Electronic Communications of the EASST
Volume 7 (2007)

Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)

A Pattern-Based Layout Algorithm for Diagram Editors

Sonja Maier and Mark Minas

16 pages

Guest Editors: Andrew Fish, Alexander Knapp, Harald Störrle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Pattern-Based Layout Algorithm for Diagram Editors

Sonja Maier [1] **and Mark Minas** [2]

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

Abstract: The diagram editor generator framework DIAMETA utilizes meta-model-based language specifications and supports *free-hand* as well as *structured editing*. We presented a *generic layout algorithm* that meets the demands of this kind of editors. The algorithm combines two concepts, constraint satisfaction and attribute evaluation, to a powerful methodology for specifying the *layout* for a particular visual language. As the *layout* specification for this algorithm is rather complex, we encapsulated basic functionality into reusable *patterns*. This paper describes this *pattern* concept of the *generic layout algorithm*, and shows how they simplify the *layout* specification of a specific language.

Keywords: Pattern, Constraint Satisfaction, Attribute Evaluation, Visual Language, Free-hand Editing, Structured Editing

1 Introduction

Several approaches and tools have been proposed to specify visual languages and to generate editors from such specifications. These attempts can be characterized by the way the diagram language is specified and by the way the user interacts with the editor and creates respectively edits diagrams. Most visual languages have a meta-model as (abstract) syntax specification. A model is essentially a class diagram of the data structure that is visualized by a diagram. When considering user interaction and the way how the user can create and edit diagrams, *structured editing* is usually distinguished from *free-hand editing*. *Structured editors* offer the user some operations that transform correct diagrams into (other) correct diagrams. *Free-hand editors*, on the other hand, allow to arrange diagram components from a language-specific set on the screen without any restrictions. The editor has to check whether the drawing is correct and what its meaning is. In both cases, a *layouter* may be used to beautify the diagram. In *free-hand mode*, the editor user has more freedom, which implies that the layouter is more complex.

In [MM07] we designed a *generic layout algorithm* that works for model-based visual languages. It meets the demands of *structured* as well as *free-hand editing*. Our algorithm was designed for the framework DIAMETA, that follows the model-driven approach to specify diagram languages. From such a specification an editor, offering *structured* as well as *free-hand editing*, can be generated. In Fig. 1 we can see an editor that was generated with DIAMETA.

For *structured editors*, *layout algorithms* were studied in the past [CMP99]. For *free-hand editors*, these *layout algorithms* cannot be applied in a straightforward way - the *layouter* has to deal with the increase of flexibility and should restrict the user only in a moderate way. In

the world of grammar-based editors, some *layout algorithms* have been established in the past [Min04]. Our *layout algorithm* operates on a meta model instead. It allows for defining a *layout* that is specialized for a certain model, i.e. a certain visual language.

One frequently used concept is attribute evaluation. An attribute evaluator is fast and best suited if the *layout* is unambiguous. This concept cannot deal with the situation that the same diagram may be represented in different ways. Especially in *free-hand mode*, a conventional attribute evaluator is not sufficient. Another concept that is frequently used for *layout* [Min04, CMP99] is constraint satisfaction. The disadvantages of this concept are that constraint satisfaction is slow in some cases and its behavior is unpredictable in some situations.

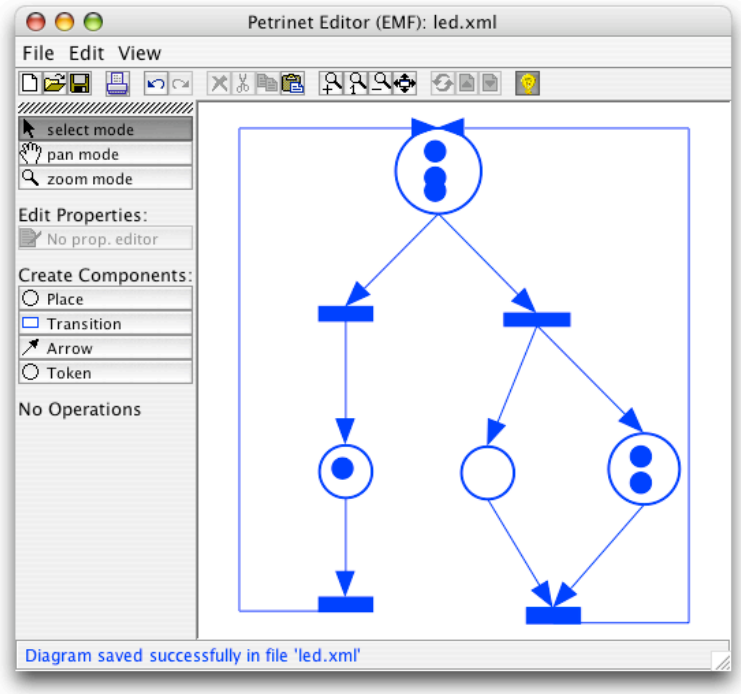


Figure 1: Petri net editor

In [MM07] we presented an algorithm that combines the two concepts, constraint satisfaction and attribute evaluation, to a powerful algorithm that is fast, flexible and behaves exactly the way we desire: Declarative constraints ensure the characteristics of the *layout*. If they are not fulfilled, a set of certain attribute evaluation rules is switched on. These rules are evaluated, and the associated attributes are updated.

