A Static Layout Algorithm for DiaMeta

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*. In this paper we present a layouting approach that is especially well suited for a *static layout*. It is based on the layout algorithm presented in [MM07a] that uses the two concepts constraint satisfaction and attribute evaluation. This algorithm is combined with graph transformations and the result is a natural way of describing the layout of visual languages. As an example we use a simplified version of *Sugiyama's algorithm*, applied to *statechart* diagrams.

Keywords: Layout Algorithm, Static Layout, Dynamic Layout, Constraints, Attribute Evaluation, Graph 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 model as (abstract) syntax specification. Models are essentially class diagrams of the data structures that are 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. The editor has to check whether the drawing is correct and what its meaning is.

One weak point of such editors is, as always, layout. When talking about layout, we need to distinguish two terms: *Layout*, the general term, and *layout refinement*. *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. Especially user interaction is considered in this context. *Layout* may also position components of the diagram from scratch without an initial layout. For structured editing, *layout* is required, as newly created components need to be positioned from scratch. For free-hand editing, either *layout* or *layout refinement* may be applied. We also need to distinguish the two terms *static layout* and *dynamic layout*. When applying a static layout algorithm to a diagram, it always returns the same visual representation of the diagram. When applying a dynamic layout algorithm, the result is influenced by the ”layout” of the initial diagram and by the user input.

In [MM07a] we presented a dynamic layout algorithm usable for model-based visual languages. It meets the demands of structured as well as free-hand editing. The algorithm combines the concepts *constraints* and *attribute evaluation* to an algorithm that is fast, flexible and behaving the desired way. This approach provides us with all we need for layout refinement. But we have recognized that *constraints* and *attribute evaluation rules* for such a specification quickly get long and complicated, especially when using a ”real world” layouting strategy.

In this paper, we extend our approach, improving this aspect. We combine graph transformation and the dynamic layout algorithm presented in [MM07a]. Graph transformations are applied, with the goal that the complexity of the *constraints* and *attribute evaluation rules* used to specify the dynamic layout algorithm is reduced. The approach also has the benefit that the layout algorithm is separated into phases, each of them treated independently. The reduction of complexity as well as the separation into phases simplifies the creation process of a new layouting strategy. The modularity of the algorithm additionally offers us perfect conditions for setting up a ”testing environment” for layout strategies.

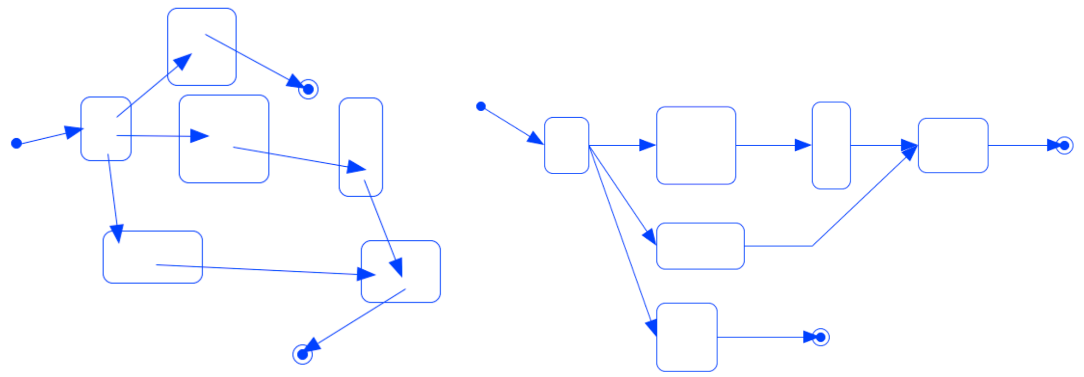

Figure 1: Sugiyama’s Algorithm applied to Statecharts

We demonstrate our approach by specifying Sugiyama’s algorithm (a simplified version), a standard layouting strategy for graphs.

We have integrated and tested our approach in DiaMeta [Min06]. 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. Figure 1 shows a statechart diagram before and after applying the Sugiyama’s algorithm. The diagram was created via a DiaMeta editor. The layout algorithm was implemented using the approach presented in this paper.

Section 2 introduces the model of statecharts, the visual language that is used as a running example. Section 3 gives an overview of DiaMeta, the environment in which the algorithm has been tested. Section 4 explains the proposed algorithm, and Section 5 gives a detailed example. Section 6 contains related work, and Section 7 concludes the paper.

