Electronic Communications of the EASST
Volume 13 (2008)

Proceedings of the Second International Workshop on Layout of (Software) Engineering Diagrams (LED 2008)

Layout Specification on the Concrete and Abstract Syntax Level of a Diagram Language

Sonja Maier, Steffen Mazanek and Mark Minas

15 pages

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

Layout Specification on the Concrete and Abstract Syntax Level of a Diagram Language

Sonja Maier[1], Steffen Mazanek[2] and Mark Minas[3]

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

Abstract: A visual language consists of several visual component types, e.g. states or transitions in DFAs. Nowadays, the language itself is usually specified via a meta model. To make a diagram look nice, a layouter is required. This layouter may either operate on the concrete syntax level, i.e., on the visual components, or on the abstract syntax level, i.e., on the model instance. In this paper we present an approach that is capable of specifying a flexible layout on both, the concrete as well as the abstract syntax level of a diagram. The approach uses pattern-based transformations. Besides structured editing, it also supports free-hand editing, a challenging task for the layouter. We introduce how such a specification can be created and examine the advantages and shortcomings of each of either operating on the concrete syntax level or on the abstract syntax level.

Keywords: layout algorithm, concrete syntax, abstract syntax, graph transformation, model transformation

1 Introduction

Each visual editor implements a certain visual language. 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 as of today have a meta model as (abstract) syntax specification. Meta models are essentially class diagrams of the data structures visualized as diagrams.

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, thus giving the user more freedom and enabling sketching [BM08]. The editor has to check whether the drawing is correct and what its meaning is. For free-hand editing a more sophisticated layouter is required: temporarily "incorrect" diagrams need to be layouted. Furthermore, modifying the appearance of a drawing might result in changes of its meaning.

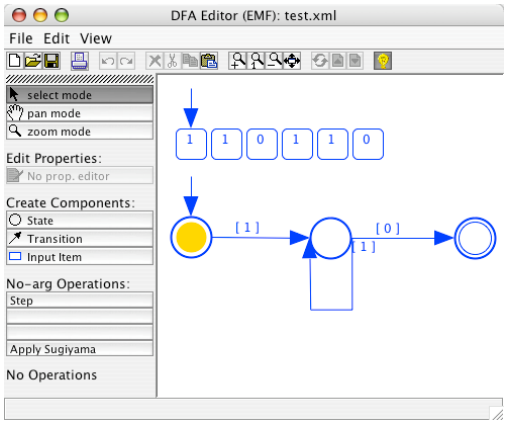


Figure 1: Editor for DFA's

When talking about layout, we need to distinguish two terms: *layout*, the general term, and *layout refinement* (sometimes called beautification [CMP99]). Layout refinement starts with an initial layout and performs minor changes to improve it while still preserving the "feel" (or "mental map" [PHG07]) of the original layout (also called least astonishment principle). Especially user interaction is considered in this context. A *layouter* may also position components of the diagram from scratch without an initial layout.

In [MM07a] we introduced a dynamic layout algorithm usable for meta-model-based diagram editors, which combines the concepts *constraints* and *attribute evaluation*. This approach provides us with all we need for layout refinement. However, we have recognized that the constraints and attribute evaluation rules that need to be specified tend to be long and complicated, especially when using a "real world" layouting strategy. We tried to improve on this aspect and combined graph transformation with this dynamic layout algorithm [MM08].

When performing experiments with this approach, using pattern-based graph transformations for layout specification proved to be a good idea. Also the level on which a layouter is based on came into question. Generally, we distinguish between the abstract and the concrete syntax level.

Often, abstract and concrete syntax show a significant redundancy. But there are also some differences: abstract syntax may contain elements that cannot be expressed on the concrete syntax level of a diagram language, whereas elements in the abstract syntax may be expressed in different ways on the concrete syntax level [KRV07].

The additional information each of these two levels offers may be considered for layout computation. Hence, we found that besides a specification primarily based on the abstract syntax of a diagram language (as done in [MM08]), also a specification that is primarily based on the concrete syntax is reasonable. Many tools operate on one of these levels - some on the abstract and some on the concrete syntax level - without considering the other. In this paper, we examine both and describe the advantages and shortcomings of each of these variants, especially in the context of free-hand editing.

We introduce how such a specification can be created and examine the flexibility this approach offers to the developer of an editor. The degrees of freedom that may be provided to the user are

depicted, meaning the possibilities of adjusting the layout to his individual preferences.