We realized that writing such a specification is rather complicated and complex. Therefore we encapsulated basic functionality as packages, as it is done in [SK03, Sch06], and give the user the opportunity to use (and reuse) these packages. They contain a set of constraints and corresponding attribute evaluation rules that are tailored to a specific problem, e.g. to the problem of arranging arrows in a graph-based visual language. These packages are called *patterns* in the following, the terminology used in [SK03, Sch06]. These *patterns* introduce another level

of abstraction on top of the specification, as *design patterns* [GHJV95] do for object-oriented software design. In order to use such a predefined *pattern*, the model must contain some special components, e.g. for the *GraphPattern*, the model must contain a class representing edges and a class representing nodes.

For most visual languages, standard layout algorithms may be specified, using predefined *patterns*. Using them simplifies the *layout* specification. If the predefined *patterns* are not sufficient, e.g. for unusual visual languages or a fancy layout, the *patterns* may be adjusted to the special needs or new *patterns* may be created. And of course it is also possible to use the algorithm in the traditional way and benefit from the complete functionality the *generic layout algorithm* offers.

In Sect. 2 we introduce the model of Petri nets, the visual language that is used as a running example. In Sect. 3 we explain the *generic layout algorithm* that we have proposed for meta model based editors. In Sect. 4 we introduce the *pattern* concept for the *generic layout algorithm*. In Sect. 5 we show how to use this concept to create the *layout* for Petri net editors. Sect. 6 summarizes some implementation details and gives an overview of DIAMETA, the environment in which the *pattern* concept was tested. Sect. 7 concludes the paper.

2 Running Example

In this section we introduce an editor for Petri nets as running example. First we describe the underlying meta model of the Petri net language. Then we explain how the diagram is visualized. Finally we give a short overview of the *layout* that we are going to define throughout the paper.

Each diagram consists of a finite set of visual components. In Petri nets, these are places, transitions, tokens, and arrows between places and transitions. Each component is determined by its attributes.

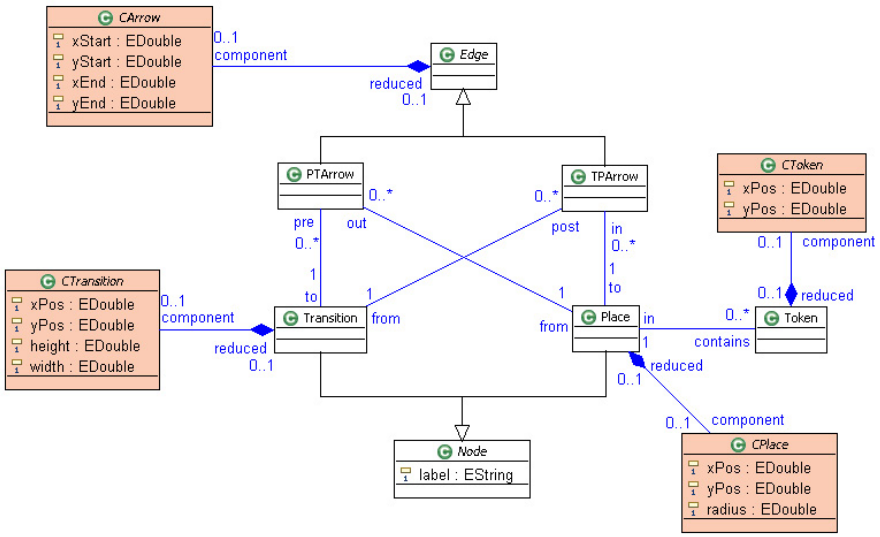


Figure 2: Meta model of Petri nets

Fig. 2 shows the meta model for Petri nets. It contains the class *Node* as an abstract base class of a Petri net's *Place* or *Transition*. The classes *Place* and *Transition* have a member attribute *label*. *Edge* is the abstract base class of a connection between places and transitions. Concrete classes of the abstract model are *Place*, *Transition*, *PTArrow*, *TPArrow*, and *Token* respectively. Transition-Place relations are represented by the associations between *Transition*, *TPArrow* and *Place*, Place-Transition relations by the associations between *Place*, *PTArrow* and *Token*. Place-Token relations are represented by the association between the classes *Place* and *Transition*.

In the meta model, the abstract syntax is described. Besides that, some aspects of the concrete syntax are included. This additional information is needed to perform layout computations. The classes *CPlace*, *CTransition*, *CArrow* and *CToken* represent aspects of the concrete syntax.

A place is visualized by a circle whose center position is determined by its attributes (*xPos*, *yPos*) and its radius by the attribute *radius*. A transition is visualized by a square whose center position is defined by the coordinate point (*xPos*, *yPos*) and its size by the attributes *width* and *height*. A token is visualized by a circle whose center position is again defined by (*xPos*, *yPos*). Its radius is a fixed value that cannot be modified by the user. *PTArrow* and *TPArrow* are visualized by arrows whose position is defined by its two end points, i.e. by two coordinate pairs (*xStart*, *yStart*) and (*xEnd*, *yEnd*).[1] In Fig. 1 we can see a sample Petri net, visualized as described above, and layouted (incrementally) as described in the following.

We are going to specify a *layout* for the Petri net editor that is based on the model presented above. During user interaction, we want to support the user with some special behavior. After user interaction, we want to get a beautified diagram as result.

- *After user interaction:* Arrows start and end exactly at the border of a component, i.e. exactly at the border of a transition or place. Arrows must have a minimal length, i.e. the components must have a minimal distance. Tokens are completely inside a place. They may not intersect the border line of the place. If possible, tokens are arranged as a list, as long as the list fits into the place.[2]

- *During user interaction:* When we move a place (or change the size of a place), arrows and tokens have to follow the place. When we move a transition (or change the size of a transition), arrows also have to follow the transition. This gives the user an easy and intuitive way of changing the visual appearance of the Petri net. He may for example rearrange tokens and places without changing the semantics of the diagram. When we move an arrow or token, nothing else is changed. With this functionality, the user may change the dynamic behavior of the Petri net. He may for example move a token from one place to another.

