figure 8.11: Sample Diagram

The pattern responsible for a correct nesting of abstraction has the two associated predicates *PInnerAttach* and *PContainmentThree*. The pattern responsible for a correct nesting of function application has the two associated predicates *POuterAttach* and *PContainmentThree*.
For each variable abstraction, one instance of *ContainmentPatternAbstraction* is created, and for each function application, one instance of *ContainmentPatternApplication* is created.



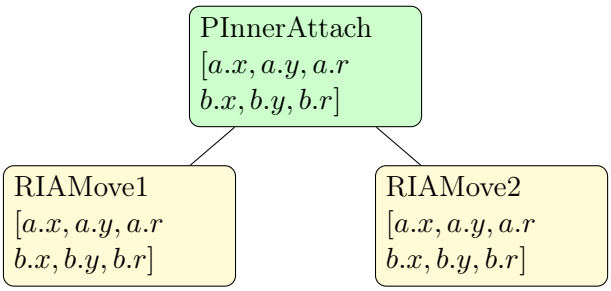
Hence, for the example shown in Figure 8.11, one instance of *Containment-PatternAbstraction* is created for the following set of components:

- $\{A, B, C\}$

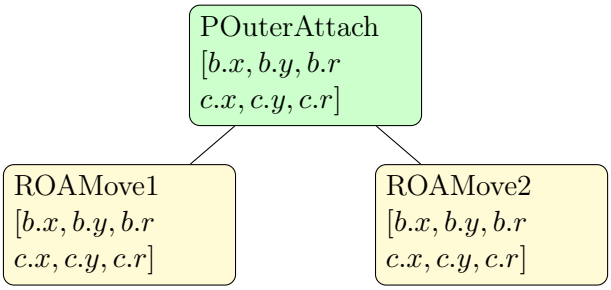
Furthermore, one instance of *ContainmentPatternApplication* is created for the following set of components:

- $\{C, D, E\}$

The predicate *PInnerAttach* has two associated rules. The rule *RIAMove1* updates the position of a, the outer circle (x-position and (or) y-position). The rule *RIAMove2* updates the position of b, the inner circle (x-position and (or) y-position).

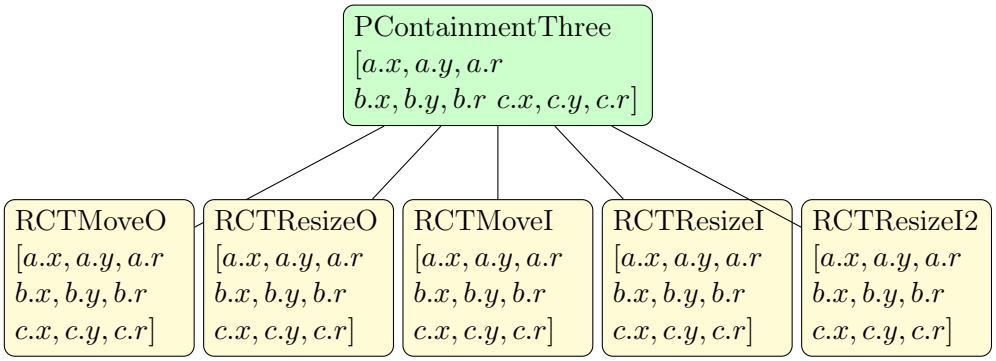


The predicate *POuterAttach* also has two associated rules. The rule *ROA-Move1* updates the position of b, the first circle (x-position and (or) y-position). The rule *ROAMove2* updates the position of c, the second circle (x-position and (or) y-position).



The predicate *PContainmentThree* has five associated rules. The rule *RCTMoveO* updates the position of a, the outer circle (x-position and (or) y-position). The rule *RCTMoveI* updates the position of c, the inner circle (x-position and (or) y-position). (It does not update the position of the third component.) The rule *RCTResizeO* updates the size of a, the outer

component (width and (or) height). The rule *RCTResizeI* updates the size of c, the inner component (width and (or) height). The rule *RCTResizeI2* updates the size of b, the second inner component, which is either attached to the border of the outer circle or to the border of the inner circle (width and (or) height).



8.1.5 Node Overlap Removal Pattern, Freeze Component Pattern & Snap to Grid Pattern

In this section, details are given about three rather unusual layout patterns, namely the node overlap removal pattern, the freeze component pattern and the snap to grid pattern.

Node Overlap Removal Pattern

In the context of the design of the node overlap removal pattern, two variants are imaginable:

- The first one is that it is not possible to move a component on top of another component. This kind of pattern can easily be defined with the help of a collision detection algorithm.
- The other one is to move away components that would overlap with the component in focus.

As already described, it was decided to integrate the second variant. The reason for this decision was that the behavior of this variant is similar to the other layout patterns: User changes are performed if the layout engine is able to compute a valid layout. They are not suppressed or tampered by the layout pattern. A drawback of this variant is that there may be cases where it interrupts users in their productive work with the editor. E.g. suppose

that the user wants to move a component from one side of the diagram to the other side of the diagram without changing the rest of the diagram. Then he or she cannot simply move the component across the rest of the diagram, because the other components would be moved away. As a solution, the user can either move the component around the diagram, or he or she can (temporarily) deactivate the node overlap removal pattern.
The feature may either be implemented via a graph drawing algorithm or via a constraint-based algorithm. The first variant was already described. The second variant will be described in the following.

Constraint-based Version The definition of constraints for the constraint-based algorithm is straightforward. For each pair of components that are next to each other horizontally, the following predicate is added:

$$(a.x + a.w < b.x) \vee (b.x + b.w < a.x)$$

For each pair of components that are next to each other vertically, the following predicate is added:

$$(a.y + a.h < b.y) \vee (b.y + b.h < a.y)$$

Related Work Several approaches for node overlap removal exist. Some related approaches are the following: In [60], an approach for node overlap removal is presented that uses spring algorithms. In [28], an algorithm for fast node overlap removal is presented. In their approach, separation constraints are generated first. Thereafter, an algorithm tries to find a solution to this constraint satisfaction problem. The algorithm aims at finding a solution that modifies the graph as few as possible. In [46], an efficient algorithm for node overlap removal is presented. This algorithm uses a proximity stress model. In [47], an efficient, proximity-preserving algorithm for node overlap removal is presented. This algorithm performs well at maintaining the graph's shape. All of the above-mentioned algorithms are essentially some sort of force-directed layout algorithm or a set of declarative constraints together with a constraint solver. Hence, they are either similar to the first variant presented, or to the second one presented.

Freeze Component Pattern

The freeze component pattern *freezes* the position and the shape of a certain component. Therefore, all values of the variables are frozen. This pattern

is quite simple, but worth mentioning, because it provides a very useful functionality. For a node a, the following predicates are added:

$$freeze(a.x) \wedge freeze(a.y) \wedge freeze(a.h) \wedge freeze(a.w)$$

For an edge e, the following predicates are added:

$$freeze(e.x1) \wedge freeze(e.y1) \wedge freeze(e.x2) \wedge freeze(e.y2)$$

Snap to Grid Pattern

The snap to grid pattern enforces components being "snapped" to a grid. This pattern contradicts the rule "a component that is currently modified by the user should not be modified by the layout engine".

8.1.6 Guidelines for the Creation of Layout Patterns

In summary, the following guidelines can be deduced from the creation of the layout patterns described in this Chapter:

- Patterns, predicates and rules should be reused, where possible.
- Parametrization should be used, in order to reduce the number of patterns, predicates and rules.
- The structure of the constraint network is crucial in terms of performance.

8.2 Integration of Layout Patterns in an Editor

The patterns shown in Table 8.1 are integrated in the four editors that were described in Chapter 3. In the next paragraphs, some details about their integration are given.

8.2.1 Graph Editor

In the graph editor, most of the patterns were added. This language does not comprise any sort of list, and hence, the list pattern was not added. In the language, no containment hierarchy is available, and as a consequence, the rectangular containment pattern as well as the circular containment pattern were not added either. The other patterns were added straightforwardly.

Pattern	Graph	Class	GUI	VEX
Tree Layout		x		
Layered Layout	x			
Circular Layout	x			
Node Overlap Removal	x	x		
Edge Connector	x	x		x
Equal Horizontal Distance	x	x	x	
Equal Vertical Distance	x	x	x	
Quadratic Component	x		x	
Minimal Size Component	x	x	x	x
Equal Height	x	x	x	
Equal Width	x	x	x	
Align in a Row	x	x	x	
Align in a Column	x	x	x	
Horizontal Alignment	x	x	x	
Vertical Alignment	x	x	x	
List		x	x	
Rectangular Containment		x	x	
Circular Containment				x

Table 8.1: Patterns in Diagram Editors

8.2.2 Class Diagram Editor

In the class diagram editor, the layered layout pattern and the circular layout pattern were not added. It was also not reasonable to add the quadratic component pattern or the circular containment pattern. The other patterns were added as follows:

- The tree layout pattern may be applied to classes together with generalizations.[1]
- The node overlap removal pattern is automatically applied to classes as well as packages inside the same package.
- The edge connector pattern is automatically applied to classes, generalizations and associations.
- The equal horizontal distance pattern as well as the equal vertical distance pattern may be applied to packages and classes. Also a "mixed list" is possible here.

[1] In the class diagram editor, only single inheritance is allowed. Hence, the application of the tree layout pattern is always possible.

- The minimal size component pattern is automatically applied to classes as well as packages. For classes and packages, different sizes were defined. In addition, the parameters height and width have different minima, due to the "standard" shape of classes and packages.
- The equal height pattern as well as the equal width pattern may be applied to classes and packages.
- The align in a row pattern and the align in a column pattern may be applied to classes and packages.
- The horizontal alignment pattern and the vertical alignment pattern may be applied to classes and packages.
- The list pattern is automatically applied to the attributes inside a class.
- The rectangular containment pattern is automatically applied to classes and packages. A package may contain classes as well as packages.

8.2.3 GUI Forms Editor

In the GUI forms editor, no edges exist. Hence, the tree layout pattern, the layered layout pattern, the circular layout pattern and the edge connector pattern were not added. Besides, the node overlap removal pattern was not added, because such a behavior is not helpful in the context of GUI creation. As all GUI components have a rectangular shape, the circular containment pattern was not needed. The other patterns were added as follows:

- The equal horizontal distance pattern and the equal vertical distance pattern may be applied to all components.
- The quadratic component pattern may be applied to all components.
- The minimal size component pattern is automatically applied to all components.
- The equal height pattern and the equal width pattern may be applied to all components.
- The align in a row pattern and the align in a column pattern may be applied to all components.
- The horizontal alignment pattern as well as the vertical alignment pattern may be applied to all components.

- The list pattern may be applied to all components inside a frame or panel.
- The rectangular containment pattern is automatically applied to frames and panels that represent the containers, and arbitrary components that represent the container elements.

8.2.4 VEX Editor

In the VEX editor, most of the layout patterns are not useful, and hence were not added. The remaining patterns were added as follows:

- The edge connector pattern is automatically applied to circles together with lines. It is also automatically applied to circles together with arrows.
- The minimal size component pattern is automatically applied to circles.
- The circular containment pattern takes care of the correct nesting of circles. It is also automatically applied.

8.3 Summary

In this chapter, several layout patterns were described, and their integration in diagram editors was discussed. In Section 8.1, several layout patterns were described in detail. In Section 8.2, it was discussed how these layout patterns are integrated in the editors described in Chapter 3.

Chapter 9

Evaluation

In this chapter, an evaluation of the approach in terms of usability and in terms of performance is given. A user study, which aims at identifying layout patterns that are commonly "needed" in diagram editors, is described in Section 9.1. In Section 9.2, a performance evaluation is presented.

9.1 User Study

In a user study, students created several visual language editors, mainly focusing on the layout engine. This user study is outlined in [71]. A more detailed description can be found in [2].
The purpose of the user study was to examine the desired properties of a layout engine for visual language editors. We had the following expectations, which turned out to be true:

- Different visual language editors require similar layout patterns.
- The combination of graph drawing algorithms and other layout algorithms is reasonable.

9.1.1 Setup

The setup of the user study was as follows: Seven groups of students, consisting of two or three students each, were asked to use DiaMeta. First, each group had to create a visual language editor to get familiar with the system. Afterwards, each group had to define a layout algorithm for this visual language. They were asked to implement a standard layout algorithm, meaning a layout algorithm that was designed for graph visualization, following the

descriptions of [109]. They were also asked to adapt the algorithm to the special requirements of their visual language editor.
The visual languages were chosen with respect to two criteria: Visual languages were chosen that comprise graph-like as well as non-graph-like parts, and it was tried to cover a wide range of visual languages. The graph drawing algorithms were chosen by examining the visual languages and selecting the ones that fit well. The following "pairs" were chosen:

- mindmaps, tree layout
- business process models, layered layout
- class diagrams, edge routing
- reducer rules (a visual language for editor specification), force-directed layout
- organizational charts, incremental connector routing
- entity relationship diagrams, polyline drawings
- circuit diagrams, interactive orthogonal drawings

9.1.2 Results

To show some of the students' design decisions, three representative examples are described in the following:

- tree layout applied to mindmaps
- layered layout applied to business process models
- edge routing applied to class diagrams

For each layout algorithm, first, the layout behavior is listed that was defined via graph drawing algorithms. Then, the layout behavior is listed that was defined outside the graph drawing algorithms.

Mindmaps

A tree layout algorithm was applied to the obvious tree structure of a mindmap (Figure 9.1). The students implemented a circular and a layered layout strategy.

- Layout behavior defined via graph drawing algorithms: Nodes should stay near the position where the user has placed them. Besides, the different node shapes (e.g. a cloud) and sizes need to be considered.
- Layout behavior defined outside graph drawing algorithms: Lists (diamonds followed by text) are required to remain attached to their owner nodes, and the order of list entries should be preserved (cf. list pattern). Links between different branches need to stay attached and must be routed without crossing other nodes (cf. edge connector pattern).

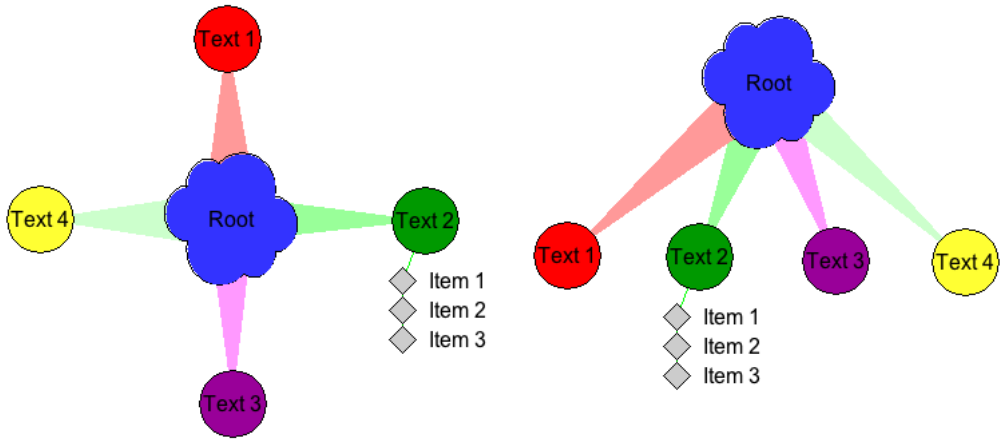


Figure 9.1: Mindmaps: Circular and Layered Layout Strategy

Business Process Models

A layered layout algorithm was applied to business process models (Figure 9.2). Here, flow objects serve as nodes and connecting objects as edges. To alter the drawing, the students have provided many options, e.g. horizontal or vertical alignment of components.

- Layout behavior defined via graph drawing algorithms: Changing the diagram should not result in objects being moved to a different layer.
- Layout behavior defined outside graph drawing algorithms: A special edge router is used to cope with nodes of different sizes (cf. edge connector pattern). Swimlanes allow for node nesting, which has to be preserved (cf. rectangular containment pattern). The layout engine should further maintain the vertical or horizontal order of these lanes (cf. list pattern).