We demonstrate our approach by specifying a rather simple layout for deterministic finite automata (DFAs).

We integrated and tested our approach in DIAMETA [Min06a]. DIAMETA 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. Fig. 1 shows a DIAMETA editor for DFAs.

Sect. 2 introduces DFAs, the visual language used as a running example. Sect. 3 depicts the decisions that need to be made prior to layouter creation. Sect. 4 explains the proposed algorithm, and gives a detailed example. Sect. 5 compares the different variants and examines the flexibility of the approach. Sect. 6 provides an overview of DIAMETA, the environment in which the algorithm has been tested and highlights some details about the implementation of the algorithm. Sect. 7 contains related work and Sect. 8 concludes the paper.

2 Running Example

As a running example, we choose deterministic finite automata. Fig. 1 shows a sample DFA. It consists of three states and three transitions. The active state is highlighted. The start state is indicated by an arrow, end states by a second circle. Additionally, the editor shows the input string `110110`. The active symbol of the input string (called input item in the following) is visualized by an arrow.

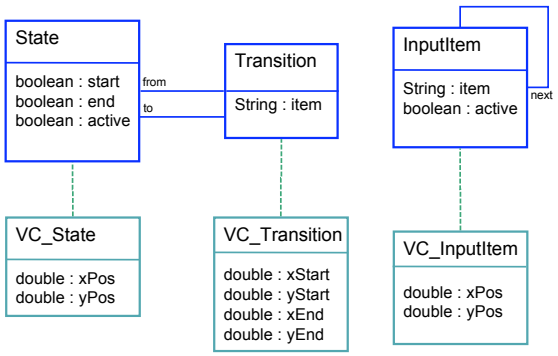


Figure 2: Meta Model

Fig. 2 shows the (rather trivial) meta model of this visual language: the meta model contains the classes that represent aspects of the abstract syntax of the visual language, *State*, *Transition* and *InputItem*. The class *State* has three attributes. The attribute *start* (*end*) indicates whether a state is a start state (end state) or not. The attribute *active* indicates whether a state is the active state or not. The class *Transition* has an attribute *item*. This attribute holds the input item that is necessary to fire the transition. Between the classes *State* and *Transition*, there are two associations *from* and *to*.

Besides the classes that represent aspects of the abstract syntax, the classes VC_State,

VC_Transition and VC_InputItem hold the significant attributes of the concrete syntax. The position of states and input items is determined by the attributes *xPos* and *yPos*, the location of the upper left corner of a state. The start and end points of an arrow are given by the attribute *xStart*, *yStart*, *xEnd* and *yEnd*.

3 Layout Decisions

Before layouter development, some questions need to be answered: which visualization aspects does the layouter cover? How is the layouter triggered? What changes should be performed? The first two questions are explored in this section, the third question is investigated in the next section.

3.1 Coverage?

Often, it is not clear which visualization aspects a layouter should cover. E.g., in our example, a start state is modeled as exactly one component. That means, the part of the system that takes care of the appearance of a component is responsible for its visualization. Alternatively, we could have modeled a start state with two components: the arrow and the circle. Then the concrete class of a circle would have contained the attributes *xPos* and *yPos* to specify its location. The concrete class of an arrow would have contained the attributes *xStart*, *yStart*, *xEnd* and *yEnd* to specify the position of the start point and the end point of an arrow. In this case, it would have been the layouter's responsibility to adjust those attributes. These decisions need to be made by the developer of the editor.

3.2 Triggered by?

The second aspect to be displayed is how and when a layouter is triggered. One variant is that a layouter is applied automatically, each time the user modifies a diagram, i.e. each time the diagram is repainted. This might happen, for example, if the user adds or removes a component. This might either be the same layouter, regardless of which component was changed, or different layouters, depending on the type of the component changed. Calling different layouters means that different options are provided to the user. For instance, in our example, if the user moves a state, the attached transitions follow the state. If the user moves a transition, only the transition is moved, nothing else.

It is also possible to call a layouter manually, e.g., by pressing a button or choosing a menu item. This is advantageous, since we can provide more than one layouter and the user can decide when he wants to relayout the diagram.

The first option is used for layout refinement, and the second option for bigger changes.

Our approach supports all these variants. In our DFA editor (Fig. 1), three different layouters are called automatically, depending on the kind of component changed. Additionally, we included a layouter that may be applied manually, by clicking the button "Apply Sugiyama".