In our specification we make use of three *patterns*. The *GraphPattern* being responsible for *layouting* the arrows, the *ListPattern* that is responsible for arranging the tokens inside a place and the *ContainmentPattern* that ensures that tokens are completely inside the place. To demonstrate the possibilities offered by the concept, we will adjust a *pattern* and we will add some additional functionality that is not supported by the *patterns*.

[1] They can be substituted by a list of bends. The editor that was created via DIAMETA actually supports bends.
[2] This is not the most intuitive layout. This behavior was introduced for explanatory reasons, as we will see later.

3 Generic Layout Algorithm

In Fig. 3 we can see a birds-eye view of the *layout algorithm* that has been presented in [MM07]. The algorithm is based on the idea that we have a set of declarative constraints (and a set of all attributes), that assure the characteristics of the *layout*. If all constraints are satisfied, the *layouter* terminates. If one or more constraints are not satisfied, the *layouter* needs to change some attributes to satisfy the constraints. Therefore it switches on one or more attribute evaluation rules. These rules in turn are responsible for updating the attributes, i.e. to satisfy the constraints.

In this section we describe this *layout algorithm* in more detail. First we describe the input parameters of the *layouter*. Then we summarize what components the *layout specification* consists of. As a last step we describe the *layout algorithm* itself. In the next section we will introduce the *pattern* concept for this *layout algorithm*.

3.1 Input

The algorithm gets as input one or two sets of attribute values - the *old values* (values before user interaction), the *user-desired values* (values after user interaction) or both. Furthermore, the *layouter* is aware of the current state. It knows whether the user is in the process of modifying a component, e.g. is currently moving a place, or has finished a modification already. It also knows, which component(s) the user has changed. In addition, the *layouter* has access to the model of the visual language.

We have to distinguish three types of user interaction: adding, modifying and removing. The selection of *old values* and *user-desired values* depends on the type of user interaction. When the user adds a component at a desired position, the *layouter* gets one set of values as input - the *user-desired values*. When the user modifies a component, e.g. moves a place from the position characterized by $xPos_{old}$ and $yPos_{old}$ to a new position $xPos_{user}$ and $yPos_{user}$, the *layouter* has two sets of values as input - the *old values* and the *user-desired values*. In case of deletion, the *layouter* gets only the *old values* as input.

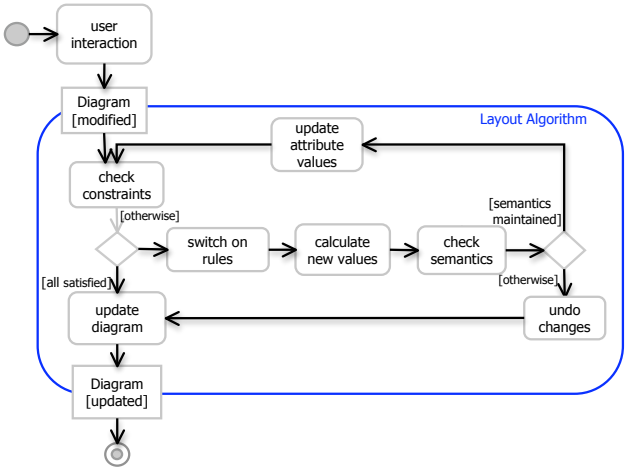


Figure 3: Birds-eye view of the *generic layout algorithm*

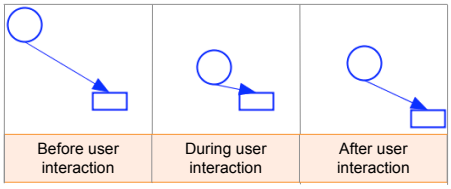


Figure 4: Moving a place

We distinguish between two states, *during modification* and *after modification* that we treat in different ways. *During modification* only the layouter is called, *after modification* first the model is updated and then the *layouter* is called, using the updated model. *During modification*, some *layouting* constraints should be satisfied immediately. The satisfaction of other constraints may be postponed to the end of the user interaction. Suppose we change the position of a place, as we can see in Fig 4. While we move the component (*during modification*), we want arrows to follow the place. As the *layouter* is responsible for updating the attributes, he needs to be called several times *during modification* of the diagram via user input in order to update the arrows. After we finished moving the place, for example, we want to satisfy the constraint that arrows have a minimal length. If an arrow does not satisfy this constraint, it is extended automatically. Minimizing the number of computations during user interaction not only speeds up the computation of the new visualization, it also gives the user more freedom.

Another aspect we take care of is the information, what component, i.e. what attributes, the user changed. In our example we distinguish between moving arrows and moving places or transitions. When we move an arrow, we just want the arrow to be moved. The places and transitions remain unchanged. If we move a place or transition, we want the arrows to remain connected to these components, and hence the arrows are changed.

3.2 Layout Specification

The *layouter* uses the attributes, the state and the model of the visual language to calculate *new values* that represent the updated diagram. To do that, it needs a *layout specification*, as introduced in [MM07]. This specification consists of a set of constraints, each of them associated with a concrete class like *Place* or *PTArrow*. For every constraint there exists a list of attribute evaluation rules. If a constraint is violated, it is its evaluation rules' task to update attributes such that the constraint is satisfied (again).

The constraints and attribute evaluation rules use the standard OCL syntax, as specified in [OMG06]. Only *current values* are changed during execution of the *layout algorithm*. All other attributes remain unchanged. Intermediate results are created each *layout* iteration.