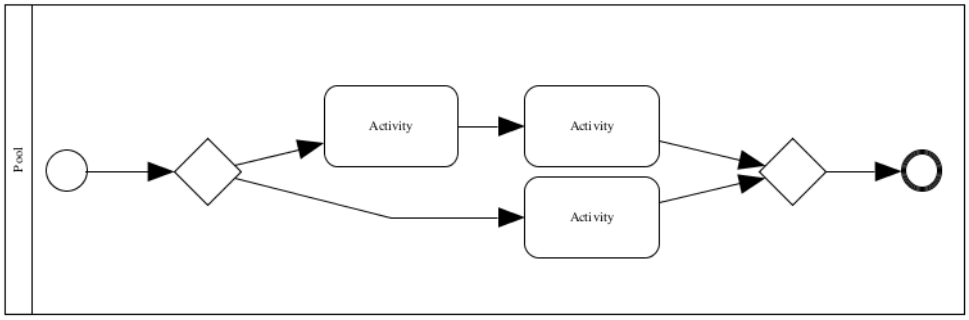


Figure 9.2: Business Process Models

Class Diagrams

For class diagrams (Figure 9.3), the students have implemented two edge routers, which may be combined. Edge routers are a somewhat different category of drawing algorithms as node positions are fixed. For class diagrams, the edge routers are applied to associations and generalizations.

- Layout behavior defined outside graph drawing algorithms: Nodes should not overlap (cf. node overlap removal pattern). Besides, attributes need to be aligned (cf. list pattern) and the nesting of packages and classes needs to be preserved (cf. rectangular containment pattern).

The first implemented edge router is an edge connector that makes sure that edges "follow" a component and start exactly at the contour of this component.

- Layout behavior defined via graph drawing algorithms: For class diagrams, edges need to follow classes, which are visualized as rectangles, or packages, whose shape is a bit more complex (cf. edge connector pattern).

The second implemented edge router is an edge positioner, whose purpose is to route edges, e.g. to introduce bend points. The algorithm especially avoids that edges cross nodes.

- Layout behavior defined via graph drawing algorithms: Again, the shape of nodes is important. For simplification, the bounding box of nodes is used as the basis of the computation. Other important requirements are that edge crossings are avoided and that two edges do not start or end at the same point.

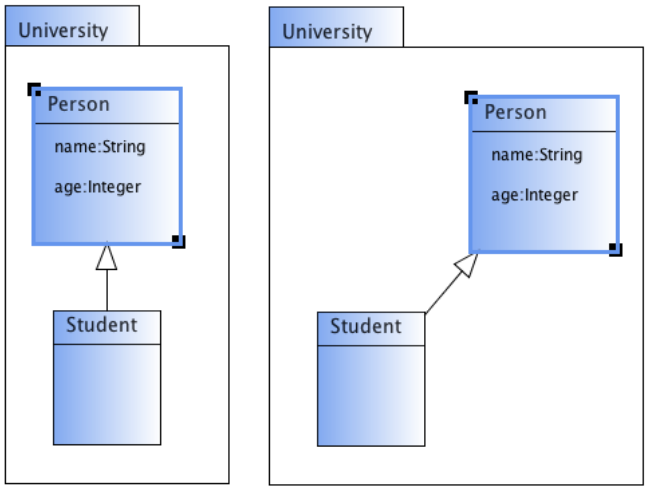


Figure 9.3: Class Diagrams: Before and After Moving Class Person

Figure 9.3 shows a sample user interaction: The initial diagram is shown on the left side. The user moves the class Person to the right. After movement, the layout engine enlarges the package, moves the attributes, and updates the generalization. The resulting diagram is shown on the right side.

9.1.3 Discussion

With the help of this user study, it was demonstrated that different editors require similar layout behavior. Evidence was given that the combination of graph drawing algorithms and other layout algorithms makes sense.
As expected, different groups of students defined similar layout patterns. For instance, some sort of edge connector pattern was defined for all three editors. In addition, some sort of list pattern was also defined for all three editors. Furthermore, some sort of rectangular containment pattern was defined for the business process model editor as well as the class diagram editor. It can be concluded that it makes sense to reuse and combine layout patterns in different layout engines.
It was further examined that some layout behavior was built into the graph drawing algorithms themselves, while other layout behavior was defined outside of the graph drawing algorithms. Most students defined the preservation of the size of nodes, the containment of nodes and the order of nodes (lists) outside the graph drawing algorithms. This functionality could be easily defined via constraint-based algorithms or rule-based algorithms. Hence, the combination of graph drawing algorithms and other layout algorithms sug-

gests itself.
One finding was a bit surprising: The students rather preferred to be in control of the layout engine instead of achieving a "perfect" layout. Without being asked, they put quite an effort on adapting the layout engine such as allowing influencing the layout at runtime. For instance, several options, such as the distance between nodes were provided for most of the graph drawing algorithms. In addition, the algorithms were adjusted to allow for options such as the preservation of the horizontal and (or) vertical ordering of nodes. Most groups also modified the framework in a sense that layout algorithms were no longer called automatically after user changes. Instead, the algorithm is explicitly called by clicking a button. Here, a tendency was apparent: Graph drawing algorithms that perform major structural changes, such as the layered layout or the circular layout, are explicitly triggered by the user. Graph drawing algorithms that only perform minor structural changes, such as the edge connector or the edge positioner, are called automatically.

9.2 Performance Evaluation

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. Performance tests showed that the only expensive part is the algorithm that controls the pattern combination. In some scenarios, the use of backtracking can lead to an explosion of cases resulting in a bad performance. The number of cases increases if many predicates involve the same variables. User studies showed that this is usually not the case in "real world scenarios". Instead, a result is found without doing backtracking at all in most cases.
In this section, a performance evaluation is given. The test environment is described in Section 9.2.1 and an introductory example is given in Section 9.2.2. The general performance test architecture is described in Section 9.2.3. In Section 9.2.4, a description of a performance test is provided for each of the layout patterns that were presented in Chapter 8. In Section 9.2.5, two performance experiments are outlined that give some insight into the performance of ad-hoc layout.

9.2.1 Test Environment

Performance was measured on a machine with the following technical details:

- Processor: 3.4 GHz Intel Core i7
- Memory: 8 GB 1333 MHz DDR3
- Operating System: Mac OS X Version 10.7.3
- Java: JDK 1.6.0_29

A special test environment was created that enables the testing of layout patterns. Each test is structured as follows:

- A diagram is created.
- One or more pattern instances are created for this diagram.
- User changes are simulated.
- The layout algorithm is executed.
- The time needed for layout computation is recorded.
- The diagram before and after layout adjustment is visualized.

9.2.2 Introductory Example

Figure 9.4(a) shows a diagram which consists of n nodes.

Pattern Instances

All n nodes are aligned vertically at the left side. In the following, the performance of two variants is examined:

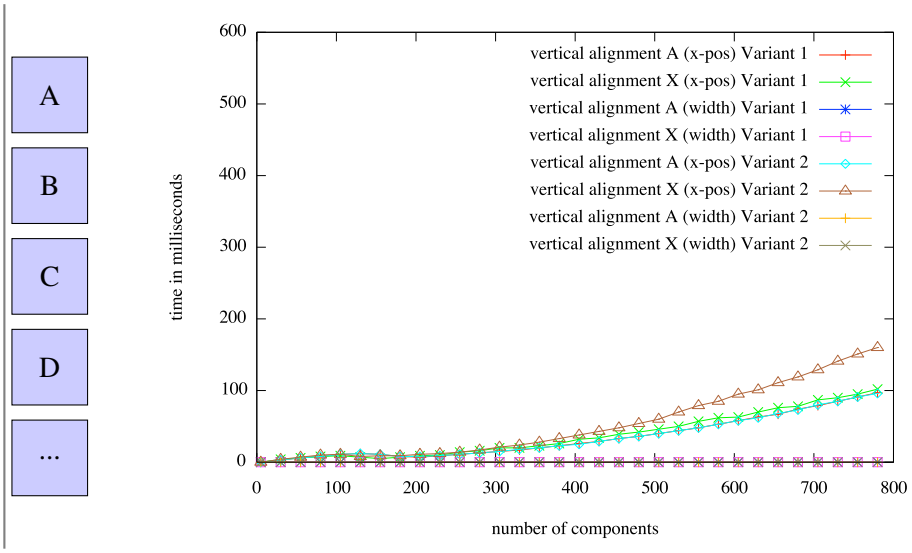
- Variant 1: Instances are created for the following sets of components: $\{A, B\}$, $\{B, C\}$, $\{C, D\}$, etc.
- Variant 2: Instances are created for the following sets of components: $\{A, B\}$, $\{A, C\}$, $\{A, D\}$, etc.

The editor developer who defines the layout pattern decides on how pattern instances are created. In an editor, usually only one variant is available. As will be seen in the following, such design decisions influence the overall performance.

User Interaction

The user may interact with the diagram in different ways: He or she may either move an arbitrary component or resize it. As a consequence, all other components need to be updated accordingly. In the following, four variants are examined:

- Component A is moved to the right.
- The width of component A is changed.
- The $n/2$-th component (component X) is moved to the right.
- The width of the $n/2$-th component (component X) is changed.



(a) Example (b) Runtime

Figure 9.4: Performance of the Introductory Example

The chart visualized in Figure 9.4(b) shows the time it takes to update the diagram shown in Figure 9.4(a), after the user changes one component. As can be seen, in case of variant 1, updating 100 components takes about 0.01 seconds and updating 800 components still takes less than 0.2 seconds. In case of variant 2, updating components takes a bit more time.
The first variant of the vertical alignment pattern performs better than the second variant. The reason for that is the construction of the different pattern instances, and hence, the structure of the constraint network. In both

variants, the movement of component A performs better than the movement of component X. Once again, the reason for that is the construction of the pattern instances. Changing the width of component A or of component X does not result in any layout changes. Hence, layout computation takes more or less no time.

9.2.3 Performance Test Architecture

In the following, tests for several patterns are presented. For each of these tests, a "construction" of pattern instances is chosen that shows a good performance. Furthermore, a user interaction is chosen that causes the layout engine to modify a huge part of the diagram.

Pattern	# Comps	# Pattern Inst.	# Ch. Comps
Layered Layout	$2n-1$	1 (n atomic)	$2n-1$
Circular Layout	$2n-1$	1 (n atomic)	$2n-1$
Node Overlap Removal	n	1 (n atomic)	2
Equal Horizontal Distance	n	$n-2$	n
Quadratic Component	n	n	1
Minimal Size Component	n	n	1
Equal Width	n	$n-1$	n
Align in a Column (center)	n	$n-1$	n
Vertical Alignment (left)	n	$n-1$	n
List	$n+1$	n	$n+1$
Rectangular Containment	n	$n-1$	n
C: Inner Attachment	n	$n-1$	n
C: Outer Attachment	n	$n-1$	n
C: Abstraction	$2n+1$	$2n$	$2n+1$
C: Application	$2n+1$	$2n$	$2n+1$

Table 9.1: Test Design

Each test is designed in a way that after the chosen user interaction is performed, if possible, all predicates that are present in the diagram need to be checked, all predicates are violated, and for each predicate, an associated rule changes the diagram. In contrast to this worst-case scenario, in real-world scenarios only a few predicates are usually affected after a user changes the diagram.
The performance depends on the number of components, the number of pattern instances and the degree of connectivity of the constraint network.
The tests are designed in such a way that they are comparable. Therefore, in each test, approximately n (atomic) pattern instances are created that in-

volve approximately n components. A user change is performed that changes exactly one component and that requires updating as many other diagram components as possible. Table 9.1 lists the tests created, together with the number of components created, the number of pattern instances created and the number of components that are changed by the layout engine.

9.2.4 Performance Tests

In the following, tests for most of the layout patterns that were described in Chapter 3 are presented.

Layered Layout

The layered layout pattern is tested as shown in Figure 9.5. n nodes and $n-1$ edges are created. Node 0 is connected with all other nodes. 1 pattern instance (alternatively n atomic pattern instances) is created that enforces the whole graph being layouted via a layered layout algorithm. A pattern instance is created for the following set of components:

- $\{0, 1, 2, ...\}$

User interaction is simulated in a sense that node 0 is moved to the bottom. As a consequence, the layout algorithm moves all other nodes to the correct position and updates all edges accordingly.

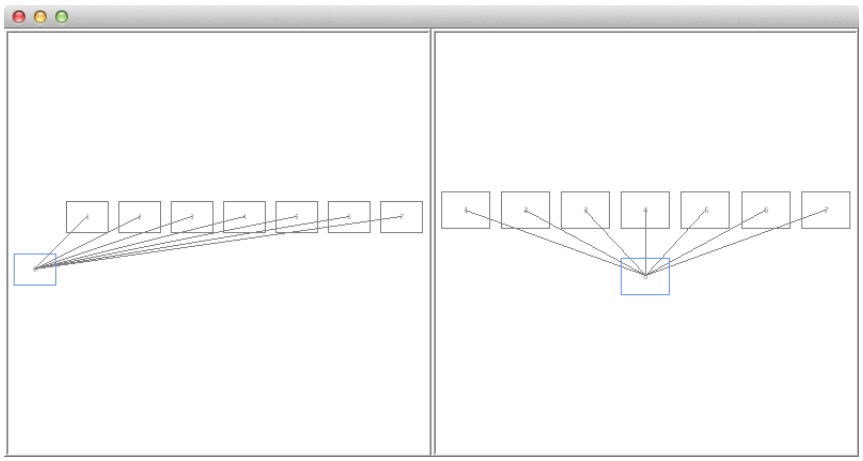


Figure 9.5: Layered Layout

Circular Layout

The circular layout pattern is tested as shown in Figure 9.6. n nodes and $n-1$ edges are created. Node 0 is connected with all other nodes. 1 pattern instance (alternatively n atomic pattern instances) is created that enforces the whole graph being layouted via a circular layout algorithm. A pattern instance is created for the following set of components:

- $\{0, 1, 2, ...\}$

User interaction is simulated in a sense that node 0 is moved to the left. The layout algorithm now updates the diagram by moving all other nodes to the correct position and updating all edges accordingly.

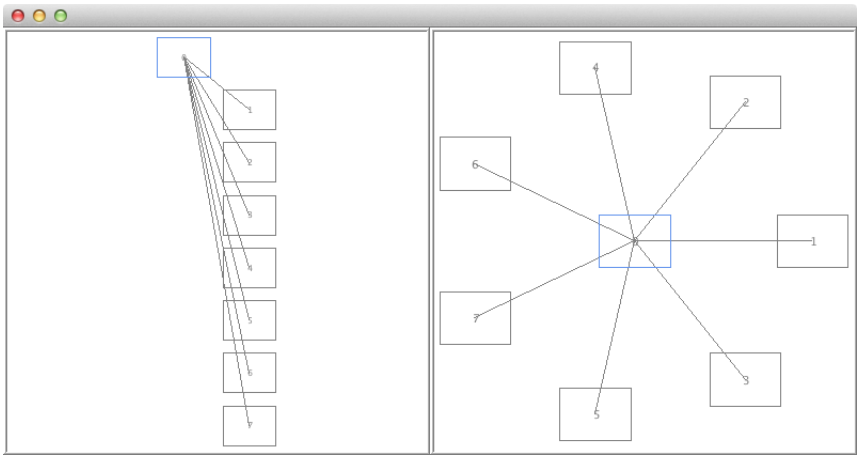


Figure 9.6: Circular Layout

Node Overlap Removal

The node overlap removal pattern is tested as shown in Figure 9.7. n nodes are created. 1 pattern instance (alternatively n atomic pattern instances) is created that ensures that the nodes do not overlap. A pattern instance is created for the following set of components:

- $\{0, 1, 2, ...\}$

User interaction is simulated in a sense that node 0 is moved to the right onto node 1. As a consequence, the layout algorithm moves some other nodes to ensure that nodes do not overlap. In the example shown in Figure 9.7, node 1 and node 2 are moved.