4 Layout Algorithm

A layouter may operate on the concrete or the abstract syntax level of a diagram language. In DIAMETA operating on the concrete syntax means operating on an attributed hypergraph, the graph model of a diagram. Operating on the abstract syntax of a diagram in DIAMETA means operating on an object model, i.e. on an *EMF* model [Min06a].

On both levels, one is actually manipulating model and view level attributes. In the hypergraph approach, the focus is on the view level. Here, significant model level attributes (e.g. *active* in the running example, as we will see) are "embedded" into the rules. On the abstract syntax level, significant view level attributes are "embedded" via visual components (e.g. VC_State).

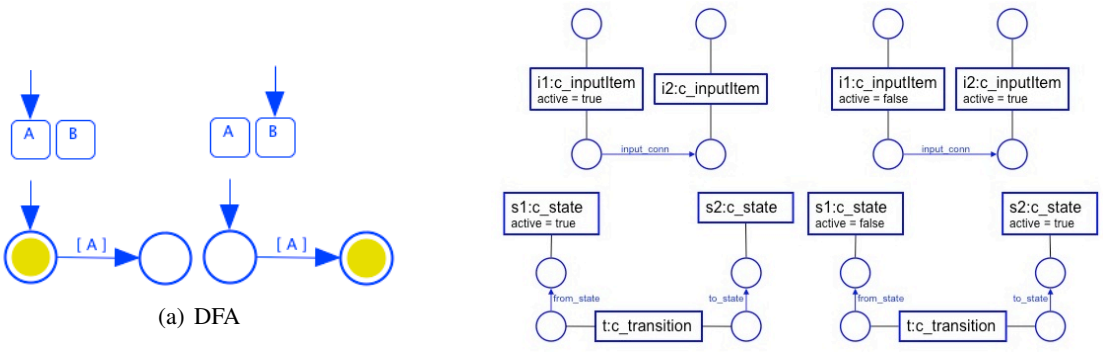
In hypergraph models (HGM) [Roz97], diagram components are represented by labeled hyperedges, i.e. edges that are allowed to be visited by an arbitrary number of nodes. Nodes represent attachment points. Furthermore, we have spatial relationship edges, representing relationships between different components. E.g., Fig. 3 shows four hypergraphs. Here, edges are represented by rectangles and outgoing lines, and nodes by circles. Spatial relationship edges are represented by arrows between circles.

Our layout algorithm identifies a match and then performs some actions. Both can be performed on the graph model as well as on the *EMF* model. We call this strategy pattern-based transformation (or transformation). Technical details are explained in Sect. 6.

In the following, we compare these variants and demonstrate the new experiences we gained from experiments. Although we choose a rather simple layout for the comparison, more sophisticated layouters are possible, of course.

4.1 Operating on Concrete Syntax Level

One alternative is operating on the concrete syntax of a diagram language. On this level, the pattern-based graph transformation is a graph transformation. This transformation primarily changes attribute values. First, we describe a simulation operation, the context in which a transformation normally is used. Afterwards we show a transformation, as it is used for layout.



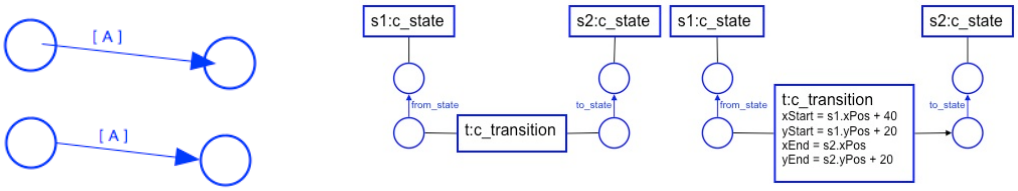
(a) DFA

(b) HGM

Figure 3: "Step"

Simulation Operation A sample simulation operation is the transformation "Step". In Fig. 3 (a) we can see a screenshot of a DFA. On the left-hand side is the DFA before applying the transformation. On the right-hand side is the diagram after executing the transformation. In Fig. 3 (b), the hypergraph representing the diagram is shown. In particular, the transformation changes certain attribute values. It sets the attribute *active* of s1 to false and the attribute *active* of s2 to true.