Constraints are responsible for switching on and off attribute evaluation rules. Attribute evaluation rules are responsible for calculating the set of *new values*. For example, constraint (1) switches on rule (2) if $xPos \leq in.xPos$. If this is not the case, $xPos$ remains unchanged.

$$[\text{after modification}]xPos \quad > \quad in.xPos \tag{1}$$

$$xPos \quad \leftarrow \quad in.xPos + 5 \tag{2}$$

We may restrict constraints and attribute evaluation rules to be checked and executed only if we are in a special state (indicated by [state] in front of the constraint or rule). For example, if we add [after modification] in front of the constraint, this constraint is checked after modification. Otherwise, this constraint is checked each time the *layouter* is called.

We may also add [o1 changed] in front of the constraint. This means that the constraint is only executed if one of the attributes of the object o1 has changed.[3]

3.3 Layout Algorithm

In Fig. 3 we can see a birds-eye view of the *generic layout algorithm*. The *layouter* is called each time the diagram was changed via user interaction. The set of *current values* consists of *user-desired values* for the attributes changed via user interaction, and *old values* for attributes the user did not change. All potentially violated *layout* constraints (that need to be checked for the current state) are checked, and the rules that were switched on are collected. Thereafter the *new values* of the attributes are calculated via attribute evaluation.

The *current values* are substituted by the *new values* and the constraints are checked again, since new constraints may have become unsatisfied due to changes performed by the *layouter*. If all constraints are satisfied, the *layouter* succeeds and reports all *new values*. Otherwise, the *layouter* has to evaluate the rules again. If the *layouter* does not succeed after a certain number of iterations (may be user defined), the *layouter* stops and returns the *user values* as result.

4 Pattern Concept for the Layout Algorithm

Creating an editor with DIAMETA is tool supported. The only part the editor developer had to write by hand had been the *layouter*. With the *layout algorithm* presented above, the editor developer is no longer burdened with this task. He now only has to provide a *layout specification*.

We are aware that writing such a specification is still rather complicated and complex. Therefore we encapsulated basic functionality, as it is done in [SK03, Sch06], and give the user the opportunity to use these *patterns*. In Fig. 5 we can see some *patterns* that were already defined. *GraphPattern*, *ContainmentPattern* and *ListPattern* will be explained in the next section, as they form the basis of the *layout specification* for the Petri net editor.

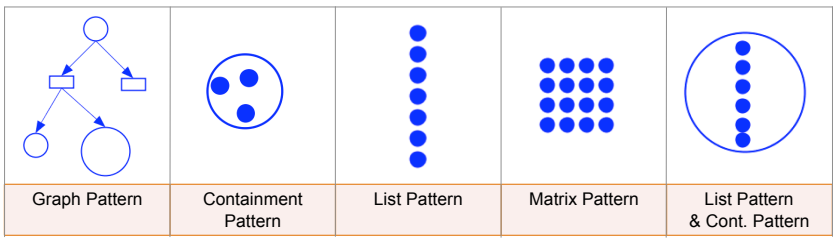


Figure 5: *GraphPattern*, *ContainmentPattern*, *ListPattern*, *MatrixPattern*

[3] Note that not only the editor user may change an object, but also the *layouter* may be responsible for changes.

In order to use these *patterns*, the user simply has to specify which *pattern* he wants to apply on what part of the model. For example, for the *GraphPattern*, he has to specify which component plays the role *Node*, and which component the role *Edge*. For our Petri net editor, places and transitions will play the role *Node* and arrows will play the role *Edge*. In our meta model, places are represented by the two classes *CPlace* and *Place* and transitions by the two classes *CTransition* and *Transition*. Arrows are represented by the two classes *CArrow* and *PTArrow* or by the two classes *CArrow* and *TPArrow*, as shown in Fig. 6.

The editor developer has the opportunity to adjust these *patterns* to his own needs. He may also combine different *patterns*, or refine a *pattern*. Of course he may also add additional functionality or create new *patterns* from scratch.

4.1 Pattern Requirements

A *pattern* contains a set of constraints and corresponding attribute evaluation rules. These constraints and attribute evaluation rules need some associations and attributes for their calculations. Consequently, a *pattern* may only be used if some requirements are fulfilled. In Fig. 6 (in the middle) we see the requirements that need to be met in order to use the *GraphPattern*. There need to be two associations between *Node* and *Edge* with the roles *from* and *to* respectively. *Node* must have the attributes *xPos*, *yPos*, *width* and *height*. *Edge* must have the attributes *xStart*, *yStart*, *xEnd* and *yEnd*.

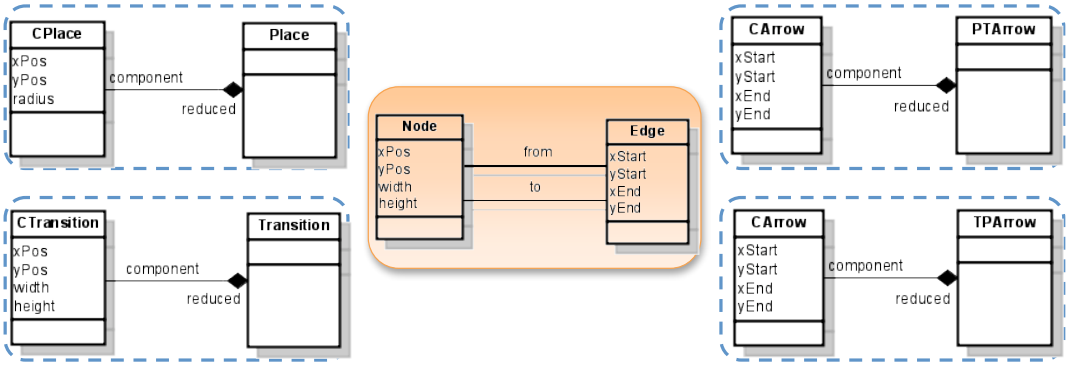


Figure 6: Requirements for the *GraphPattern*

In our example place does not offer the attributes *width* and *height*. All other requirements are already met. We could add these attributes, but we do not want to change the meta model. In this case, the editor developer may introduce a mapping between a required component and another available component. We introduce a bidirectional mapping between *height* and *radius* and a bidirectional mapping between *width* and *radius*:[4]

$$\begin{array}{lcl|lcl} height & \leftarrow & 2*radius & width & \leftarrow & 2*radius \\ radius & \leftarrow & height/2 & radius & \leftarrow & width/2 \end{array}$$