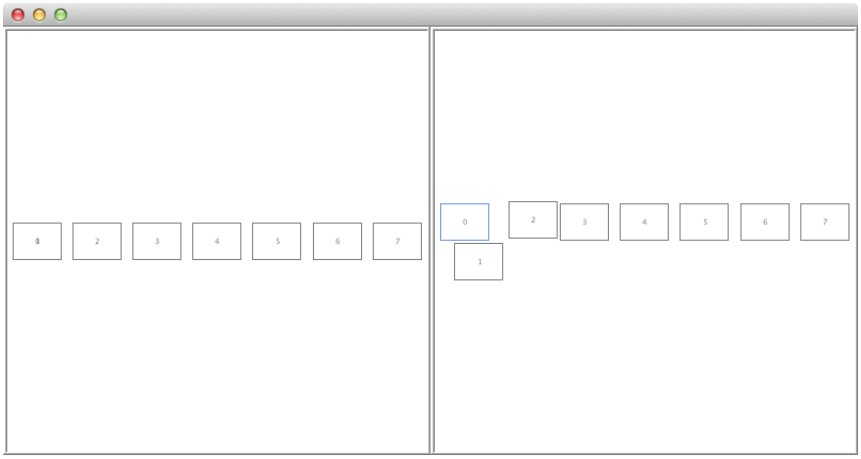


Figure 9.7: Node Overlap Removal

Performance of Graph Drawing Algorithms

The performance of patterns that encapsulate graph drawing algorithms is shown in Figure 9.8. The overall performance is better than the performance of constraint-based patterns, but worse than the performance of rule-based patterns. E.g. updating 100 components requires about 0.02 seconds.

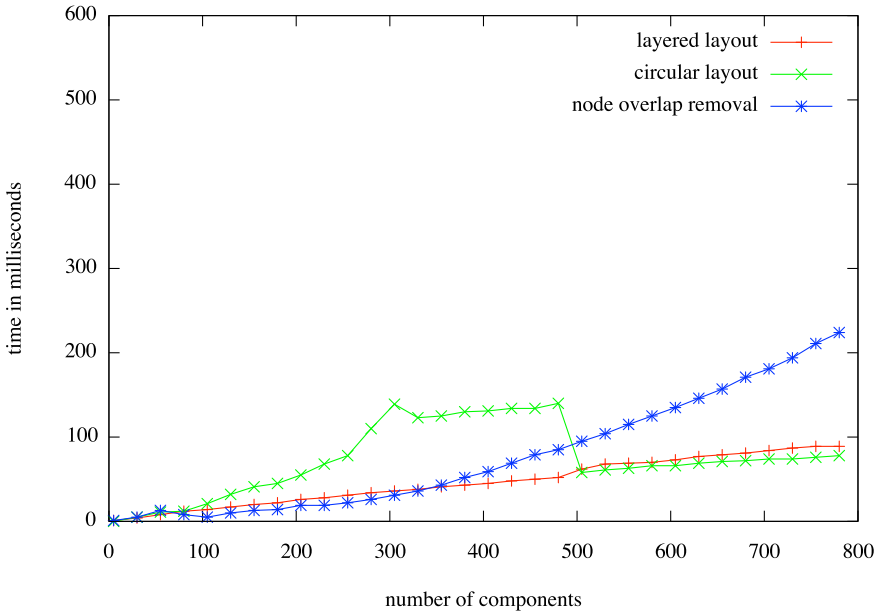


Figure 9.8: Performance of Graph Drawing Algorithms

The curve that corresponds to the circular containment pattern shows some irregularities. These irregularities have their origins in the third-party graph drawing algorithm that is encapsulated in the circular layout pattern.

Equal Horizontal Distance

The equal horizontal distance pattern is tested as shown in Figure 9.9. The diagram consists of n nodes. $n-2$ pattern instances are created that enforce these nodes having an equal distance. Pattern instances are created for the following sets of components:

- $\{0,1,2\}$, $\{1,2,3\}$, $\{2,3,4\}$ etc.

User interaction is simulated in a sense that node 0 is moved to the right. As a consequence, the layout algorithm moves all other nodes to an appropriate position.

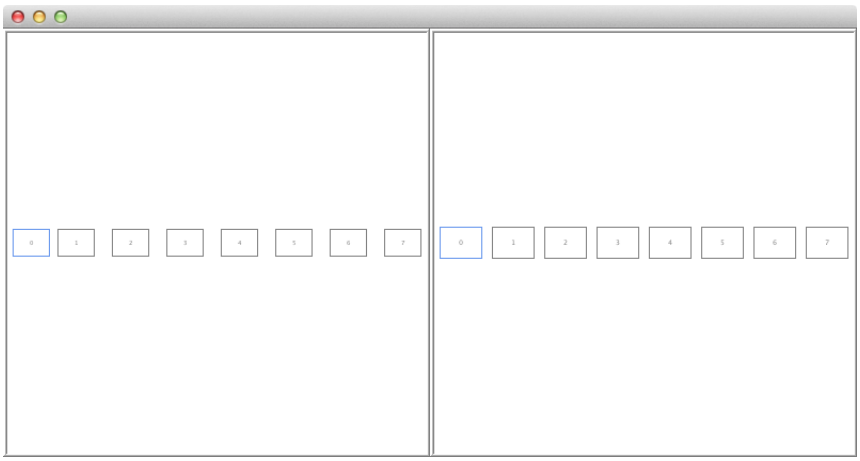


Figure 9.9: Equal Horizontal Distance

Performance of Constraint-based Algorithms

The performance of the pattern that encapsulates a constraint-based algorithm is shown in Figure 9.10. In the figure, performance is compared to the performance of a rule-based layout pattern that provides the (more or less) same functionality. As can be seen, the rule-based version performs much better than the constraint-based version. Nevertheless, the performance of the constraint-based version is still satisfactory: E.g. updating 100 components requires less than 0.1 seconds. (Note that the chart has another scale.)

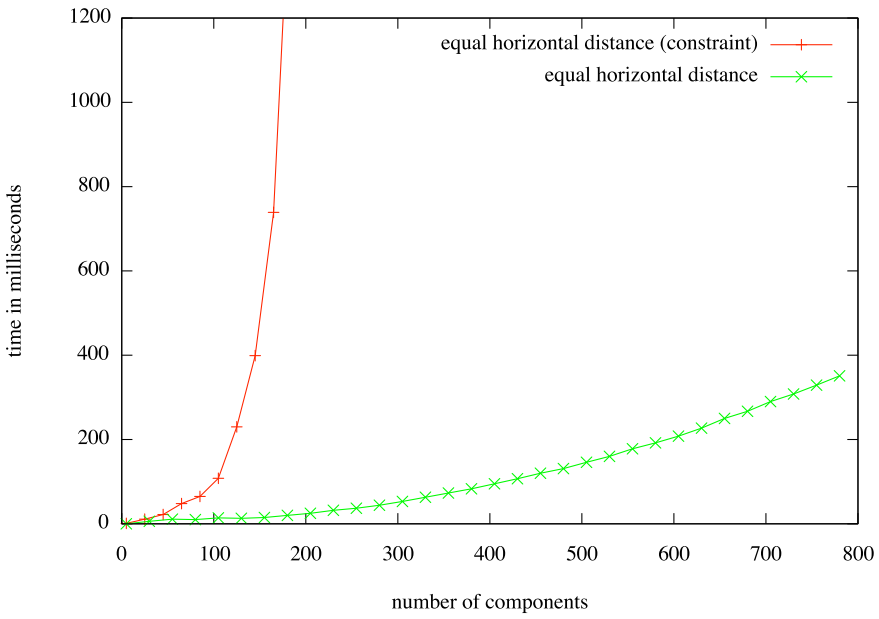


Figure 9.10: Performance of Constraint-based Algorithms

Constraint-based algorithms perform worst in our setting. Reasons for that are:

- Constraint solvers are general and powerful, and do not provide a problem-specific solution.
- The integration of the constraint solver in the framework is not perfect. The constraint solver receives an almost unordered set of many small csp-constraints as input. A reformulation and an ordering of these csp-constraints could increase the overall performance.
- After user changes, the CSP is not solved incrementally. Instead, the CSP is solved from scratch each time the user modifies the diagram.

Quadratic Component

The quadratic component pattern is tested as shown in Figure 9.11. n nodes are created. n pattern instances are created that enforce the n nodes being quadratic. Pattern instances are created for the following sets of components:

- $\{0\}$, $\{1\}$, $\{2\}$ etc.

User interaction is simulated in a sense that the height of node 1 is increased. The layout algorithm now updates the diagram by increasing the width of node 1 as well.

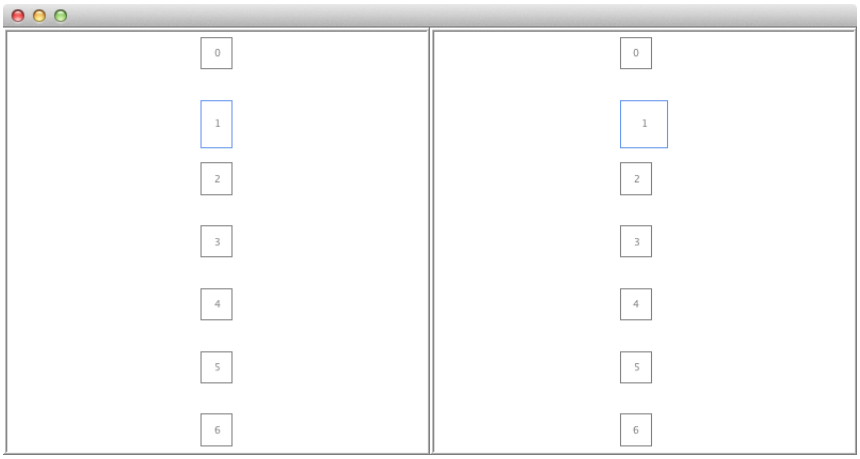


Figure 9.11: Quadratic Component

Minimal Size Component

The minimal size component pattern is tested as shown in Figure 9.12. n nodes are created. n pattern instances are created that enforce the n nodes having a minimal size. Pattern instances are created for the following sets of components:

- $\{0\}$, $\{1\}$, $\{2\}$ etc.

User interaction is simulated in a sense that the width of node 0 is decreased. This action is not allowed by the layout algorithm, and hence the algorithm undos the user change.

Equal Width

The equal width pattern is tested as shown in Figure 9.13. n nodes are created. $n - 1$ pattern instances are created that enforce all nodes having the same width. Pattern instances are created for the following sets of components:

- $\{0, 1\}$, $\{1, 2\}$, $\{2, 3\}$ etc.

User interaction is simulated in a sense that the width of node 0 is increased. The layout algorithm updates the diagram by increasing the width of all other nodes.

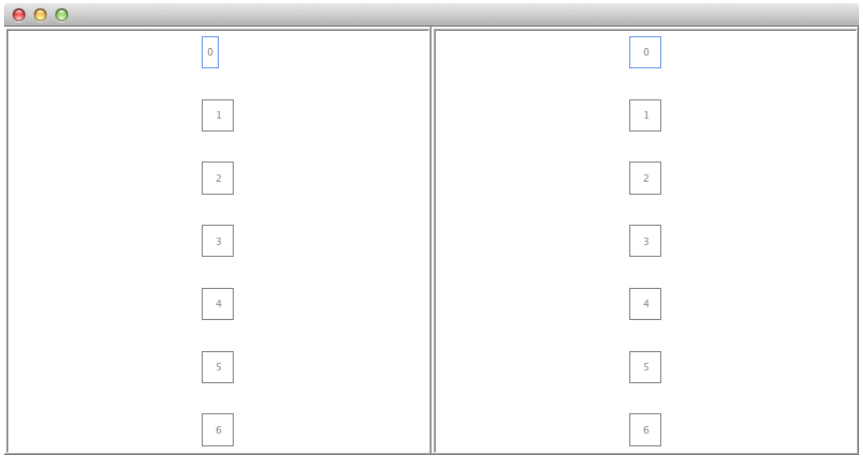


Figure 9.12: Minimal Size Component

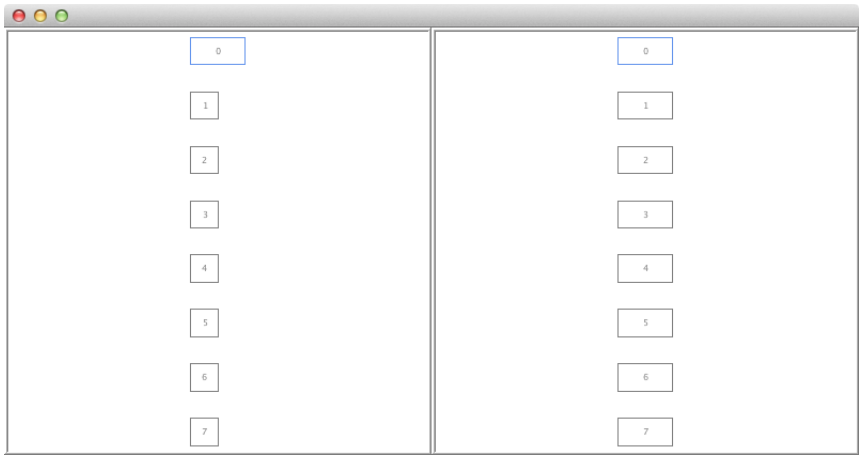


Figure 9.13: Equal Width

Align in a Column (Center)

The align in a column (center) pattern is tested as shown in Figure 9.14. n nodes are created. $n-1$ pattern instances are created that enforce the n nodes being aligned in a column at the center. Pattern instances are created for the following sets of components:

- $\{0,1\}$, $\{1,2\}$, $\{2,3\}$ etc.

User interaction is simulated in a sense that node 0 is moved to the left. The layout algorithm updates the diagram by moving all other components to the left.

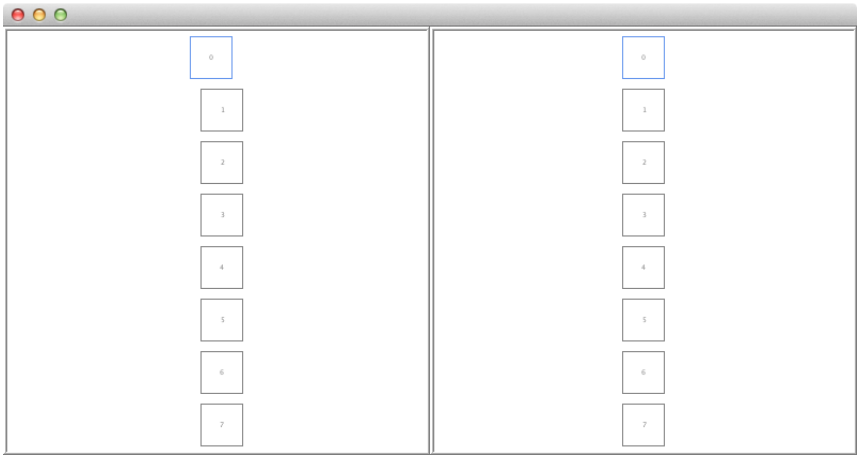


Figure 9.14: Align in a Column (Center)

Vertical Alignment (Left)

The vertical alignment (left) pattern is tested as shown in Figure 9.15. n nodes are created. $n-1$ pattern instances are created that enforce the n nodes being vertically aligned at the left side. Pattern instances are created for the following sets of components:

- $\{0,1\}$, $\{1,2\}$, $\{2,3\}$ etc.

User interaction is simulated in a sense that node 0 is moved to the right. The layout algorithm updates the diagram by moving all other components to the right.

List

The list pattern is tested as shown in Figure 9.16. One container node and n contained nodes are created. n pattern instances are created that enforce the contained nodes being arranged as a list. Pattern instances are created for the following sets of components:

- $\{-,1,0\}$, $\{1,2,0\}$, $\{2,3,0\}$, $\{3,4,0\}$ etc.