Layout Transformation After specifying a simulation operation, we now specify two layout transformations. The first transformation is responsible for layouting transitions, the second for layouting input items. As we will see, these transformations are quite similar to the graph transformation shown in the last paragraph.



(a) Layout

(b) HGM

Figure 4: Transitions

In Fig. 4 (a) a transition before and after applying the layout transformation for layouting transitions is shown. Fig. 4 (b) visualizes the hypergraph before and after applying the transformation. The attributes *xStart*, *yStart*, *xEnd* and *yEnd* of the object *t* are changed if the following constraint evaluates to *false*. This means that the transformation is applied if the arrow does not start and end at the "correct" position:[1]

```
xStart = s1.xPos + 40 & yStart = s1.yPos + 20
& xEnd = s2.xPos & yEnd = s2.yPos + 20
```

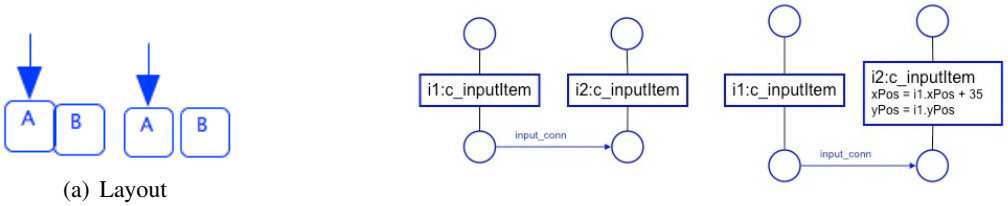
The layout transformation consists of two parts: a match and some actions. The match identifies a part of the diagram, to which the actions can be applied. The match is a pattern consisting of two states, connected by a transition. The actions performed update the attributes *xStart*, *yStart*, *xEnd* and *yEnd*.

In Fig. 5 (a) two input items before and after applying the layout transformation for layouting input items can be seen. Fig. 5 (b) shows the hypergraph before and after applying this transformation. The attributes *xPos* and *yPos* of the object *i2* are changed if the following constraint evaluates to *false*, meaning the transformation is applied if the item is not at the correct position:

```
xPos = i1.xPos + 35 & yPos = i1.yPos
```

The match is a pattern consisting of two adjacent input items. The action changes the attributes *xPos* and *yPos*.

[1] $20 = radius$; $40 = 2 * radius$; $35 = width + 5$



(a) Layout

(b) HGM

Figure 5: Input items

Both transformations are applied to all matches. For the transformation being responsible for layouting transitions, this is done in an arbitrary order. For the transformation that is responsible for layouting input items, this is done from left to right. To specify the order of application, a special (DIAMETA-specific) language is used.

Both transformations are triggered automatically: the first transformation, if a state is changed, and the second transformation if an item is changed, respectively.

In the examples, only attribute values are changed, not the structure of the hypergraph itself. Most of the time, this is sufficient. In some cases, also structural changes are needed. E.g., for some layout algorithms, the layouter might choose to temporarily add additional components, and then remove them later again. This mechanism is, e.g., used in Sugiyama's Algorithm for layouting graphs (Fig. 6). The circle with the dashed line is temporarily added. We implemented this algorithm on top of our framework [MM08].

In the example (Fig. 6), bendpoints are added - an element that is only available on the concrete syntax level. There is no representation of bendpoints in the abstract syntax. Hence bendpoints cannot be modified by the layouter that operates on the abstract syntax level.