[4] Both directions are required to implement the methods *getWidth()*, *setWidth()*, *getHeight()* and *setHeight()*.

4.2 Pattern usage and Pattern adjustment

If all requirements are met, the *pattern* may be used (*pattern usage*). A *pattern* consists of a set of constraints and attribute evaluation rules. E.g. the *GraphPattern* consists of the following constraints and attribute evaluation rules.

The following four constraints (left side) associated with the classes *PTArrow* and *TPArrow* assure that arrows start and end *exactly* at the top or bottom of a component, as we can see in Fig. 4. (*xPos*, *yPos*) is located in the top left corner of a component. The first (last) two constraints are checked if the component, at which the arrow starts (ends) has changed. The associated attribute evaluation rules (right side) update arrows, if they are not at the right position.

$$\begin{array}{rcl|rcl} \text{[from changed] } xStart & = & from.xPos + \frac{from.width}{2} & xStart & \leftarrow & from.xPos + \frac{from.width}{2} \\ \text{[from changed] } yStart & = & from.yPos + from.height & yStart & \leftarrow & from.yPos + from.height \\ \text{[to changed] } xEnd & = & to.xPos + \frac{to.width}{2} & xEnd & \leftarrow & to.xPos + \frac{to.width}{2} \\ \text{[to changed] } yEnd & = & to.yPos & yEnd & \leftarrow & to.yPos \end{array}$$

To assure that arrows have a minimal length, we introduce a constraint associated with the classes *PTArrow* and *TPArrow*. This constraint is checked after user interaction has finished:

$$\text{[after modification] } (xEnd - xStart)^2 + (yEnd - yStart)^2 > 1000$$

The associated rules extend an arrow, if it is shorter than the minimal length required. If the component, at which the arrow starts (ends) has changed, the component, at which the arrow ends (starts) is moved. As the arrow stays connected to this component, it is automatically extended to the required length.

$$\begin{array}{rcl} \text{[from changed] } to.xPos & \leftarrow & to.xPos_{t(i-1)} + \frac{to.xPos_{t(i-1)} - from.xPos}{|to.xPos_{t(i-1)} - from.xPos|} \\ \text{[from changed] } to.yPos & \leftarrow & to.yPos_{t(i-1)} + \frac{to.yPos_{t(i-1)} - from.yPos}{|to.yPos_{t(i-1)} - from.yPos|} \\ \text{[to changed] } from.xPos & \leftarrow & from.xPos_{t(i-1)} + \frac{from.xPos_{t(i-1)} - to.xPos}{|from.xPos_{t(i-1)} - to.xPos|} \\ \text{[to changed] } from.yPos & \leftarrow & from.yPos_{t(i-1)} + \frac{from.yPos_{t(i-1)} - to.yPos}{|from.yPos_{t(i-1)} - to.yPos|} \end{array}$$

In each *pattern* we introduced some constants. These constants have an initial value and may be overridden by the user (*pattern adjustment*). They are used in the constraints and attribute evaluation rules. E.g. for the *GraphPattern*, the attribute *minLength* (the 1000 in the constraint) may be overridden. This changes the minimal length of an arrow. This mechanism made *pattern* more flexible. Experiments showed that they were now applicable in more situations.

4.3 Pattern combination and Pattern refinement

It is possible to use more than one *pattern* for *layout specification* (*pattern combination*). In our example, we will combine the two *patterns ContainmentPattern* and *ListPattern*. The *ContainmentPattern* will be responsible to keep tokens inside a place. The *ListPattern* is responsible for arranging tokens as a list, if the constraints of the *ContainmentPattern* still can be satisfied.

Right now, *pattern combination* is done by applying the *patterns* one after another.

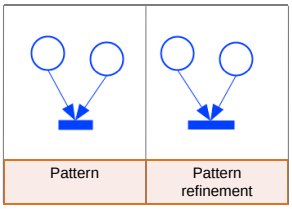


Figure 7: *Pattern refinement*

In our example, first the *ListPattern* is applied, and then the *ContainmentPattern*. This mechanism will be substituted by an enhanced priority concept in future implementations.

We may also add additional constraints and attribute evaluation rules to a *pattern* (*pattern refinement*). In our example we add a constraint that assures that transitions have a minimal width and height (Fig. 7). Up to now, all constraints and attribute evaluation rules are collected. No simplification or error check is utilized. The editor creator has to ensure that constraints and attribute evaluation rules are reasonable.

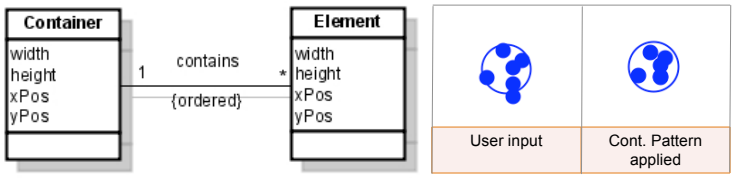
5 Pattern-Based Layout for the Petri net editor

We now explore a concrete example - the *layout* declaration for the Petri net editor. We present the *patterns* that are used in more detail, and show how they are adjusted, combined and refined to the language specific *layout* desired.