User interaction is simulated in a sense that the container node is moved to the right. The layout algorithm now updates the diagram by moving all contained nodes to the right.

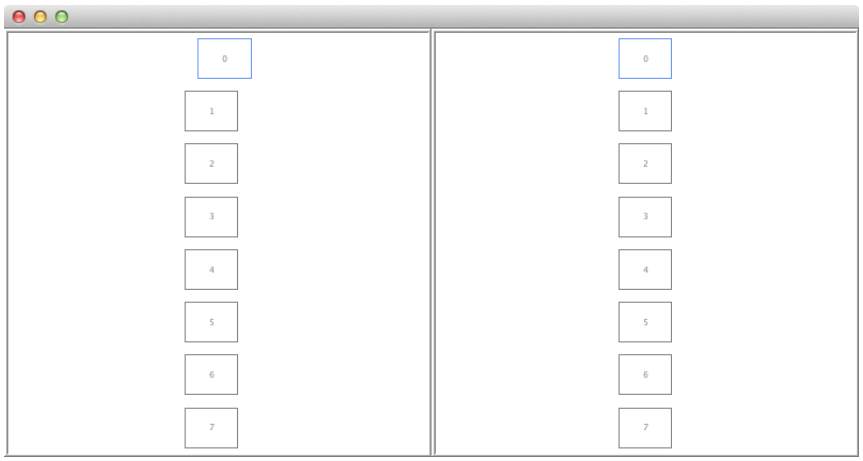


Figure 9.15: Vertical Alignment (Left)

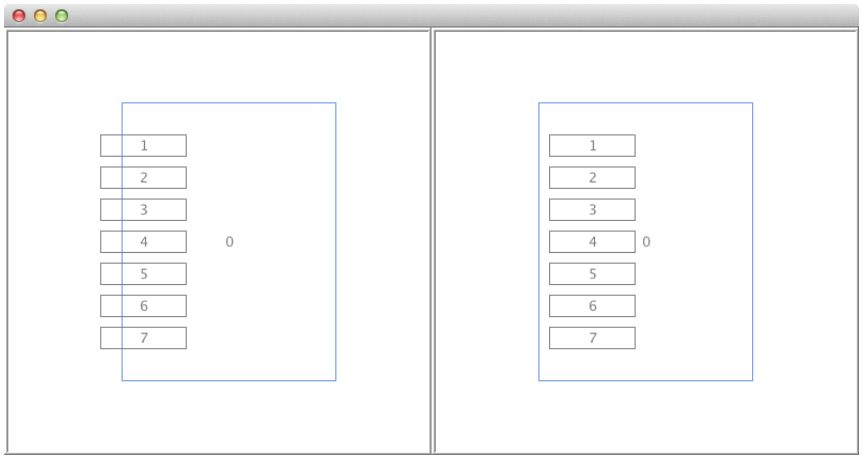


Figure 9.16: List

Rectangular Containment

The rectangular containment pattern is tested as shown in Figure 9.17. n rectangles are created. $n-1$ pattern instances are created that enforce all rectangles being correctly nested. Pattern instances are created for the following sets of components:

- $\{0,1\}$, $\{1,2\}$, $\{2,3\}$ etc.

User interaction is simulated in a sense that rectangle 0 is moved to the bottom. The algorithm now updates the diagram by increasing the height of all other rectangles.

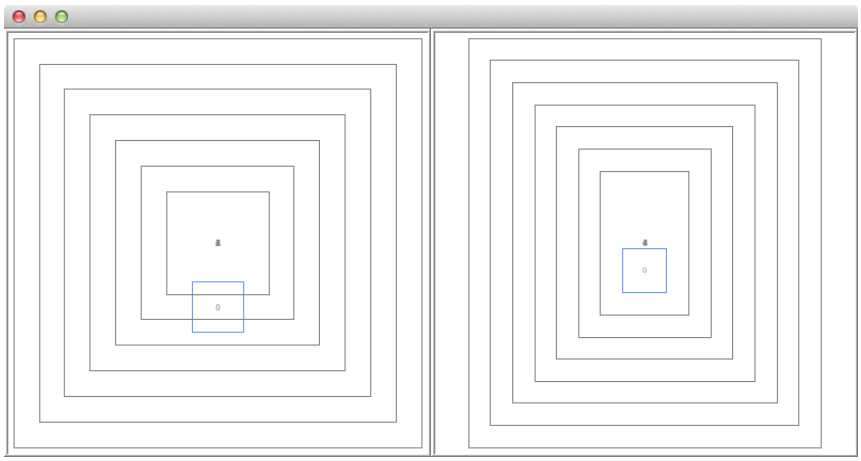


Figure 9.17: Rectangular Containment

Performance of Rule-based Algorithms

The performance of patterns that encapsulate rule-based algorithms is shown in Figure 9.18, and is quite convincing. E.g. updating 100 components requires less than 0.02 seconds.

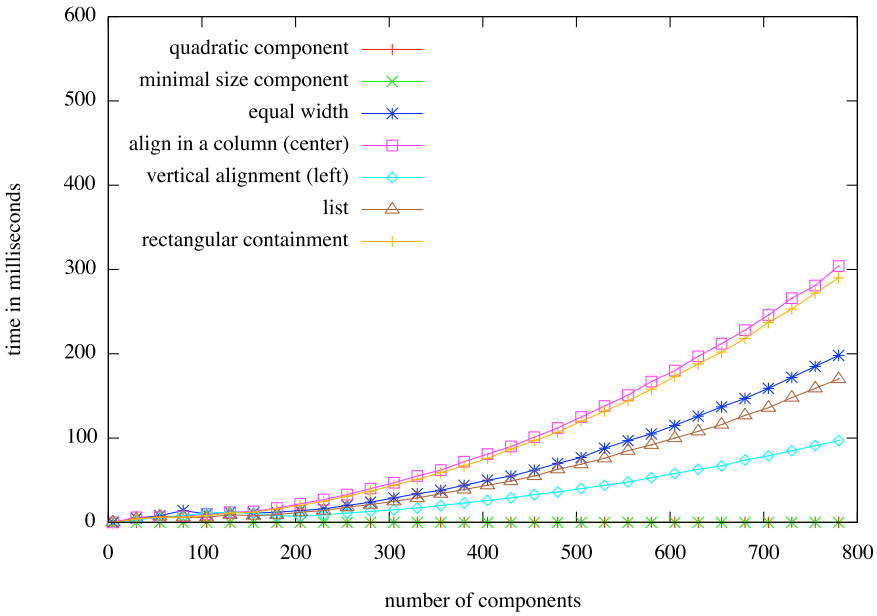


Figure 9.18: Performance of Rule-based Algorithms

Circular Containment

The test of the circular containment pattern consists of four parts: In the first two tests, the inner attachment as well as the outer attachment are tested individually. In the other two tests, the nesting of abstractions and the nesting of applications are tested individually as well.

Inner Attachment The first part is tested as shown in Figure 9.19. n circles are created. $n-1$ pattern instances are created that enforce the correct inner attachment of the circles. Pattern instances are created for the following sets of components:

- $\{0,1\}$, $\{1,2\}$, $\{2,3\}$ etc.

User interaction is simulated in a sense that circle 0 is moved to the right. The algorithm now updates the diagram by moving all other components to the right.

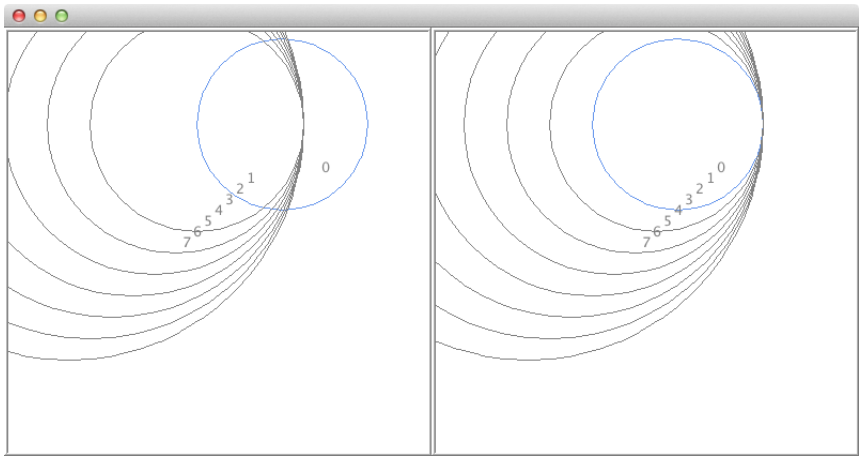


Figure 9.19: Circular Containment: Inner Attachment

Outer Attachment The second part is tested as shown in Figure 9.20. n circles are created. Each circle has one circle at the outer border of the circle. $n-1$ pattern instances are created that enforce the correct outer attachment of the circles. Pattern instances are created for the following sets of components:

- $\{0,1\}$, $\{1,2\}$, $\{2,3\}$ etc.

User interaction is simulated in a sense that circle 0 is moved to the right. The algorithm updates the diagram by moving all other circles to the right.

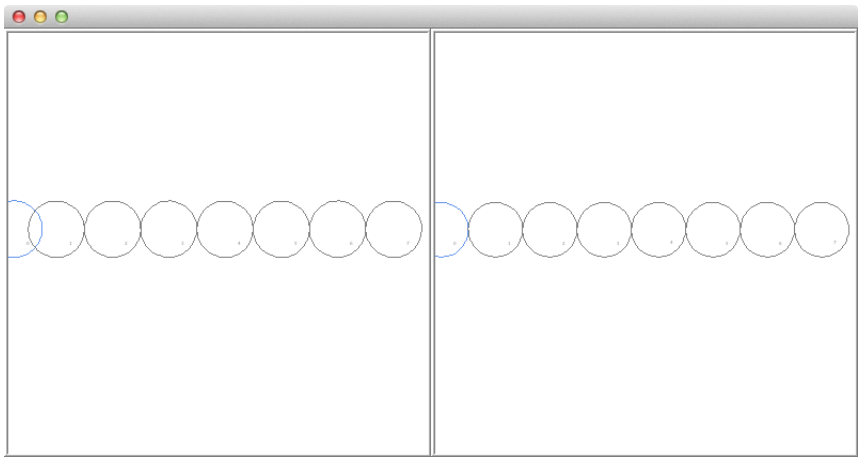


Figure 9.20: Circular Containment: Outer Attachment

Abstraction The third part is tested as shown in Figure 9.21. $2n+1$ circles are created. The structure is as follows: Each circle contains two circles: One at the inner border of the circle, and one inside the circle. About $2n$ pattern instances are created that enforce the correct nesting of the circles. Pattern instances are created for the following sets of components:

- Inner Attachment: $\{0,1\}$, $\{2,3\}$, $\{4,5\}$ etc.
- Circular Containment: $\{0,1,2\}$, $\{2,3,4\}$, $\{4,5,6\}$ etc.

User interaction is simulated in a sense that circle $2n$ (14 in the example) is moved to the left. The algorithm now updates the diagram by moving all other circles to the left.

Application The fourth part is tested as shown in Figure 9.22. $2n+1$ circles are created. The structure is as follows: Each circle contains two circles: One inside the circle, and one at the outer border of this circle. About $2n$ pattern instances are created that enforce the correct nesting of the circles. Pattern instances are created for the following sets of components:

- Outer Attachment: $\{1,2\}$, $\{3,4\}$, $\{5,6\}$ etc.
- Circular Containment: $\{0,1,2\}$, $\{2,3,4\}$, $\{4,5,6\}$ etc.

User interaction is simulated in a sense that circle $2n$ (14 in the example) is moved to the left. The algorithm now updates the diagram by moving all other circles to the left.

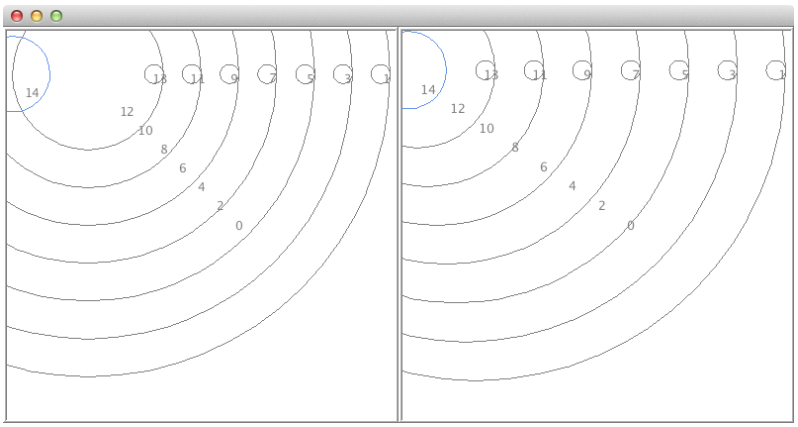


Figure 9.21: Circular Containment: Abstraction

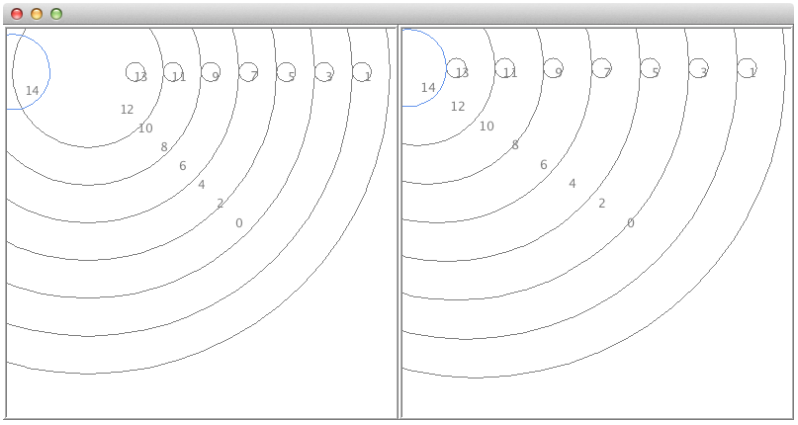


Figure 9.22: Circular Containment: Application

Performance of Circular Containment Pattern for VEX Editor

The performance of the circular containment pattern is shown in Figure 9.23. As can be seen, the performance of the inner attachment and the outer attachment is better than the performance of the abstraction and of the application. This is because the abstraction part contains the inner attachment part and the application part contains the outer attachment part.

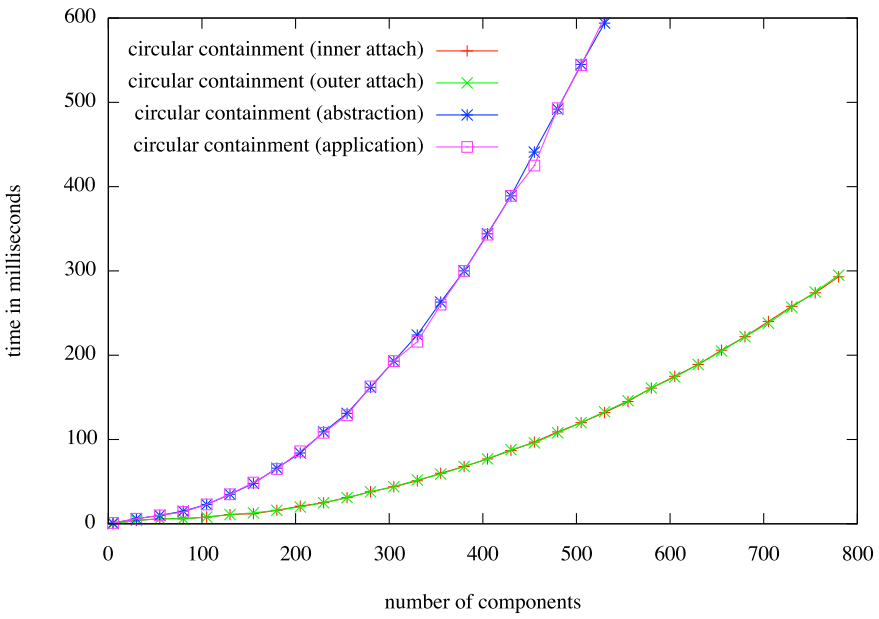


Figure 9.23: Performance of Circular Containment Pattern

Vertical Alignment (Variant 1 and Variant 2)

In Chapter 5, two variants of the vertical alignment pattern were described. In the first variant, the Elems PMM contains the attributes x, y, w, and h, which are mapped to the components' attributes x, y, w and h. As these attributes are equal, no mapping p-constraints are required. In the second variant, the Elems PMM contains the attributes t, b, l, and r, which are mapped to the components' attributes x, y, w and h. In this variant, several mapping p-constraints are required. As a consequence, the constraint network that corresponds to the second variant is more complex than the constraint network that corresponds to the first variant.
The design of the test for the second variant is equal to the design of the test for the first variant that was already described.