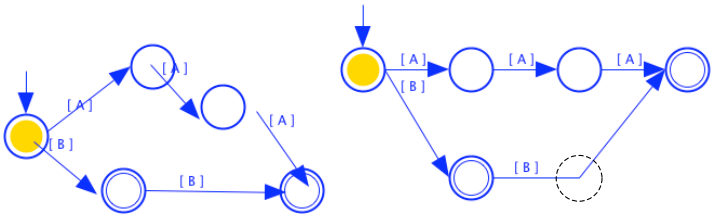


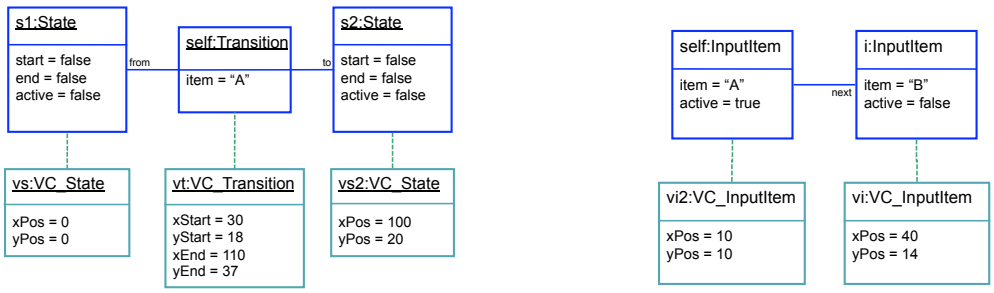
Figure 6: Structural Changes

The examples shown here are very simple. A more flexible layout is possible, e.g., one could demand that transitions are longer than some fixed value, or that transitions start and end at the border of a state. The full power of the actions, e.g. using incremental actions, was shown in [MM07a].

4.2 Operating on Abstract Syntax Level

Alternatively, one could operate primarily on the abstract syntax level of a diagram language. On this level, the pattern-based transformation is a model transformation.

To work directly on the *EMF* model, classes for the visual component types were introduced. These are the classes VC_State, VC_Transition and VC_InputItem (see Fig. 2). This means that we do not strictly operate on the abstract syntax, as we extended the abstract syntax with classes that represent aspects of the concrete syntax, the aspects needed for layout specification.



(a) Transition (b) Input item

Figure 7: Object Diagram

The transformation responsible for layouting transitions operates on an object diagram, e.g. as shown in Fig. 7 (a). The transformation is again specified via a match and an action. The complete specification is the following:[2]

```
[self.from changed]
  self.vcomp.xStart := self.from.vcomp.xPos + 40
[self.from changed]
  self.vcomp.yStart := self.from.vcomp.yPos + 20
[self.to changed]
  self.vcomp.xEnd := self.to.vcomp.xPos
[self.to changed]
  self.vcomp.yEnd := self.to.vcomp.yPos + 20
```

The match is implicitly given. E.g. the first two rules may only be applied if we have two states connected by the link *from*. An action consists of a constraint (1) and an attribute evaluation rule (2). If (1) is not satisfied, then (2) is executed.

```
(1) false
(2) self.vcomp.xStart := self.from.vcomp.xPos + 40
```

The transformation responsible for layouting input items operates on an object model, e.g. as shown in Fig. 7 (b). The corresponding layout specification is the following:

```
[self.next changed]
  self.vcomp.xPos := self.next.vcomp.xPos - 35
[self.next changed]
  self.vcomp.yPos := self.next.vcomp.yPos
```

All transformations are triggered automatically when the object denoted in the squared brackets was changed by the user or the layouter.

[2] Here we may also provide an arbitrary OCL expression.

5 Evaluation

In this section we evaluate the approach. We will demonstrate the flexibility this approach offers and compare the two levels of application.

5.1 Flexibility

First of all, we may decide whether we operate on the concrete syntax level or on the abstract syntax level of a diagram language.

Second, a layouter offering a flexible behavior can be specified. E.g., one can specify a transformation that assures that a transition is longer than a minimal length. This way, also mental map preservation can be achieved. E.g., one could introduce a constraint that assures that a component is never moved more than, say for example, 10 pixel.

Third, we provide different ways to trigger the layouter: either automatically, eventually depending on the component changed, or applied by hand, i.e. explicitly by the editor user.

Every user has its own "feeling" of what a good layouter is. Similarly every developer prefers another way to specify a layouter. Our approach offers an environment for performing experiments and will give us a better understanding of what a "good" layouter is for most users and what the most pleasant environment is for developers to create such a layouter.

5.2 Comparison

For many visual languages, a similarity between abstract and concrete syntax can be observed. But there are also visual languages where this is not the case. One example are tree diagrams, as presented in [Min06b]. In our example, the concrete syntax contains more information. But it is also imaginable that the abstract syntax contains more information. Right now, this is not possible with DIAMETA. For this reason, we currently integrate Triple Graph Grammars in DIAMETA.

On both levels, free-hand editing provides an additional challenge: careless changes of the diagram might result in changes of the meaning of the diagram. We may overcome this problem by prohibiting any structural changes in the abstract syntax, e.g. by using the mechanism proposed in [RS96]. But, as motivated in Sect. 4, we want to allow some structural changes, and hence there is a need to extend this approach.

The two levels may be compared considering the different information that is contained as well as their syntactic differences. Currently, both abstractions contain more information than minimally needed. E.g., if layout would be solely specified on the hypergraph, one could completely remove concrete syntax elements from the EMF model.