5.1 *ContainmentPattern*

The *ContainmentPattern* assures that components are completely inside a surrounding component. They may not intersect the border line of the surrounding component.

In order to apply the *ContainmentPattern*, the model must contain the following components. Between *Container* and *Element*, there must be a 1-to-many association. *Container* must have the attributes *width*, *height*, *xPos* and *yPos*. *Element* must have the same attributes (Fig. 8).



(a) Requirements (b) Application

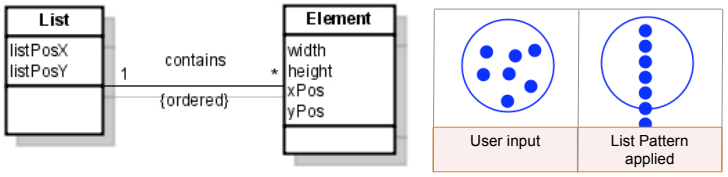
Figure 8: *ContainentPattern*

We apply the *ContainmentPattern* to places as *Container*, and tokens as *Element*. In order to use the pattern we need to add the attributes *width* and *height* to places and tokens. For places we introduce a bidirectional mapping between *height* and *radius* and between *width* and *radius*, as described in Sect. 4.1. For tokens, these are fixed values: 20 for both, an unidirectional mapping (*height* ← 20, *width* ← 20).[5] In Fig. 8 we see an excerpt of a Petri net. On the left side we see the user input, on the right side the result after applying the *ContainmentPattern*.

[5] Note that all tokens are moved inside the square width*height, not into a circle. To change this, we would need to define a specialized *pattern*.

5.2 *ListPattern*

The *ListPattern* is responsible for arranging a set of components as a list. The *ListPattern* requires that the meta model contains a 1-to-many association between *List* and *Element*. Furthermore, the *List* must have the attributes *listPosX* and *listPosY*. These are the coordinates where the list starts. *Element* must have the attributes *width*, *height*, *xPos* and *yPos*. We can see the requirements in Fig. 9.



(a) Requirements (b) Application

Figure 9: *ListPattern*

We apply the *ListPattern* to places as *List* and tokens as *Element*. In order to use the *pattern*, we must add the attributes *listPosX* and *listPosY* to place. For *listPosX* and *listPosY* we define a bidirectional mapping. ($listPosX \leftarrow xPos + width/2$, $xPos \leftarrow listPosX - width/2$ and $listPosY \leftarrow yPos$, $yPos \leftarrow listPosY$). *Transition* already has all attributes required.

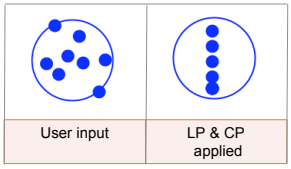


Figure 10: Cont. and *ListPattern*

The *ListPattern* provides several customization options. For example we may choose whether to align elements vertically or horizontally. By default they are aligned vertically, as used for our Petri net editor. In Fig. 9 we see an excerpt of a Petri net. On the left side we see the user input, and on the right side the result after applying the *ListPattern*. In Fig. 10 we can see what happens if we apply both - the *ContainmentPattern* as well as the *ListPattern*.

5.3 *GraphPattern*

As the third and last *pattern* we use the *GraphPattern*, the *pattern* that was already described in the last section. It demands that arrows start and end at the border of transitions and places.

In addition, arrows must have a minimal length. The *GraphPattern* may be applied if the requirements shown in Fig. 6 are met. We apply the *GraphPattern* to places and transitions as *Node* and arrows as *Edge*. The *GraphPattern* also provides several customization options. For example, we may change the minimal length of arrows. This opportunity is used in order to specify our *layout*. Or we may arrange components from left-to-right, instead of top-to-bottom. In Fig. 11 we can see the user input on the left side. On the right side we see the result after applying the *GraphPattern*.

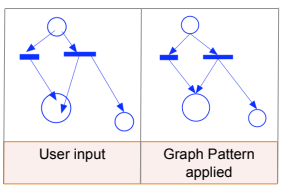


Figure 11: *GraphPattern*

5.4 Complete Layout

To demonstrate the simplicity and flexibility of the *pattern concept*, we include all concepts described in Sect. 4 in the complete layout. We use the *patterns* described above (*pattern usage* and *pattern combination*). We change the minimal length of arrows required (*pattern adjustment*) and we require that transitions must have a minimal size (*pattern refinement*). We override the attribute *minLength* of the *GraphPattern* to change the minimal length of arrows, and we add an additional constraint and its corresponding attribute evaluation rules to assure the minimal size of transitions. The interesting part of the layout specification is the following:

```
gp = new GraphPattern(CArrow,CPlace,CTransition);
lp = new ListPattern(CPlace,CToken));
cp = new ContainmentPattern(CPlace,CToken);

gp.adjust("minLength := 100");

Constraint constr = new Constraint("width > 100",CTransition);
constr.addRule("width := width + 10");
gp.refine(constr);
```

6 Implementation

In this section, we will give an overview of DIAMETA, the environment in which the algorithm was tested and explain how the algorithm was integrated in the framework. We will then examine the layout algorithm and the pattern concept in terms of usability and performance.

6.1 Integration of the Layout Algorithm in DIAMETA

DIAMETA provides an environment for rapidly developing diagram editors based on meta-modeling. Each DIAMETA editor is based on the same editor architecture which is adjusted to the specific diagram language.