Performance of Patterns with Mapping Constraints

The performance of the first and the second version of the vertical alignment pattern are shown in Figure 9.24. As can be seen in the figure, the performance of the second variant is worse than the performance of the first variant. This is because there is an increased number of predicates that need to be checked, and an increased number of attributes that need to be updated.

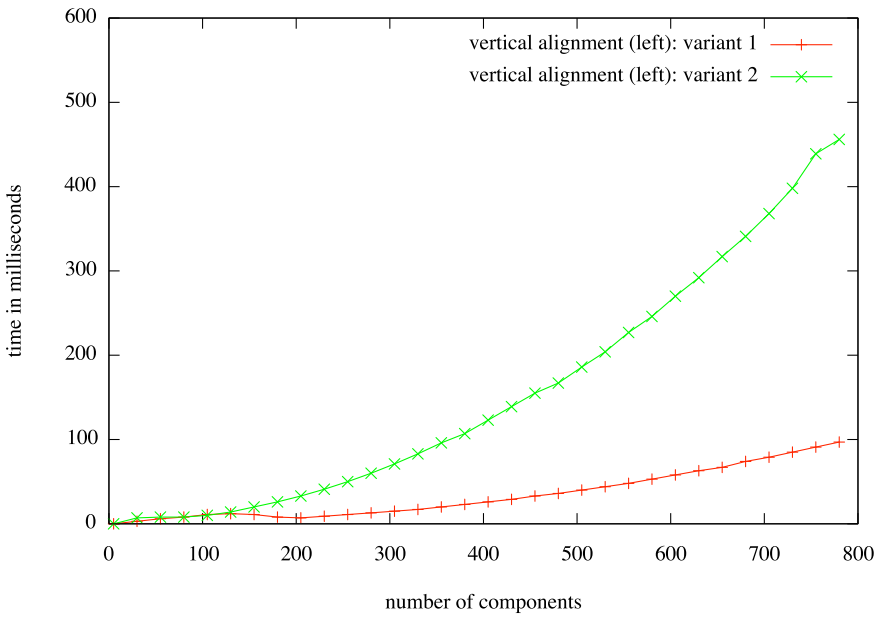


Figure 9.24: Performance of Patterns with Mapping Constraints

9.2.5 Performance Experiments

In the following, two performance experiments are presented. These experiments were already outlined in [73]. They give some insight into the performance of global ad-hoc layout, local ad-hoc layout and local ad-hoc layout with propagation.
In the two performance experiments, ad-hoc layout is computed for the vertical alignment (left) pattern only. Furthermore, no pattern instances are present in the diagram before ad-hoc layout is computed.

Ad-hoc Layout: Experiment 1

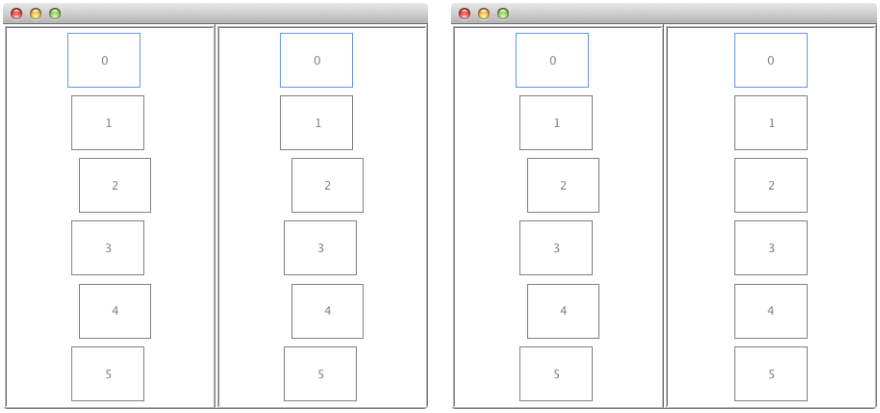
The first performance experiment starts with a diagram that consists of n components that are arranged almost vertically. The user moves the topmost component to the left. As a consequence, the GAL as well as the LAL/P align all components vertically, whereas the LAL aligns the moved component and its lower neighbor only.
Figure 9.25 shows an example: The diagram consists of $n = 6$ components that are arranged almost vertically. The left figure in Figure 9.25(a) and the left figure in Figure 9.25(b) show the diagram after the user has moved component 0. The right figure in Figure 9.25(a) shows the diagram after the LAL was computed and the diagram was updated: The two topmost

components are aligned vertically. The right figure in Figure 9.25(b) shows the diagram after the GAL or the LAL/P was computed and the diagram was updated: All components are aligned vertically. In case of the LAL, pattern instances were created for the following sets of components for the example:

- $\{0, 1\}$

In case of the GAL and the LAL/P, pattern instances were created for the following sets of components for the example:

- $\{0, 1\}$, $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{4, 5\}$



(a) Diagram - LAL (b) Diagram - GAL & LAL/P

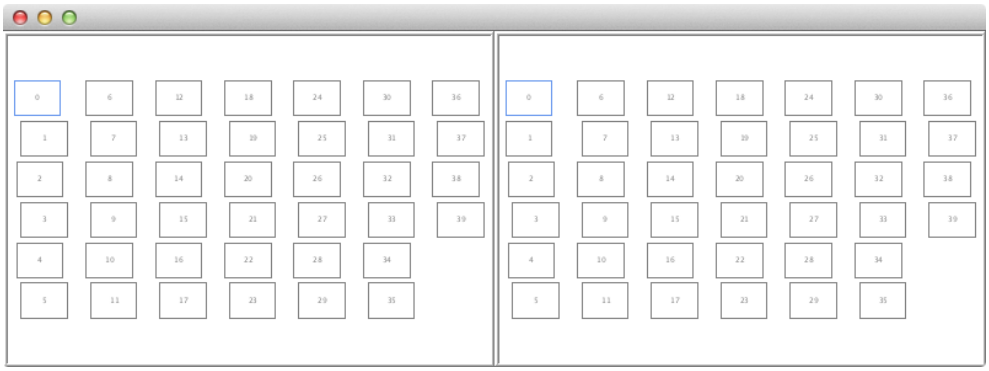
Figure 9.25: Automatic Ad-hoc Layout (Test 1)

Figure 9.27 shows the time in milliseconds it takes to compute the GAL (red), the LAL (green) and the LAL/P (blue) for n components. As can be seen, the computation of the LAL is very fast. The computation of the GAL and the LAL/P is more time-consuming. For instance, the computation of the LAL for $n = 40$ components takes about 1 millisecond, the computation of the GAL takes about 30 milliseconds, and the computation of the LAL/P takes about 90 milliseconds.

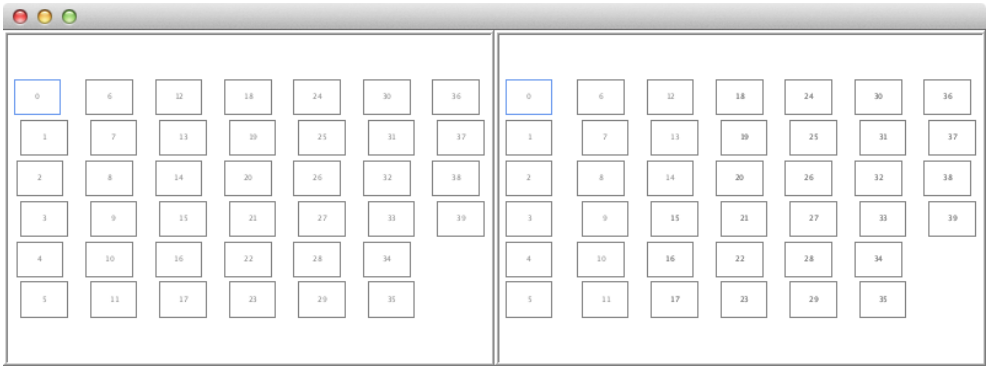
Ad-hoc Layout: Experiment 2

The second performance experiment starts with a diagram that consists of n components that are arranged as a matrix. The matrix is almost quadratic, e.g. a diagram with 100 components has 10 rows and 10 columns. The user

moves the component in the top-left corner to the left. As a consequence, the GAL as well as the LAL/P align the leftmost column vertically, whereas the LAL aligns the moved component and its lower neighbor only.



(a) Diagram - LAL



(b) Diagram - GAL & LAL/P

Figure 9.26: Automatic Ad-hoc Layout (Test 2)

Figure 9.26 shows an example: The diagram consists of $n = 40$ components, which are arranged as a matrix. The left figure in Figure 9.26(a) and the left figure in Figure 9.26(b) show the diagram after the user has moved component 0. The right figure in Figure 9.26(a) shows the diagram after the LAL was computed: The components 0 and 1 are aligned vertically. The other components have not been modified. The right figure in Figure 9.26(b) shows the diagram after the GAL or the LAL/P was computed: The components of the leftmost column are aligned vertically. The other components have not been modified. In case of the LAL, pattern instances were created for the following sets of components for the example:

- $\{0, 1\}$

In case of the GAL and the LAL/P, pattern instances were created for the following sets of components for the example:

- $\{0, 1\}$, $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{4, 5\}$

Figure 9.28 shows the time in milliseconds it takes to compute the GAL (red), the LAL (green) and the LAL/P (blue) for n components. As can be seen, the computation of the LAL is very fast. The computation of the LAL/P takes a bit more time, but is still acceptable for the use in an interactive environment. In contrast, the GAL is rather time-consuming. For instance, the computation of the LAL for the example shown in Figure 9.26 ($n = 40$ components) takes about 1 millisecond, the computation of the LAL/P takes about 5 milliseconds, and the computation of the GAL takes about 30 milliseconds.

Discussion

The most striking difference between the results of the two performance experiments is that GAL performs better than LAL/P in the first experiment, while it is slower in the second experiment. GAL performs better than LAL/P in the first experiment, because all components are involved in layout computation in both cases, and GAL does not need to take the neighborhood-relation of components into account. In the second performance test, LAL/P performs better than GAL, because only a small subset of components are involved in layout computation.
As experiments showed, usually only a small subset of components is involved in layout computation in practical scenarios. Therefore, the first experiment can be considered as worst-case scenario, while the second one as average-case scenario. Our summarization is that GAL rapidly results in performance issues, whereas LAL as well as LAL/P turned out to be practically usable. This is due to the fact that layout improvements are only computed locally, and hence performance does not (directly) depend on the size of the diagram, but rather depends on the layout behavior defined for it.

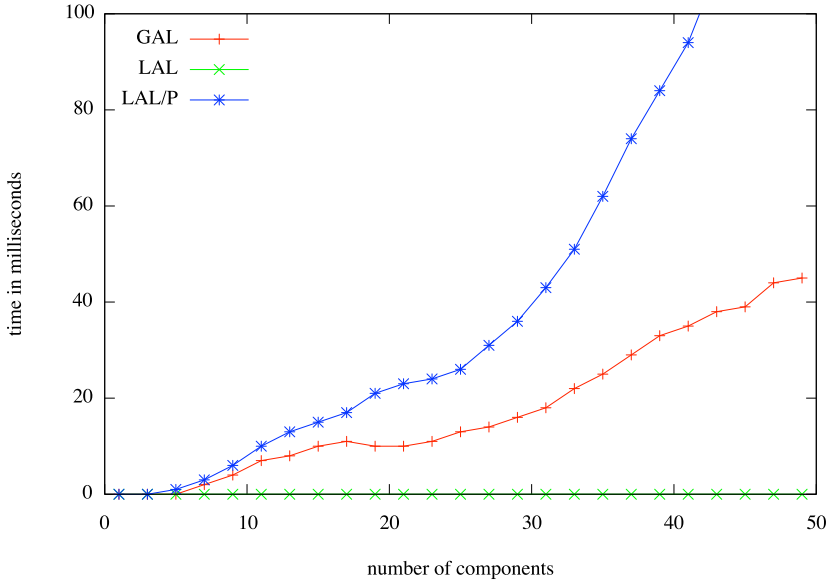


Figure 9.27: Automatic Ad-hoc Layout (Test 1): Performance

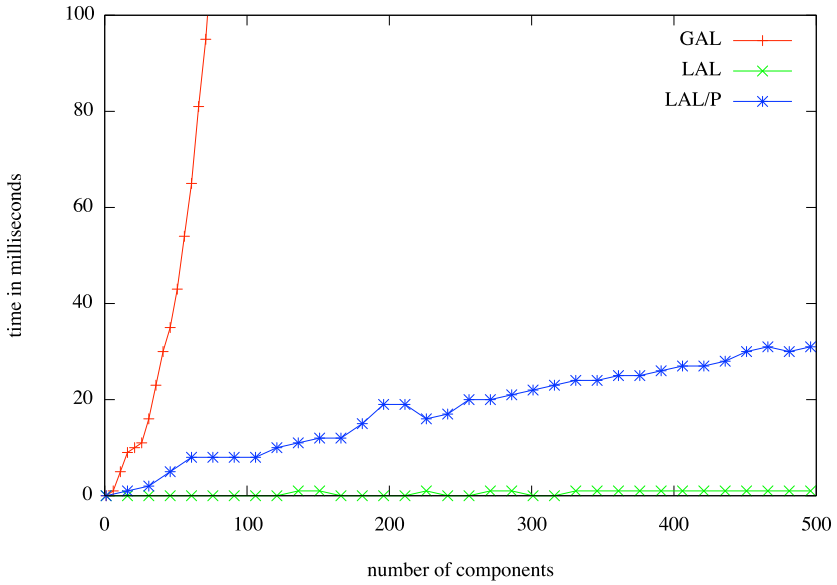


Figure 9.28: Automatic Ad-hoc Layout (Test 2): Performance

9.3 Summary

In this chapter, an evaluation of the approach in terms of usability and in terms of performance was given. A user study, which aims at identifying layout patterns that are commonly "needed" in diagram editors, was described in Section 9.1. In Section 9.2, a performance evaluation was presented.

9.3.1 Performance Evaluation

In the performance tests, each layout pattern was tested on its own. The combination of layout patterns was not tested via automatic tests, but via informal user experiments. These user experiments showed that the performance is convincing.
As mentioned in the beginning, the most expensive part of layout computation is the algorithm that controls the pattern combination. The performance of this algorithm usually depends on the number of components, the number of pattern instances and the degree of connectivity of the constraint network. In order to create some sort of worst-case scenario, the tests were designed in a sense that the number of pattern instances as well as the degree of connectivity of the constraint network was maximized. The tests showed that the performance in this scenario is still convincing.
As experiments showed, in contrast to this worst-case scenario, real-world scenarios usually only affect a few pattern instances after a user changes the diagram. Hence, performance is much better.
The performance tests mentioned in this chapter are based on a prototypical implementation of the concepts described in this thesis. Performance could be improved by enhancing the following:

- Currently, layout information is computed from scratch each time it is needed.
  - The reuse of already computed "layout information" would improve performance.
  - The computation of the "layout information" in advance would also improve performance.
- At the moment, the layout is continuously updated during movement. This means that the layout is more or less updated "for every pixel", or more precisely, after each mouse event. Updating the layout less often would be a good alternative.

Chapter 10

Conclusions

In this chapter, the thesis is concluded. A summary is given in Section 10.1. In Section 10.2, some related areas are mentioned, which had a point of contact with this thesis. Some future directions are discussed in Section 10.3.