In [Baa06], a strict definition of "correct" concrete syntax is given. It claims that each diagram only corresponds to one model. Our approach was kept more generally, as we only recommend, but do not demand this.

Concrete Syntax Specifying layout transformations on the concrete syntax is more intuitive than specifying layout transformations on the abstract syntax. The reason is that a layouter changes the visualization, not the model.

But it is also possible to "destroy" the diagram by a transformation on the concrete syntax level, because arbitrary changes in the hypergraph may be performed. E.g. you are allowed to add edges that cannot be mapped to the object model. Consequently, this kind of transformation needs to be created more carefully, assuming that the developer brings along the required knowledge. Even then it is hard to avoid all problems.

Besides, you may access visualization information that is not available on the abstract syntax level. The number of bendpoints of an arrow, or the angle between arrow and circle are two examples for this kind of information. Components contained in an "incorrect" part of a diagram are also available on the concrete syntax level and may be considered in the layout specification. They are not mapped to the object model. But it is possible that some information is not represented in the concrete syntax, only in the abstract syntax, even though this is not the case in the example presented.

Abstract Syntax For the formulation of constraints and attribute evaluation rules, the standard OCL syntax is used. This has the advantage that a developer already familiar with OCL is not required to learn a new language. Additionally, the developer is prevented from "destroying" the diagram, as only meaningful changes are possible on the object model. But at some points it is not as intuitive as operating on the concrete syntax and the developer is more restricted. E.g., between input items we do not have a bi-directional link, only the link *next*. Consequently, we decided to layout input items from right to left, not from left to right, as this was accomplished more easily.

A layout strictly based on the abstract syntax means that visual components are not available, and hence cannot take into account any of this information, e.g. the position of a component. In our approach, we extended the abstract syntax with classes that represent the aspects of the concrete syntax that were needed for layout specification. One advantage of this approach is that we explicitly name the information that we use for layout computation, one disadvantage is that we do generate some specification overhead. Another advantage is that we can utilize the additional information the abstract syntax provides.

6 Implementation

In this section, we are going to introduce DIAMETA, the environment the algorithm is implemented in. It is needed to understand the context in which the layout algorithm is used. In particular it generates an overview of the implementation of pattern-based transformations. They can either operate on the abstract or on the concrete syntax. In the first case, they operate on an attributed hypergraph. In the second case, they operate on an *EMF* model.

The editor generator framework 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. DIAMETA's tool support for specification and code generation, primarily the DIAMETA Designer is described in this section. The description of the architecture is postponed to the next section.

6.1 DIAMETA Framework

The DIAMETA environment consists of an editor framework and the DIAMETA Designer. The framework is basically a collection of Java classes and provides the dynamic 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 enter 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, the interaction specification and the layout specification.

A language's class diagram is specified as an *EMF* model. The *EMF* compiler is used to create Java code that represents this model. The editor developer uses the DIAMETA Designer for specifying the concrete syntax and the visual appearance of diagram components, e.g., states are drawn as circles. The DIAMETA Designer generates Java code from this specification. 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 Layout Algorithm

Our layout algorithm uses pattern-based transformations, either operating on the concrete or on the abstract syntax level. A transformation consists of a match and an action, which itself is made up of a constraint and an attribute evaluation rule. On the concrete syntax level, transformations operate on the attributed hypergraph. On the abstract syntax level, transformations operate on the *EMF* model. All transformations - graph transformations as well as model transformations - are controlled by the *layouter control*.

Actions Actions are based on the algorithm presented in [MM07a]. This algorithm is responsible for satisfying violated constraints. We have to specify a set of given declarative constraints, assuring the characteristics of the layout. If all constraints are satisfied, the algorithm terminates. If one or more constraints are violated, some attributes are changed via the attribute evaluation rules provided, and afterwards the constraints are checked again.

6.3 DIAMETA Architecture

Fig. 8 shows the structure which is common to all DIAMETA editors - editors generated and based on DIAMETA. The editor supports free-hand editing by means of the included drawing tool which is part of the editor framework, but which has been 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, specified by the editor developer and generated by the DIAMETA Designer, is responsible for the visual representation of the language specific components. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (e.g., position, size).