6.1.1 Architecture

Fig 12. shows the structure which is common to all DIAMETA editors [Min06a, Min06b]. The editor supports *free-hand editing* by means of the included drawing tool which is part of the editor framework, and can be adjusted by the DIAMETA Designer. With this drawing tool, the user is able to create, arrange and modify the diagram components of the particular diagram language. Editor specific program code, which has been specified by the editor developer and generated by the DIAMETA Designer, is responsible for the visual representation of these language specific components. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (position, size, etc.).

The sequence of processing steps necessary for *free-hand editing* starts with the modeler and ends with the model checker; the modeler first transforms the diagram into an internal model, the graph model. The reducer then creates the diagrams instance graph that is analyzed by the model analyzer. This last processing step identifies the maximal sub diagram which is (syntactically)

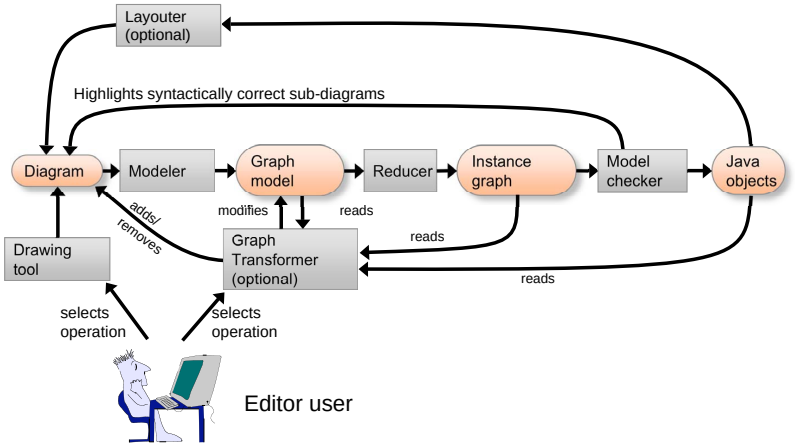


Figure 12: Architecture of a diagram editor based on DIAMETA

correct and provides visual feedback to the user by drawing those diagram components in a certain color; errors are indicated by another color. However, the model analyzer not only checks the diagrams abstract syntax, but also creates the object structure of the diagram's syntactically correct sub diagram.

Then the *layouter* is (optionally) called. It modifies attributes of diagram components and thus the diagram *layout* is based on the (syntactically correct sub diagrams) object structure that was created in the last processing step.

6.1.2 Framework

This section completes the description of DIAMETA and outlines its environment supporting specification and code generation of diagram editors that are tailored to specific diagram languages. The DIAMETA environment shown in Fig. 13 consists of an editor framework, the DIAMETA Designer and the DIAMETA *Layout Generator*.[6]

The framework that is basically a collection of Java classes, provides the generic editor functionality, which is necessary for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of its model, and second, the visual appearance of diagram components, the concrete syntax of the diagram language, the reducer rules and the interaction specification. Besides that, he may provide a *layout specification*, if he wants to define a specific *layout*. We may use the pattern concept in this specification.

DIAMETA can either use the Eclipse Modeling Framework (EMF version) [AKRS06, Min06a] or MOFLON (MOF version) [BBM03, Min06b] for specifying language models and generating their implementations. Our algorithm implementation is based on the EMF version. But with minor changes, the algorithm and the pattern concept may also work with the MOF version instead. A languages class diagram is specified as an EMF model that the editor developer creates

[6] The *Layout Generator* is the implementation of the *generic layout algorithm* presented in this paper.

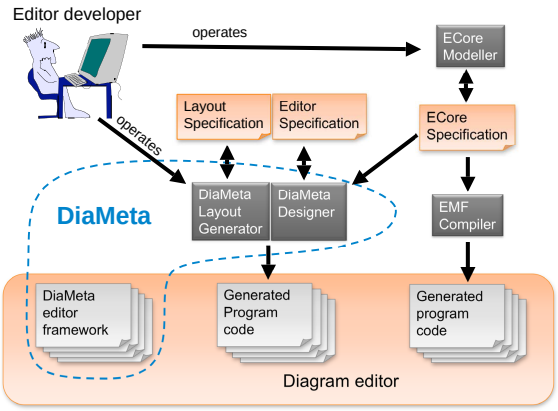


Figure 13: Generating diagram editors with DIAMETA

by using the EMF modeler. The EMF compiler, being part of the EMF plugin for Eclipse, is used to create Java code that represents the model. The EMF compiler creates Java classes (respectively interfaces) for the specified classes. The editor developer uses the DIAMETA Designer for specifying the concrete syntax and the visual appearance of diagram components, e.g. places are drawn as circles, transitions as rectangles, and *edges* as arrows. The DIAMETA Designer generates Java code from this specification. In addition, we can provide a *layout specification*, e.g. we may iapply the *GraphPattern* to arrows, places and transitions. The DIAMETA *Layout Generator* generates Java code from this specification. This Java code, together with the Java code generated by the DIAMETA Designer, the Java code created by the EMF compiler and the editor framework, implement an editor for the specified diagram language.

6.2 Usability and Performance

In the last subsection we described how the *layout algorithm* (and thus the *pattern concept*) was integrated in DIAMETA. We examine the algorithm in terms of usability and performance, as it is done in [DFAB98].