10.1 Summary

In this thesis, a general approach for the definition of layout behavior for diagram editors was described. In Chapter 2, related work was discussed. Chapter 3 introduced four visual languages, namely the running examples used in this thesis. In addition, several layout patterns were described that are used in editors for these visual languages. The concept of layout patterns was described in Chapter 4. In Chapter 5, details were given about the control algorithm whose purpose it is to combine different layout patterns. In Chapter 6, the user-controlled instantiation of layout patterns and some special features that are useful in the context of user-controlled instantiation of layout patterns were described. In Chapter 7, the concept of layout suggestions and of ad-hoc layout were introduced. In Chapter 8, several layout patterns were described in detail. In Chapter 9, an evaluation of the approach in terms of usability and in terms of performance was given. Finally, in Appendix A, some specification and implementation details are provided. In the approach, layout functionality is defined via layout patterns that encapsulate certain layout behavior and thus make reuse easier. Instantiation of layout patterns may either be performed automatically or user-controlled. Layout behavior can be defined (amongst others) via one of the following approaches: Graph drawing algorithms, constraint-based algorithms or rule-based algorithms. Different layout patterns may be combined by the help of the control algorithm. The pattern-based layout approach enables two

new features, namely layout suggestions and automatic ad-hoc layout. The usability evaluation and the performance evaluation show that the approach fulfills the requirements of an editor user and that it is practically usable.
All in all, with the help of the layout approach presented in this thesis, a layout engine for a new diagram editor can be created quite fast, as already defined layout behavior can be reused. Hence, the approach is especially well suited for DiaMeta, an environment that aims at creating prototypes of diagram editors as fast as possible.
Furthermore, with the help of this approach, it is possible to define a great variety of functionalities that support the user in an inherently dynamic environment.

10.2 Application Areas

The layout approach presented in this thesis can also be used in other areas. Some application areas that were touched during the development of the layout approach are described in the following.

10.2.1 Layout Specification via Graph Transformations

In this thesis, three different types of layout patterns were described, namely layout patterns that encapsulate graph drawing algorithms, constraint-based algorithms and rule-based algorithms.
In [65], a fourth category was sketched: It is possible to define a layout algorithm via a graph (or model) transformation. Algorithms that consist of multiple phases are well suited to be defined via graph transformations. One example is the Sugiyama algorithm [89, 35], which creates a layered layout. The algorithm has several phases: (1) cycle removal, (2) layering, (3) node ordering, and (4) coordinate assignment. These phases can easily be defined via graph transformations, as was shown in [65].

10.2.2 Diagram Import

In DiaMeta, the editor user usually creates a diagram. Internally, a meta-model instance is automatically created that corresponds to this diagram. The creation of this meta-model instance is straightforward. In contrast, the automatic creation of a diagram from a meta-model instance is more complex. In [88], it is described how the automatic creation of a diagram can be performed in DiaGen. In [3], it is outlined how triple graph grammars [99] can be used to enable this functionality in DiaMeta. TGGs are well

suited, because they allow for a bidirectional transformation between the graph model and the instance graph (see Appendix A). One crucial step during diagram import is the computation of an initial layout. The approach presented in this thesis can potentially be useful in this context.

10.2.3 Different Views of a Diagram

In [52, 51], DiaMeta was extended in a sense that it now provides different views of the same diagram. If the user modifies the diagram in one of the views, the other views need to be updated. This means that also the layout needs to be updated. In this context, the approach described in this thesis might be helpful.

Fish-eye viewing and semantic zooming [55] are related topics here. Their realization also requires a powerful layout approach.

10.2.4 Layout for Three-dimensional Diagrams

In [113], a three-dimensional variant of DiaMeta editors was presented. In [112], GEF3D, a three-dimensional variant of GEF was described. These three-dimensional diagram editors could be combined with the layout approach presented in this thesis. The combination should be straightforward.

10.2.5 Animated Visual Languages

Animated visual languages describe diagrams that change over time. One example are petri nets. A petri net mainly consists of places, transitions and arcs. It can be executed, which means that enabled transitions are fired nondeterministically one after another. Strobl *et al.* describe in [104, 105] a framework for generating editors for animated visual languages. The framework is an extension of the DiaMeta framework. Ermel *et al.* also describe a framework for the creation of editors for animated visual languages [33, 7]. The layout approach presented in this thesis might be helpful when the diagram needs to be updated after one execution step.

Games are also some sort of animated visual languages. DiaMeta can be used to define simple state-based games, such as Ludo [58, 66, 94]. It can even be used to prototype more complex games [74]. Once again, the layout approach could be utilized in these "editors".

10.2.6 Sketched Diagrams

The most natural way of drawing a diagram is via sketching. E.g. in [50], a general approach for the recognition of diagrams is described. In [22], another approach is described, which, for instance, enables sketching and recognition of Euler diagrams. In [91, 22], Plimmer *et al.* describe how a sketched diagram can be updated automatically after layout modification.
In [18, 19, 17], Brieler *et al.* describe an extended version of the DiaGen framework, which enables the generation of visual language editors that allow for sketching a diagram with a stylus. With the help of the approach presented by Plimmer *et al.*, the sketched diagram could be beautified by using the layout approach presented in this thesis. As the extended version is based on DiaGen, not on DiaMeta, the layout approach cannot be used straightforwardly.

10.2.7 Diagram Editors for Multi-Touch Screens

In [44], a graph editor is described that runs on an iPod, iPhone or iPad. Such an editor also requires a layout engine. The layout approach presented in this thesis could also be used here.
These kinds of input devices provide somewhat different interaction mechanisms. Amongst others, one or more fingers can be used as input. For such input devices, the layout approach could be extended in a sense that it also supports such interaction mechanisms. E.g. shaking the device could result in updating the layout of the whole diagram.
In [41, 42], Frisch *et al.* describe some layout behavior that is specifically tailored to multi-touch screens. The layout behavior described in this paper is hand-coded. Such layout behavior could also be defined via the layout approach described in this thesis.

10.3 Future Directions

In this section, first of all, some usability improvements concerning the specification of layout patterns and the definition of a layout engine for a certain visual language editor are discussed. Thereafter, the creation of an initial layout with the help of the approach presented in this thesis is sketched.

10.3.1 Specification of the Layout Engine

The specification of the layout engine for a certain visual language editor consists of two parts:

- The specification of layout patterns.
- The integration of layout patterns into an editor.
  - The choice of layout patterns and the definition of correspondences between LM and PMs.
  - The extension of the GUI of the visual language editor.

Implementation of Layout Patterns

First, layout patterns need to be specified. At the moment, language-dependent layout patterns as well as language-independent layout patterns are written by hand.
It turned out that the most natural procedure during the creation of a language-independent layout pattern is as follows:

1. Creation of a language-dependent version of the layout pattern.
2. Manual "transformation" of this language-dependent layout pattern into a language-independent one.

Instead of writing the language-dependent version of a layout pattern by hand, an abstract specification mechanism could be used. Here, some sort of (visual) DSL could be invented. Based on this specification, the corresponding Java code could be generated.
Furthermore, the transformation of a language-dependent layout pattern into a language-independent one could be performed (partially) automatic.
A tool for the specification of layout patterns could even be generated. In this context, also some sort of *visual* specification of layout patterns could be enabled. This idea as well as a prototypical implementation was outlined in [62].

Integration of Layout Patterns into an Editor

Besides the specification of layout patterns, they need to be integrated into an editor. First, layout patterns need to be chosen, which is performed manually at the moment. GUI support for this part would be straightforward. Second, correspondences between the LM and the PMs need to be defined. At the moment, these correspondences are also defined manually. Instead, a certain transformation language could be used. As EMF [102] is used for the specification of the LMM and the PMMs, a specification language that is based on EMF, e.g. based on Xtext [36], would be a good choice. Alternatively, even a visual model transformation language could be used. Last, the

integration into the framework needs to be performed. For that purpose, the DiaMeta framework as well as the GEF framework were extended in such a way that nothing else needs to be specified.

Discussion

All the improvements mentioned above involve the introduction of a new DSL and (or) GUI. As a consequence, the editor developer would need to get used to a new DSL and (or) GUI.
In our opinion, for many editor developers, a plain Java interface (as it is used now) is the better alternative. This has the benefit that the editor developer can use a language he or she is familiar with. He or she does not need to get used to a new DSL and (or) GUI. This has also the benefit that it is easier to react on modifications, as changes only imply the change of the Java interface, not the change of the DSL and (or) the GUI.

10.3.2 Initial Layout

The pattern-based layout approach presented in this thesis could be used for the creation of an initial layout. In order to create an initial layout, the following procedure is proposed:

- All attribute values are initialized.
- All pattern instances are created that are triggered by the automatically applied layout patterns.
- Zero or more additional layout pattern instances are created. Here, a pattern that encapsulates a graph drawing algorithm is a good choice.
- The initial layout is computed
- The additionally created pattern instances are removed again.
- The user refines the layout.

The input format for a graph could, for instance, be GraphML [48]:

```
<graphml>
  <graph id="g">
    <node id="A"/>
    <node id="B"/>
    <node id="C"/>
    <node id="D"/>
```

```
    <edge source="A" target="B"/>
    <edge source="B" target="C"/>
    <edge source="B" target="D"/>
  </graph>
</graphml>
```

Firstly, all attributes are initialized: Each node is located at the position $(0, 0)$, has the width 0 and the height 0. Each edge starts at the position $(0, 0)$ and ends at the position $(0, 0)$. For this diagram, the following pattern instances are automatically created: Instances of the minimal size pattern are created for the sets $\{A\}$, $\{B\}$, $\{C\}$ and $\{D\}$ of components. An instance of the edge connector pattern is created for the following set of components: $\{A, B, C, D, e1, e2, e3\}$. $e1$ is the edge between the components A and B, $e2$ is the edge between the components B and C, and $e3$ is the edge between the components B and D. In addition, the layered layout pattern is manually applied to all nodes and edges. As a consequence, the diagram is arranged as can be seen in Figure 10.1.
In this example, all attributes are initialized with the value 0. Alternatively, attributes could also be initialized with other values. This way, the "quality" of the initial layout could possibly be improved.

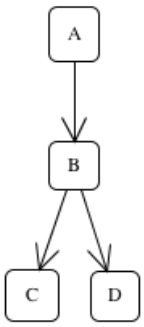


Figure 10.1: Initial Layout of the Diagram

Open Issues

In the example shown above, this procedure leads to an acceptable layout. But this may not be the case for other visual languages. In general, the following question needs to be answered: Which additional layout pattern instances should be created?

Besides, the layout algorithm is usually triggered by a user modification. But what is the "user modification" in this scenario? As a first solution, an arbitrary node could be chosen being the changed component.

Structured Editing

The mechanism for initial layout creation could be further used for structured editing. Here, a layout pattern could be applied to the newly created components and (maybe) to some components that are in the direct neighborhood of these components. At this point, this is only a rough idea, and has to be examined in more detail.

Content Assist

In [80, 81, 83, 79], Mazanek *et al.* present some new methods for syntax-based user assistance. These methods comprise the possibility to automatically extend incomplete diagrams and to generate correct diagrams of an arbitrary size. After modifying a diagram or after generating a diagram from scratch, an important issue is the layout of the newly created diagram components, as discussed in [82]. This problem is quite similar to structured editing, and the layout approach presented in this thesis could be helpful. These new methods for syntax-based user assistance were developed for DiaGen, and hence, are based on hypergraph grammars instead of meta-models. As a consequence, the layout approach described in this thesis cannot be used straightforwardly.

10.4 Summary

In this chapter, the thesis was concluded. A summary was given in Section 10.1. In Section 10.2, some related areas were mentioned, and in Section 10.3, some future directions were discussed.
The two future directions mentioned were chosen, because some ideas have already been developed. In general, future directions can be categorized as follows:

- Usability enhancement for the editor user and the editor developer.
- Extension of the layout approach.
- Utilization of the layout approach in other application domains.

Appendix A

Specification and Implementation

All concepts described in this thesis are implemented in Java, and are part of a layout framework. This framework enables the specification of layout patterns, comprises the control algorithm and provides a test environment. Furthermore, a library of layout patterns and one or more unit tests for each of these layout patterns are part of this framework.
In this chapter, some specification and implementation details are given. Details about the layout framework itself are given in Section A.1. In Section A.2, some details are given about the integration of a layout engine, which is defined by the help of this framework, into DiaMeta and into GEF editors.

A.1 Layout Framework

With the help of the framework, layout patterns, which were described in Chapter 8, may be specified. These patterns may be instantiated, and the instances may be combined via the control algorithm described in Chapter 5, which is also part of the framework. The framework makes user-controlled layout behavior (Chapter 6) possible. Furthermore, it enables layout suggestions and automatic ad-hoc layout (Chapter 7).
In addition, the framework provides a test environment, which enables the specification of unit tests for layout patterns. In this test environment, it is possible to measure performance. In addition, layout modifications that are performed by the layout patterns can be visualized. For instance, the performance tests presented in Chapter 9 were defined with the help of this test environment, and also the figures shown in Chapter 9 were generated with the help of this test environment.
The framework comprises a library of layout patterns and unit tests for these

patterns. Amongst others, all layout patterns described in Chapter 8 are part of the framework, and for each of these patterns, a unit test exists.

A.2 Integration into Diagram Editors

In this section, the integration of a layout engine, which is created with the help of the layout framework, into DiaMeta editors as well as into GEF editors is described. First of all, the transformation between the LM and the PMs is described in general. Afterwards, special details are given for the two types of editors.

A.2.1 Transformation

The integration of the layout engine into a diagram editor implicates the transformation between different models, namely the LM and the different PMs. As already described in Chapter 4, this transformation is hand-coded, and is not part of the framework.
A transformation is also required at another point, namely inside the layout patterns themselves:

- Graph drawing libraries require a certain input format. For that purpose, a "transformation" between the Graph PM and the input format of the algorithms that are included in the Jung library and a "transformation" between the Graph PM and the input format of the algorithms that are included in the yFiles library was defined.

- Constraint solvers also require a certain input format. For that purpose, a "transformation" between the PM and the input format of the constraint solver was defined. E.g. in case of the equal horizontal distance pattern, a transformation between the List PMM and the input format for the QOCA constraint solver was defined.

All these transformations could be standardized, and could potentially be defined via a model transformation language, as already mentioned in the future work section of Chapter 4.

A.2.2 Integration into DiaMeta Editors

The layout framework was initially designed for DiaMeta editors. In DiaMeta, the abstract syntax of a visual language is defined via an Ecore meta-model.

Architecture

An overview of the architecture is shown in Figure A.1. In a DiaMeta editor, an editor user, first of all, draws a diagram. Internally, this diagram is represented by a graph model. This graph model is transformed into a so-called instance graph. Then, an ASMM instance is created for this instance graph. This is done by solving a constraint satisfaction problem, as described in [87]. Finally, a CSMM instance is created from the diagram itself. The ASM together with the CSM forms the LM, which forms the basis for layout computation. Based on the LM, layout pattern instances are created. Some of these instances are automatically created, while others are created by the editor user. The layout engine gets the LM as well as all layout pattern instances as input. After layout computation, all attribute changes are collected, and the diagram is updated. Based on the changed diagram, the graph model, the instance graph, the ASM and the CSM are updated automatically.