The sequence of processing steps, necessary for free-hand editing, starts with the modeler and ends with the model analyzer: the modeler first transforms the diagram into an internal model, the graph model. The reducer then creates the diagram's instance graph that is analyzed by the model

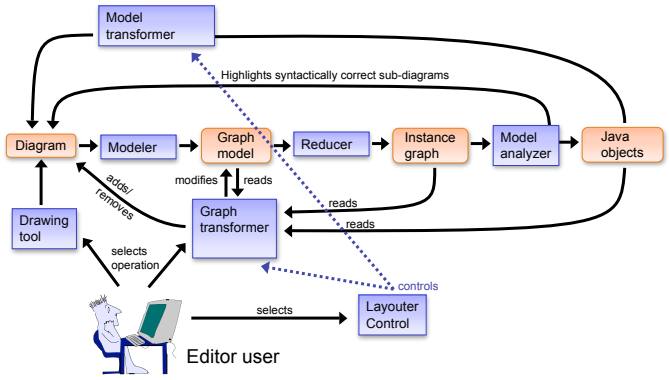


Figure 8: Architecture of DIAMETA

analyzer. This last processing step identifies the maximal subdiagram which is (syntactically) correct and provides visual feedback to the user by drawing those diagram components in a certain color. The model analyzer not only checks the diagram's abstract syntax, but also creates the object structure of the diagram's correct subdiagram.

Structured editing / simulation operations modify the graph model by the means of the graph transformer and add or remove components to respectively from the diagram. The visual representation of the diagram and its layout is then computed by the layouter.

The graph transformer and the model transformer are both optional, but essential for the layout algorithm proposed.

7 Related Work

Layout Specification Many comparable tools, like AToM3 [LV02] or Tiger [EEHT05], offer the possibility of using a standard layout algorithm, such as FlowLayout. Besides this, some tools, like DiaGen [Min03], offer the possibility of using constraints for layout specification. Most tools additionally allow to write the layouter by hand, as only a small subset of layouters can be realized by the mechanisms provided. With the approach presented we try to fill this gap.

Transformations Many tools support graph or model transformation, but only rarely use them in the context of layout specification. Guerra and de Lara describe Event Driven Grammars [GL07]. Rules in these grammars may be triggered by user actions, and are combined with triple graph transformation systems. Rules may be defined especially for layout. If a rule is applicable, it is executed and attribute values are updated. In this approach, attributes are updated through graph transformations. In our approach, we either change the graph model or the object model by pattern-based transformation. Attributes are updated by actions giving us more freedom, especially when considering dynamic layout.

Visual Specification As of now we only allow to specify layout textually. For the future, we plan to specify the layouter visually. Most promising for operating on the abstract syntax are Fujaba's Story Diagrams [FNTZ98] and VisualOCL [EW05]. Fujaba's Story Diagrams offer the possibility of specifying transformations on the object diagram visually. VisualOCL is a visual language to specify OCL constraints. Most promising for operating on the concrete syntax is the visual specification of transformations as it is done, for instance, in Tiger or AToM3.

Visualization of DFAs In [JMM04], DFAs are used as a running example to present the toolkit CIDER. The toolkit provides a simple layout using transformation rules which can constrain the attributes of new symbols (used in the DFA example to handle DFA execution). Furthermore, it provides application specific layout such as graph layout (used in the DFA example for laying out the DFA).

Dynamic Layout Some work had been performed that takes user interaction into account, and to preserve the "mental map". Most of the algorithms are hand coded. With our approach, this is directly included in the editor specification. In order to create a new layout algorithm, graph transformations can be used instead of plain Java code. This has the consequence that the creation and adaption of layouting strategies is easier, and hence experiments in this context are simplified.

Ware et al. state in [WPCM02] that (graph) aesthetics are taken as axiomatic, and have not been empirically tested. They argue that human pattern perception can tell us much that is relevant to the study of aesthetics. With our approach we created a platform to easily perform this kind of studies, not only for graphs but for all kinds of visual languages. They take into account aspects of the concrete syntax as well as of the abstract syntax of the diagram language.

Purchase et al. state in [PHG07] that dynamic graph layout algorithms have only recently been developed. They anticipated that maintaining the "mental map" between time slices assists with the comprehension of the evolving graph. In DIAMETA, not only automatic time-slices, but also time-slices triggered by user interaction are seen. Besides this, free-hand editing provides an initial layout that needs to be considered. With our approach, many degrees of freedom are available that may be considered when creating a new layout algorithm.