Schmidt demonstrated in [SK03, Sch06] that a small number of *patterns* is sufficient to implement a great variety of visual languages. After investigation of several visual languages, we specified via DiaMeta in the past, we came to the same result. For these languages, the editor developer can use the predefined *patterns*. This simplifies the *layout* specification. For a visual languages' layout that needs more flexibility, these *patterns* may be adjusted or refined. The editor developer may create new ones or use the *generic layout algorithm* in the traditional way. Consequently, the specification of the *layout* becomes rather complex and complicated, but on the other hand we have the whole functionality available. The language used is OCL, a standard language. This has the advantage that only a short training phase is needed, if OCL is known. The week point of the specification is that you have to write a correct specification with no tool support, e.g. you have no error reporting or error correction. Besides that, no constraint or rule simplification is performed.This will be the focus of further research. All in all the *generic layout algorithm* in combination with the *pattern concept* is a powerful tool for specifying a *layout* for

a specific visual language. The editor developer himself may decide how much effort he wants to put onto the *layout specification*.

The weak point of most algorithms solely based on constraint satisfaction is performance [CMP99]. In our algorithm we provide the constraints as well as the solution to these constraints (attribute evaluation rules). This has the consequence that layout computation is no longer time consuming. Thus performance is (up to now) no issue. E.g. for the presented example, layouting a diagram that contains 200 components (50 places, 50 transitions, 50 tokens and 50 arrows) takes less than 0.5 seconds. For further details, please refer to [MM07].

7 Conclusions and Prospects

The diagram editor generator framework DIAMETA makes use of meta-model-based language specifications and supports *free-hand* as well as *structured editing*. The algorithm described in [MM07] is a modular and *generic layout algorithm* that meets the demands of this kind of editors. The fundamental concept of the algorithm is constraint satisfaction combined with attribute evaluation in the sense that constraints are used to activate particular attribute evaluation rules. This combination gives the *layouter* the flexibility it needs to support *free-hand* as well as *structured editing*. By means of the example we saw that it is possible to define a *layout algorithm* for diagrams that supports the user during user interaction (incrementally), and meanwhile grants the user plenty of freedom. Furthermore, a *layouted* diagram is displayed at any time.

We realized that writing such a specification is rather complicated. Therefore we encapsulated basic functionality, and give the user the opportunity to use (and reuse) these *patterns*. *Patterns* introduce another level of abstraction on top of the specification, as *design patterns* do in object oriented software design. A *pattern* is basically a set of constraints and attribute evaluation rules that is tailored to a specific problem, e.g. to the problem of arranging arrows in a graph-based visual language. *Patterns* may be used if some requirements are satisfied. It may be adjusted to a visual language as required. The editor developer has the opportunity to combine different *patterns* or refine a *pattern*. Of course he may also add additional functionality or create new *patterns* from scratch. Using the *pattern* concept made writing a specification easier, without loosing the flexibility of the original, sufficiently efficient *generic layout algorithm*.

Up to now creating a specification or defining a *pattern* has to be done by hand. The next step will be to introduce GUI support for creating *patterns* and also for creating a whole specification.

Extensive case studies are planned, and may lead to an enhanced *pattern concept*. The extension will include a priority concept for constraints and will offer a possibility to integrate existing *layouter* and constraint solver. We will apply the concept to graph-based as well as other visual languages. We will examine the applicability to diagrams of different sizes. We will define different views for the same visual language. Till now we focused on an incremental layout, in future case studies we will also examine a complete automatic layout. We will investigate *free-hand* as well as *structured editing*.

Identifying *patterns* in the model automatically is also imaginable. Then the program could suggest *patterns* applicable, and the only thing the user has to do is selecting the *pattern* desired. The program could even change the model such that a specific *pattern* is applicable. This idea will be the focus of further research.

Bibliography

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink and Warmer (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*. Lecture Notes in Computer Science (LNCS) 4066, pp. 361–375. Springer Verlag, Heidelberg, 2006.

[BBM03] F. Budinsky, S. A. Brodsky, E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[CMP99] S. S. Chok, K. Marriott, T. Paton. Constraint-Based Diagram Beautification. In *VL '99: Proceedings of the IEEE Symposium on Visual Languages*. P. 12. IEEE Computer Society, Washington, DC, USA, 1999.

[DFAB98] A. Dix, J. Finley, G. Abowd, R. Beale. *Human-computer interaction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[Min04] M. Minas. VisualDiaGen – A Tool for Visually Specifying and Generating Visual Editors. In *Applications of Graph Transformation with Industrial Relevance, Proc. 2nd Intl. Workshop AGTIVE'03, Charlottesville, USA, 2003, Revised and Invited Papers*. Lecture Notes in Computer Science 3062. Springer-Verlag, 2004.

[Min06a] M. Minas. Generating Meta-Model-Based Freehand Editors. Appears in Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), September 21-22, 2006, Satellite event of the 3rd International Conference on Graph Transformation, 2006.

[Min06b] M. Minas. Generating Visual Editors Based on Fujaba/MOFLON and DiaMeta. In Giese and Westfechtel (eds.), *Proc. Fujaba Days 2006, Bayreuth, Germany, September 28-30, 2006*. Pp. 35–42. 2006. Technical Report tr-ri-06-275 Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Institut für Informatik.

[MM07] S. Maier, M. Minas. A Generic Layout Algorithm for Meta-model based Editors. In *Applications of Graph Transformation with Industrial Relevance, Proc. 3rd Intl. Workshop AGTIVE'07, Kassel, Germany*. 2007.

[OMG06] OMG. Object Constraint Language (OCL) Specification, Version 2.0. 2006.

[Sch06] C. Schmidt. *Generierung von Struktureditioren für anspruchsvolle visuelle Sprachen*. PhD thesis, Universität Paderborn, D-33098 Paderborn, Germany, 2006.

[SK03] C. Schmidt, U. Kastens. Implementation of visual languages using pattern-based specifications. *Softw. Pract. Exper.* 33(15):1471–1505, 2003.