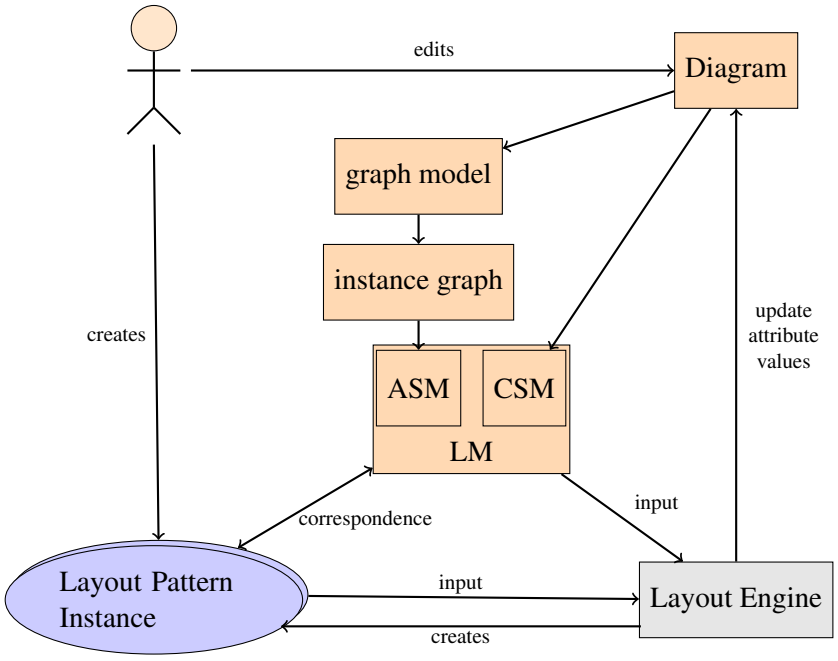


Figure A.1: Integration into DiaMeta

Challenges

In the following paragraphs, some challenges are described that occurred during the integration of the layout engine into DiaMeta editors.

Extension of the GUI First of all, the GUI of the editor had to be extended. The extensions were already described in detail in Chapter 6 and Chapter 7 and may be categorized as follows:

- GUI elements that enable the creation of user-controlled layout pattern instances.
- The visualization of pattern instances in a list as well as in the diagram itself.
- A menu that allows setting of several options.

Freehand Editing One characteristic of DiaMeta that had to be carefully taken into account is freehand editing. As already mentioned in Chapter 6, a special syntax preservation mode was introduced in order to preserve the syntax during user modifications and during layout adjustments.
The presence of freehand editing in a diagram editor also implies that a diagram may be (partially) incorrect. In DiaMeta, this means that this part of the diagram cannot be "transformed" into a LMM instance. As a consequence, this part of the diagram cannot be beautified by the layout approach presented in this thesis. Coping with (partially) incorrect diagrams is up to future work.

Editor Generation Another important characteristic of DiaMeta is that it is an editor generation framework. The only part that had to be implemented by hand is the layout engine. With the layout approach described in this thesis, a step forward is taken, and the implementation of the layout engine is simplified: First of all, layout patterns can be reused, and secondly, new layout patterns can be defined quite easy. As a next step, the layout engine could even (partially) be generated.

Save and Load It turned out that it makes sense that the layout pattern instances the user created are stored when the diagram is saved. Therefore, the save mechanism of DiaMeta was extended as follows:

- The diagram itself is stored in an XML file. This file remains unchanged.
- Layout pattern instances are stored in a separate XML file. The information needing to be stored is the type of the pattern instance and the diagram components this pattern instance is applied to.

A.2.3 Integration into GEF Editors

The layout framework is not restricted to being used in DiaMeta editors, but can be used in any visual language editor. In order to prove this, the layout framework was used for the creation of the layout engine of a GEF editor. In [59], the GEF editor itself, and the integration of the layout engine into this editor is described. Furthermore, some newly developed layout patterns are introduced.

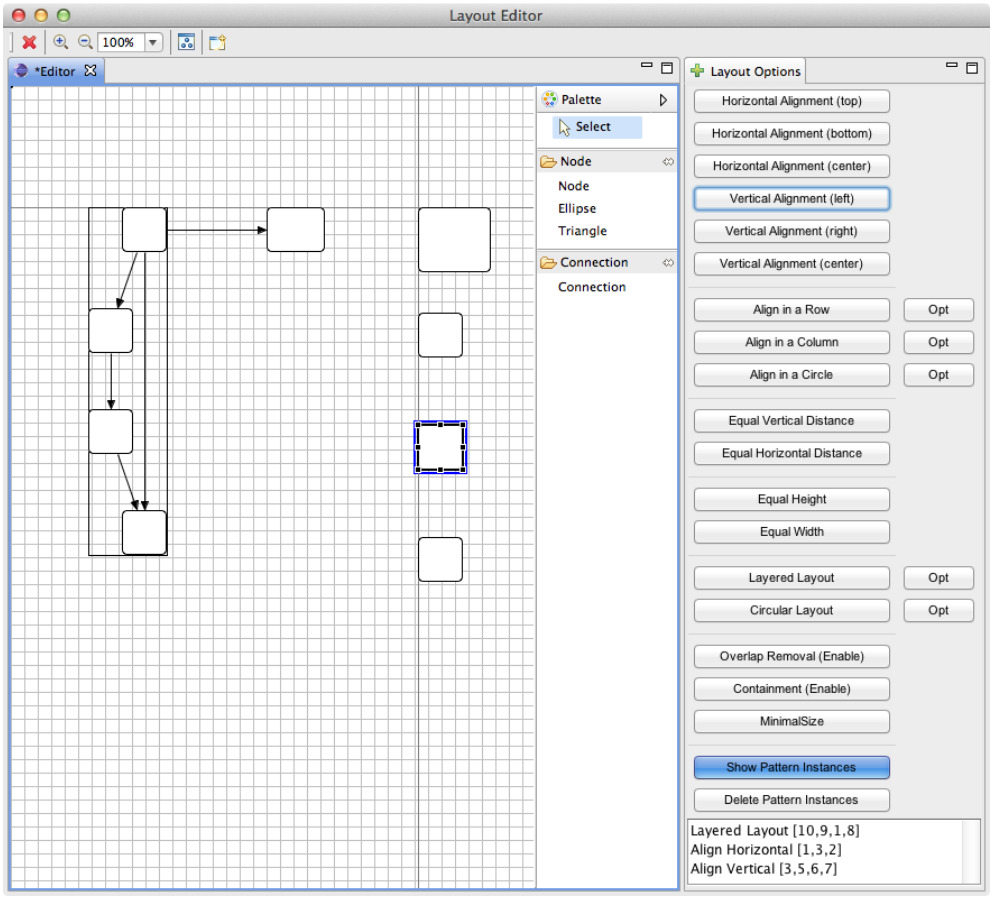


Figure A.2: GEF Graph Editor

GEF is a framework that enables the creation of visual language editors for the Eclipse Workbench. The editor that was created with this framework is a simple graph editor. A screenshot of the editor can be seen in Figure A.2. The architecture of a GEF editor follows the model-view-controller pattern (MVC pattern). In the graph editor, the model is defined via an Ecore meta-model. This design decision was made in order to allow for a straightforward creation of the layout engine.

Architecture

An overview of the architecture is shown in Figure A.3. In a GEF editor, a user, first of all, draws a diagram. Internally, the diagram is represented by an Ecore meta-model instance. Based on the Ecore model, layout pattern instances are created. Some of these instances are automatically created, others are created by the editor user. The layout engine gets the Ecore model as well as all layout pattern instances as input. After layout computation, all attribute changes are collected, and the Ecore model is updated. Based on the changed Ecore model, the diagram is updated automatically.

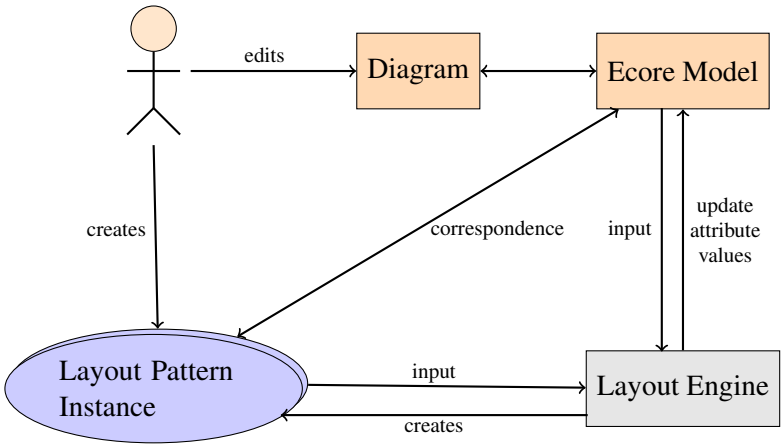


Figure A.3: Integration into GEF

Challenges

In the following paragraphs, some challenges are described that occurred during the integration of the layout engine into GEF editors.

Extension of the GUI As for a DiaMeta editor, the GUI of the editor needs to be extended. The only difference is that no syntax preservation mode is needed in the GEF editor, as GEF editors do not support freehand editing. The visualization of layout pattern instances in the diagram itself turned out to be a bit more complicated than expected. The visualization of the layout pattern instances cannot be drawn into the diagram itself, as this would imply that this visualization would be treated as a diagram component. Hence, this would, for instance, imply that this "component" can be selected, and that handles are created for this "component". Instead, the visualization of pattern instances has to be drawn into a new layer.

Diagram Preview during Modification In GEF editors, only a preview of the diagram is shown while the editor user modifies the diagram, not the diagram itself. As a consequence, the layout of the diagram is not updated during modification. This behavior led to some usability issues, as the editor user was not able to see the layout adjustments that were triggered by his or her diagram modifications. To cope with this challenge, the GEF editor was modified in a sense that the diagram itself is now updated during modification.

Components and Connections The GEF framework is designed for the creation of editors for graph-based visual languages. Therefore, components and connections are treated differently in this framework. As a consequence, up to this point, layout patterns that modify connectors are not possible. In order to support this type of layout patterns, the GEF editor would need to be modified. One possibility would be to define edges as components instead of as connections.

Editor Implementation vs. Editor Generation The GEF editor described in this section is implemented, not generated from an abstract specification as it is done in case of the DiaMeta editor. Hence, the creation of such an editor is rather time consuming.
Graphiti [115] is based on EMF and GEF. It enables an easy creation of homogeneous graphical editors. Therefore, it bridges the gap between GEF and EMF and tries to minimize the complexity of GEF by offering predefined features. The layout approach described in this thesis can also be used for such an editor, and already fulfills the aim of being quite easy.
The Graphical Modeling Framework (GMF) [31] is also based on EMF and GEF. It allows for the generation of GEF editors. The layout approach described in this thesis can also be used for such an editor.

A.3 Summary

In this chapter, some specification and implementation details were given. Details about the layout framework itself were given in Section A.1. In Section A.2, some details were given about the integration of a layout engine, which is defined by the help of this framework, into DiaMeta editors and into GEF editors.
In summary, it can be concluded that a layout engine, which is defined by the help of the layout framework, can be integrated into (almost) any visual

language editor that is implemented in Java. Only minor adaptions of the editor are necessary, mainly affecting the GUI of the editor.

List of Listings

List of Tables

List of Figures

List of Abbreviations

ASM ..... abstract syntax model

ASMM ..... abstract syntax meta model

CSM ..... concrete syntax model

CSMM ..... concrete syntax meta model

CSP ..... constraint satisfaction problem

EMF ..... eclipse modeling framework

GAL ..... global ad-hoc layout

GEF ..... graphical editing framework

GMF ..... graphical modeling framework

LAL ..... local ad-hoc layout

LAL/P ..... local ad-hoc layout with propagation

LM ..... layout model

LMM ..... layout meta model

TGG ..... triple graph grammar

XML ..... extensible markup language

Bibliography

[1] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[2] Alkhalifa, E. M., and Gaid, K. *Cognitively Informed Intelligent Interfaces: Systems Design and Development.* IGI Global, 2012.

[3] Backen, T. Analyse und Umsetzung von Tripel-Graph-Grammatiken als Hilfsmittel zur Diagrammanalyse. Master's thesis, Universitaet der Bundeswehr Muenchen, 2008.

[4] Badros, G. *Extending interactive graphical applications with constraints.* PhD thesis, University of Washington, 2000.

[5] Badros, G., Borning, A., and Stuckey, P. J. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer Human Interaction 8* (2001), 267–306.

[6] Bier, E. A., and Stone, M. C. Snap-dragging. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'86)* (1986), ACM, pp. 233–240.

[7] Biermann, E., Ehrig, K., Ermel, C., and Taentzer, G. Generating Eclipse Editor Plug-Ins Using Tiger. In *Proceedings of Applications of Graph Transformations with Industrial Relevance* (2008), vol. 5088 of *LNCS*, Springer-Verlag, pp. 583–584.

[8] Borning, A. *ThingLab - A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979.

[9] Borning, A., Freeman-Benson, B., and Wilson, M. Constraint Hierarchies. *LISP and symbolic computation 5*, 3 (1992), 223–270.

[10] Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. Solving linear arithmetic constraints for user interface applications. In *Proceedings*

of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST'97) (1997), ACM, pp. 87–96.

[11] Bottoni, P., and Costagliola, G. On the definition of visual languages and their editors. In *Proceedings of Dagrammatic Representation and Inference* (2002), vol. 2317 of *LNCS*, Springer-Verlag, pp. 337–396.

[12] Bottoni, P., and Grau, A. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)* (2004), IEEE Computer Society, pp. 83–90.

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

[14] Bottoni, P., Guerra, E., and de Lara, J. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information and Software Technology 52*, 8 (2010), 821–844.

[15] Brand, S. *Rule-Based Constraint Propagation.* PhD thesis, University of Amsterdam, 2004.

[16] Branke, J. Dynamic graph drawing. *Drawing Graphs, Methods and Models* (2001).

[17] Brieler, F. *A Generic Approach to the Recognition and Analysis of Sketched Diagrams Using context Information.* PhD thesis, Universitaet der Bundeswehr Muenchen, 2010.

[18] Brieler, F., and Minas, M. Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '08)* (2008), ACM, pp. 181–188.

[19] Brieler, F., and Minas, M. A model-based recognition engine for sketched diagrams. *Journal of Visual Languages and Computing 21*, 2 (2010), 81–97.

[20] Citrin, W., Hall, R., and Zorn, B. Programming with visual expressions. In *Proceedings of the 11th International IEEE Symposium on*

Visual Languages (VL'95) (1995), IEEE Computer Society, pp. 294–301.

[21] DE LARA, J., AND GUERRA, E. Pattern-based Model-to-Model Transformation. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT'08)* (2008), vol. 5214 of *LNCS*, Springer-Verlag, pp. 426–441.

[22] DELANEY, A., PLIMMER, B., STAPLETON, G., AND RODGERS, P. Recognizing Sketches of Euler Diagrams Drawn with Ellipses. In *Proceedings of 16th International Conference on Distributed Multimedia Systems (DMS'10)* (2010).

[23] DOKULIL, J., AND KATRENIAKOVA, J. Edge Routing with Fixed Node Positions. In *Proceedings of the 12th International Conference Information Visualisation (IV '08)* (2008), IEEE Computer Society, pp. 626–631.

[24] DONG, J., YANG, S., AND ZHANG, K. Visualizing Design Patterns in Their Applications and Compositions. *IEEE Transactions on Software Engineering 33*, 7 (2007), 433–453.

[25] DUBE, D. Graph Layout for Domain-Specific Modeling. Master's thesis, McGill University, 2006.

[26] DWYER, T., KOREN, Y., AND MARRIOTT, K. IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 821–828.

[27] DWYER, T., LEE, B., FISHER, D., QUINN, K. I., ISENBERG, P., ROBERTSON, G., AND NORTH, C. A Comparison of User-Generated and Automatic Graph Layouts. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 961–968.

[28] DWYER, T., MARRIOTT, K., AND STUCKEY, P. Fast node overlap removal. In *Proceedings of Graph Drawing 2006 (GD'06)* (2006), vol. 3843 of *LNCS*, Springer-Verlag, pp. 153–164.

[29] DWYER, T., MARRIOTT, K., AND WYBROW, M. Dunnart: A constraint-based network diagram authoring tool. In *Proceedings of Graph Drawing 2008 (GD'08)* (2009), vol. 5417 of *LNCS*, Springer-Verlag, pp. 420–431.

[30] Eclipse: GEF. *http://www.eclipse.org/gef/*. 2012.

[31] Eclipse: GMF. *http://www.eclipse.org/modeling/gmp/*. 2012.