8 Conclusions and Future Work

In this paper we presented a layout approach that can either operate on the concrete or on the abstract syntax level of a diagram language. In the case of operating on the concrete syntax, this means graph transformation and on the abstract syntax this means model transformation. We used DFAs as an example and integrated and tested our approach in DIAMETA. We examined the assets and drawbacks of both levels of application. On the concrete syntax level, the layout specification was more natural, whereas on the abstract syntax level, the specification was less error prone. An intelligent combination of both approaches will combine the advantages of both variants in the future. On both levels there is additional information available which can be used for layout computation. As we saw for both variants, just a few simple rules lead to an acceptable

layout. In case the editor developer or the editor user demands a more sophisticated layout, this is also possible.

Future work has to investigate how layouters interfere with user interactions. E.g., a layouter that moves components away complicates the process of diagram creation. During user interaction, a dynamic layouting strategy that that follows the least astonishment principle is more adequate. To identify the "best" layouting strategy, we will need to perform empirical studies. With the algorithm presented, a testing environment was created to conduct these studies easier.

Currently, the specification of layout is based on an abstraction of the visualization, either on the abstract syntax level or on the concrete syntax level. A graphical specification based on the visualization itself is developed at the moment.

We have developed a flexible approach that offers us a platform for establishing experiments of how developers and editor users want to interact with DIAMETA, a tool that allows free-hand editing.

Bibliography

[Baa06] T. Baar. *Correctly Defined Concrete Syntax for Visual Modeling Languages*. 2006.

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

[BM08] F. Brieler, M. Minas. Recognition and Processing of Hand Drawn Diagrams Using Syntactic and Semantic Analysis. In *Advanced Visual Interfaces (AVI), Naples, Italy,*. 2008.

[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.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of Visual Editors as Eclipse Plug-Ins. In *ASE '05: Proceedings of the 20th IEEE/ACM Intl. Conference on Automated software engineering*. New York, NY, USA, 2005.

[EW05] K. Ehrig, J. Winkelmann. Model Transformation from VisualOCL to OCL using Graph Transformation. In *Proc. Intl. Workshop on Graph and Model Transformation (GraMoT'05)*. ENTCS. Estonia, Tallinn, 2005.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In *Proc. of the* 6^{th} *Intl. Workshop on Theory and Application of Graph Transformation*. 1998.

[GL07] E. Guerra, J. de Lara. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and Systems Modeling*, 2007.

[JMM04] A. R. Jansen, K. Marriott, B. Meyer. Cider: A Component-Based Toolkit for Creating Smart Diagram Environments. In *Diagrams*. Pp. 415–419. 2004.

[KRV07] H. Krahn, B. Rumpe, S. Voeslkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In Engels et al. (eds.), *MoDELS*. Lecture Notes in Computer Science 4735, pp. 286–300. Springer, 2007.

[LV02] J. de Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *FASE '02: Proceedings of the 5th Intl. Conference on Fundamental Approaches to Software Engineering*. London, UK, 2002.

[Min03] 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.

[Min06a] M. Minas. Generating Meta-Model-Based Freehand Editors. Electronic Communications of the EASST, Proc. of 3rd Intl. Workshop on Graph Based Tools, Natal, Brazil, 2006.

[Min06b] M. Minas. Syntax analysis for diagram editors: a constraint satisfaction problem. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*. Pp. 167–170. ACM, New York, NY, USA, 2006.

[MM07a] 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.

[MM07b] S. Maier, M. Minas. A Pattern-Based Layout Algorithm for Diagram Editors. In *Electronic Communications of the EASST, Proc. Workshop LED'07, Coeur d'Alene, Idaho, USA*. 2007.

[MM08] S. Maier, M. Minas. A Static Layout Algorithm for DiaMeta. In *Graph Transformation and Visual Modeling Techniques, Proc. 7th Intl. Workshop GT-VMT'08, Budapest, Hungary*. 2008.

[PHG07] H. C. Purchase, E. Hoggan, C. Görg. How Important is the "Mental Map"? – an Empirical Investigation of a Dynamic Graph Layout Algorithm. In Kaufmann and Wagner (eds.), *Graph Drawing, Germany*. 2007.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RS96] J. Rekers, A. Schürr. A graph based framework for the implementation of visual environments. In *VL*. 1996.

[WPCM02] C. Ware, H. Purchase, L. Colpoys, M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 2002.