[32] Eclipse: ZEST. *http://www.eclipse.org/gef/zest/*. 2012.

[33] Ehrig, K., Ermel, C., Haensgen, S., and Taentzer, G. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)* (2005), ACM, pp. 134–143.

[34] Eiglsperger, M. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach*. PhD thesis, Eberhard-Karls-Universitaet Tuebingen, 2003.

[35] Eiglsperger, M., Siebenhaller, M., and Kaufmann, M. An efficient implementation of sugiyama's algorithm for layered graph drawing. In *Proceedings of Graph Drawing 2004 (GD'04)* (2004), vol. 3383 of *LNCS*, Springer-Verlag, pp. 155–166.

[36] Eysholdt, M., and Behrens, H. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM conference on Systems, Programming, Languages and Applications: Software for Humanity (Companion) (SPLASH'10)* (2010), ACM.

[37] Fish, A., Khazaei, B., and Roast, C. User-comprehension of Euler diagrams. *Journal of Visual Language and Computing 22*, 5 (2011), 340–354.

[38] France, R. B., Kim, D.-K., Ghosh, S., and Song, E. A UML-based Pattern Specification Technique. *IEEE Transactions on Software Engineering 30*, 3 (2004), 193–206.

[39] Freeman-Benson, B. N., and Maloney, J. The DeltaBlue algorithm: An incremental constraint hierarchy solver. In *Proceedings of 8th Annual International Phoenix Conference on Computers and Communications* (1989), IEEE Computer Society, pp. 538–542.

[40] Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver. *Communications of the ACM 33*, 1 (1990), 54–63.

[41] Frisch, M., Heydekorn, J., and Dachselt, R. Investigating Multi-Touch and Pen Gestures for Diagram Editing on Interactive Surfaces.

In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS'09)* (2009), ACM, pp. 149–156.

[42] Frisch, M., Kleinau, S., Langner, R., and Dachselt, R. Grids & guides: multi-touch layout and alignment tools. In *Proceedings of the 2011 annual conference on human factors in computing systems (CHI'11)* (2011), ACM, pp. 1615–1618.

[43] Fuhrmann, H., Spoenemann, M., Matzen, M., and von Hanxleden, R. Automatic Layout and Structure-Based Editing of UML Diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)* (2010).

[44] Gaertner, S. Konzeptionierung und Umsetzung eines Grapheditors für das iPhone. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2009.

[45] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[46] Gansner, E., and Hu, Y. Efficient node overlap removal using a proximity stress model. In *Proceedings of Graph Drawing 2009 (GD'09)* (2009), vol. 5417 of *LNCS*, Springer-Verlag, pp. 206–217.

[47] Gansner, E. R., and Hu, Y. Efficient, Proximity-Preserving Node Overlap Removal. *Journal of Graph Algorithms and Applications 14*, 1 (2010), 53–74.

[48] GraphML. *http://graphml.graphdrawing.org/.* 2012.

[49] GraphViz. *http://www.graphviz.org/.* 2012.

[50] Hammond, T., and Davis, R. LADDER, a sketching language for user interface developers. *Computer and Graphics 29*, 4 (2005), 518–532.

[51] Jarasch, G., Maier, S., Kingsbury, P., Minas, M., and Schulte, A. Design methodology for an artificial cognitive system applied to human-centred semi-autonomous uav guidance. In *Proceedings of the Second International Conference on Humans Operating Unmanned Systems (HUMOUS'10)* (2010).

[52] Jung, S. Erweiterung von DiaGen-DiaMeta um die Möglichkeit der Nutzung mehrerer Sichten. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2009.

[53] KELLY, S. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)* (2004), ACM.

[54] KINDLER, E., AND WAGNER, R. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Tech. rep., Universitaet Paderborn, 2007.

[55] KOETH, O., AND MINAS, M. Structure, abstraction, and direct manipulation in diagram editors. In *Proceedings of the 2nd International Conference on Diagrammatic Representation and Inference (Diagrams'02)* (2002), vol. 2317 of *LNCS*, Springer-Verlag, pp. 290–304.

[56] KRAUSE, F. Layout graphartiger visueller Sprachen. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2010.

[57] LAUDER, A., AND KENT, S. Precise Visual Specification of Design Patterns. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP'98)* (1998), vol. 1445 of *LNCS*, Springer-Verlag, pp. 114–134.

[58] LECAT, A.-C. Umsetzung des Spiels Pacman mithilfe von DiaMeta. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2008.

[59] LEWERENZ, P. Identifikation und Realisierung von Layout-Funktionalitaet in Diagramm-Editoren. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2012.

[60] LI, W., EADES, P., AND NIKOLOV, N. Using spring algorithms to remove node overlapping. In *Proceedings of the 2005 Asia-Pacific Symposium on Information Visualisation (APVis'05)* (2005), pp. 131–140.

[61] MAIER, S., MAZANEK, S., AND MINAS, M. Layout Specification on the Concrete and Abstract Syntax Level of a Diagram Language. In *Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED'08)* (2008), vol. 13, Electronic Communications of the EASST.

[62] MAIER, S., MAZANEK, S., AND MINAS, M. Visual Specification of Layout. In *Proceedings of Graph Drawing 2009 (GD'09)* (2009), vol. 5417 of *LNCS*, Springer-Verlag, pp. 443–444.

[63] Maier, S., and Minas, M. A Pattern-Based Layout Algorithm for Diagram Editors. In *Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED'07)* (2007), vol. 7, Electronic Communications of the EASST.

[64] Maier, S., and Minas, M. A Generic Layout Algorithm for Meta-Model Based Editors. In *Proceedings of Applications of Graph Transformations with Industrial Relevance* (2008), vol. 5088 of *LNCS*, Springer-Verlag, pp. 66–81.

[65] Maier, S., and Minas, M. A Static Layout Algorithm for DiaMeta. In *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08)* (2008), vol. 10, Electronic Communications of the EASST.

[66] Maier, S., and Minas, M. Ludo meets DiaMeta. In *Preliminary Proceedings of Applications of Graph Transformations with Industrial Relevance* (2008), vol. 5088 of *LNCS*, Springer-Verlag, pp. 493–513.

[67] Maier, S., and Minas, M. Pattern-Based Layout Specifications for Visual Language Editors. In *Proceedings of the 1st International Workshop on Visual Formalisms for Patterns* (2009), Electronic Communications of the EASST.

[68] Maier, S., and Minas, M. Rule-based Diagram Layout using Meta Models. In *Proceedings of the International Conference on Distributed Multimedia Systems (DMS'09)* (2009).

[69] Maier, S., and Minas, M. Specification of a Drawing Facility for Diagram Editors. In *Proceedings of the 13th International Conference on Human-Computer Interaction* (2009), vol. 5611 of *LNCS*, Springer-Verlag, pp. 850–859.

[70] Maier, S., and Minas, M. Combination of Different Layout Approaches. In *Proceedings of the 2nd International Workshop on Visual Formalisms for Patterns* (2010), vol. 31, Electronic Communications of the EASST.

[71] Maier, S., and Minas, M. Interactive diagram layout. In *Proceedings of the 2010 annual conference on human factors in computing systems (CHI'10)* (2010), ACM, pp. 4111–4116.

[72] Maier, S., and Minas, M. Integration of a Pattern-based Layout Engine into Diagram Editors. In *Proceedings of Applications of Graph*

Transformations with Industrial Relevance (2012), Springer-Verlag (to appear).

[73] Maier, S., and Minas, M. Layout Improvement in Diagram Editors by Automatic Ad-hoc Layout. In *Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'12)* (2012), vol. 47, Electronic Communications of the EASST.

[74] Maier, S., and Volk, D. Facilitating language-oriented game development by the help of language workbenches. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share* (2008), ACM, pp. 224–227.

[75] Mak, J. K. H., Choy, C. S. T., and Lun, D. P. K. Precise Modeling of Design Patterns in UML. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)* (2004), IEEE Computer Society, pp. 252–261.

[76] Mapelsden, D., Hosking, J., and Grundy, J. Design pattern modelling and instantiation using DPML. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications (CRPIT '02)* (2002), Australian Computer Society, Inc, pp. 3–11.

[77] Marriott, K., Chok, S. S., and Finlay, A. A Tableau Based Constraint Solving Toolkit for Interactive Graphical Applications. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP'98)* (1998), vol. 1520 of *LNCS*, Springer-Verlag, pp. 340–354.

[78] Masui, T. HyperSnapping. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)* (2001), IEEE Computer Society, pp. 188–194.

[79] Mazanek, S. *Exploiting hypergraph grammars for the realization of syntax-based user assistance in diagram editors.* PhD thesis, Universitaet der Bundeswehr Muenchen, 2010.

[80] Mazanek, S., Maier, S., and Minas, M. An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT'08)* (2008), vol. 5214 of *LNCS*, Springer-Verlag, pp. 39–53.

[81] MAZANEK, S., MAIER, S., AND MINAS, M. Auto-completion for diagram editors based on graph grammars. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)* (2008), IEEE Computer Society, pp. 242–245.

[82] MAZANEK, S., MAIER, S., AND MINAS, M. Exploiting the Layout Engine to Assess Diagram Completions. In *Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED'08)* (2008), vol. 13, Electronic Communications of the EASST.

[83] MAZANEK, S., AND MINAS, M. Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (Models'09)* (2009), vol. 5795 of *LNCS*, Springer-Verlag, pp. 322–336.

[84] META OBJECT GROUP. *Meta Object Facility (MOF) Core Specification*. 2006.

[85] MINAS, M. *Spezifikation und Generierung graphischer Diagrammeditoren*. PhD thesis, Friedrich-Alexander-Universitaet Erlangen-Nuernberg, 2001.

[86] MINAS, M. Generating Meta-Model-Based Freehand Editors. In *Proceedings of the International Workshop on Graph-based Tools (GraBaTs'06)* (2006), vol. 1, Electronic Communications of the EASST.

[87] MINAS, M. Syntax analysis for diagram editors: a constraint satisfaction problem. In *Proceedings of the working conference on Advanced visual interfaces (AVI'06)* (2006), ACM, pp. 167 – 170.

[88] MINAS, M., AND STRÜBER, F. Unparsing of Diagrams with DiaGen. In *Proceedings of the First International Conference on Graph Transformation (ICGT'02)* (2002), vol. 2505 of *LNCS*, Springer-Verlag, pp. 302–316.

[89] NASSI, I., AND SHNEIDERMAN, B. Flowchart techniques for structured programming. *ACM SIGPLAN Notices 8*, 8 (1973), 12–26.

[90] O'MADADHAIN, J., FISHER, D., SMYTH, P., WHITE, S., AND BOEY, Y. Analysis and Visualization of Network Data using JUNG. *Journal of Statistical Software 10* (2005), 1–35.

[91] Plimmer, B., Purchase, H., Yang, H. Y., Laycock, L., and Milburn, J. Preserving the Hand-drawn Appearance of Graphs. In *Proceedings of the International Conference on Distributed Multimedia Systems (DMS'09)* (2009).

[92] Purchase, H., and Samra, A. Extremes Are Better: Investigating Mental Map Preservation in Dynamic Graphs. In *Proceedings of the 5th International Conference on Diagrammatic Representation and Inference* (2008), vol. 5223 of *LNCS*, Springer-Verlag, pp. 60–73.

[93] Purchase, H. C., McGill, M., Colpoys, L., and Carrington, D. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In *Proceedings of the 2001 Asia-Pacific Symposium on information Visualization (APVis'01)* (2001), Australian Computer Society, Inc, pp. 129–137.

[94] Rensink, A., Dotor, A., Ermel, C., Jurack, S., Kniemeyer, O., Lara, J., Maier, S., Staijen, T., and Zündorf, A. Ludo: A Case Study for Graph Transformation Tools. In *Proceedings of Applications of Graph Transformations with Industrial Relevance* (2008), vol. 5088 of *LNCS*, Springer-Verlag, pp. 493–513.

[95] Ryall, K., Marks, J., and Shieber, S. An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST '97)* (1997), ACM, pp. 97–104.

[96] Sannella, M. Skyblue: A Multi-way Local Propagation Constraint Solver for User Interface Construction. In *Proceedings of ACM Symposium on User Interface Software and Technology (UIST'04)* (1994), ACM, pp. 137–146.

[97] Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. *Software: Practice and Experience 23*, 5 (1993), 529–566.

[98] Schmidt, C. *Generierung von Strukturediteren fuer anspruchsvolle visuelle Sprachen.* PhD thesis, Universitaet Paderborn, 2006.

[99] Schuerr, A. Specification of Graph Translators with Triple Graph Grammars. *Graph-Theoretic Concepts in Computer Science 903* (1995), 151–163.

[100] Seelisch, A.-C. Musterbasierte Spezifikation des Diagrammlayouts in Editoren fuer visuelle Sprachen. Master's thesis, Universitaet der Bundeswehr Muenchen, 2009.

[101] Stapleton, G., and Rodgers, P. Drawing Euler Diagrams with Circles and Ellipses. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)* (2011), IEEE Computer Society, pp. 209–212.

[102] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[103] Stoerrle, H. On the Impact of Layout Quality to Understanding UML Diagrams. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)* (2011), IEEE Computer Society, pp. 135–142.

[104] Strobl, T., and Minas, M. Specifying and generating editing environments for interactive animated visual models. In *Proceedings of the 9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10)* (2010), vol. 29, Electronic Communications of the EASST.

[105] Strobl, T., Minas, M., Pleuss, A., and Vitzthum, A. From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs'10)* (2010), vol. 32, Electronic Communications of the EASST.

[106] Sutherland, I. E. Sketchpad: A man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop (DAC'64)* (1964).

[107] The Choco Team. Choco: an Open Source Java Constraint Programming Library. In *Proceedings of the CPAIOR Workshop on Open-Source Software for Integer and Constraint Programming* (2008).

[108] Tidwell, J. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, 2005.

[109] Tollis, I., Eades, P., Di Battista, G., and Tamassia, R. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[110] Turgenev, I. *Fathers and Sons.* 1862.

[111] Van Welie, M., and Van der Veer, G. Pattern languages in interaction design: Structure and organization. In *Proceedings of Interact'03* (2003).

[112] Von Pilgrim, J., Duske, K., and McIntosh, P. Eclipse GEF3D: Bringing 3D to existing 2D editors. *Information Visualization 8*, 2 (2009), 107–119.

[113] Voss, V. Dreidimensionale Darstellung zweidimensionaler visueller Sprachen. Master's thesis, Universitaet der Bundeswehr Muenchen, 2008.

[114] Ware, C., Purchase, H., Colpoys, L., and McGill, M. Cognitive measurements of graph aesthetics. *Information Visualization 1*, 2 (2002), 103–110.

[115] Wenz, M. Graphiti Building Graphical Editors the Easy Way. In *EclipseCon'11* (2011).

[116] Wettberg, T. Verwendung eines Constraint Solvers zur Beschreibung des Layouts modellbasierter Editoren. Bachelor's thesis, Universitaet der Bundeswehr Muenchen, 2010.

[117] Wiese, R., Eiglsperger, M., and Kaufmann, M. yFiles: Visualization and Automatic Layout of Graphs. In *Proceedings of Graph Drawing 2002 (GD'02)* (2002), vol. 2265 of *LNCS*, Springer-Verlag, pp. 588–590.

[118] Wybrow, M. *Using semi-automatic layout to improve the usability of diagrammnig software.* PhD thesis, Monash University, 2008.

[119] Wybrow, M., Marriott, K., Mciver, L., and Stuckey, P. Comparing usability of one-way and multi-way constraints for diagram editing. *ACM Transactions on Computer-Human Interaction 14*, 4 (2008).

[120] Zanden, B. T. V., Halterman, R., Myers, B. A., Miller, R., Szekely, P., Giuse, D. A., Kosbie, D., and McDaniel, R. Lessons learned from programmers' experiences with one-way constraints. *Software: Practice and Experience 35*, 13 (2005), 1275–1298.