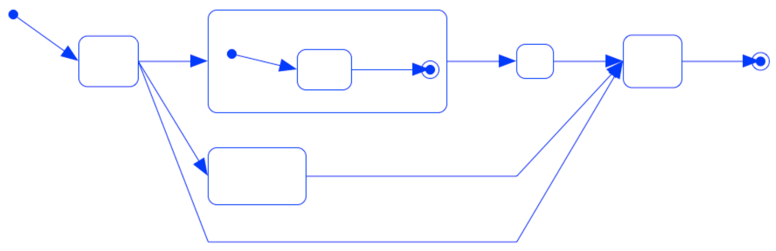

Figure 2: Statechart Diagram

2 Running Example

As a running example we use statecharts. In Figure 2 we see a layouted statechart diagram.

Figure 3 shows the meta model of simplified statecharts presented in this paper. We have ”states” (class *State*) and ”transitions” (class *Transitions*). A ”state” can either be a ”transition source” (class *TransSource*) or a ”transition target” (class *TransTraget*). A ”transition” connects one ”transition source” with one ”transition target” (roles *from* and *to*). A ”transition source” or ”transition target” may be connected with an arbitray number of ”transitions” (roles *inv_from* and *inv_to*). A ”transition source” can either be an ”initial state” (class *InitState*) or a ”labeled state” (class *LabeledState*). A ”transition target” can either be a ”labeled state” or a ”final state” (class *FinalState*). A ”labeled state” can be an ”or state” (class *OrState*), an ”and state” (class *AndState*) or a ”plain state” (class *PlainState*). An ”and state” must contain at least two ”and compartments” (class *AndCompartment*). A ”state container” (class *StateContainer*) may contain one or more ”states”. A ”state container” can either be an ”or state” or an ”and compartment”.

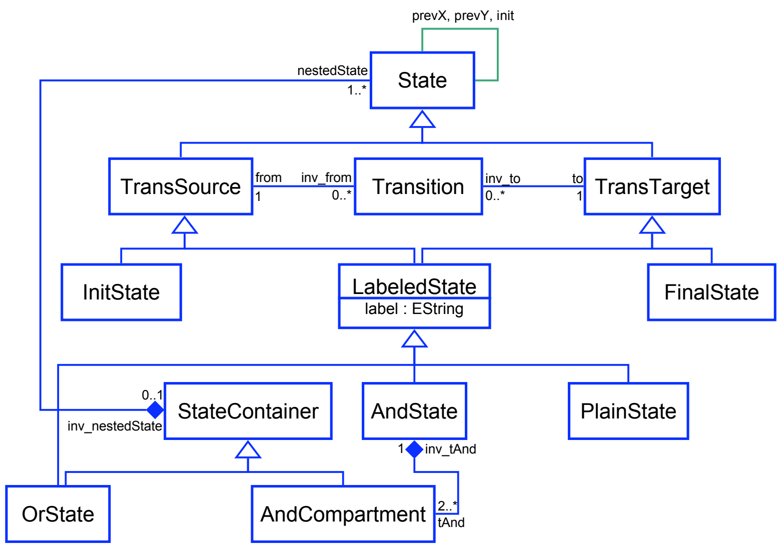

Figure 3: Meta Model of Statecharts

For the layout specification, we added three associations with the roles *prevX*, *prevY* and *init* to the meta model. Their usage will be explained in Section 5.

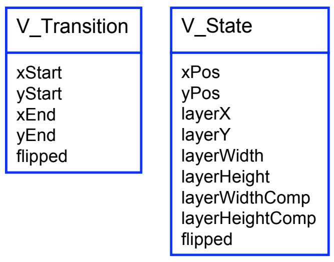

Figure 4: Visual Components

Visual components are created for "states" and "transitions" (Figure 4). For the visual representation, we distinguish between "initial states", "labeled states" and "final states". A "transition" is visualized by an arrow with the start point (*xStart,yStart*) and the end point (*xEnd,yEnd*). It also has the attribute *flipped*, that will be described later. "Initial states" are visualized by a filled circle, "final states" by two circles, and "labeled states" by a rounded rectangle. Its position is described by its top left corner (*xPos,yPos*) and its size by its width (*width*) and height (*height*). A state also has the attributes *layerX*, *layerY*, *layerWidth*, *layerHeight*, *layerWidthComp*, *layerHeightComp* and *flipped*, that will be described later.

3 DiaMeta

In this section, we are going to introduce DiaMeta, the environment the algorithm was implemented in. It is needed to understand the context in which graph transformations and the dynamic layout algorithm are used. In particular it generates an overview of the implementation of graph transformations and the implementation of the dynamic layout algorithm, and how they were combined. The most important fact that is introduced is that graph transformations operate on an internal graph model, whereas the dynamic layout algorithm operates on the object model.

3.1 DiaMeta Architecture

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. This architecture is described in this paragraph. DiaMeta's tool support for specification and code generation, primarily the DiaMeta Designer are postponed to the next paragraph. Figure 5 shows the struc-

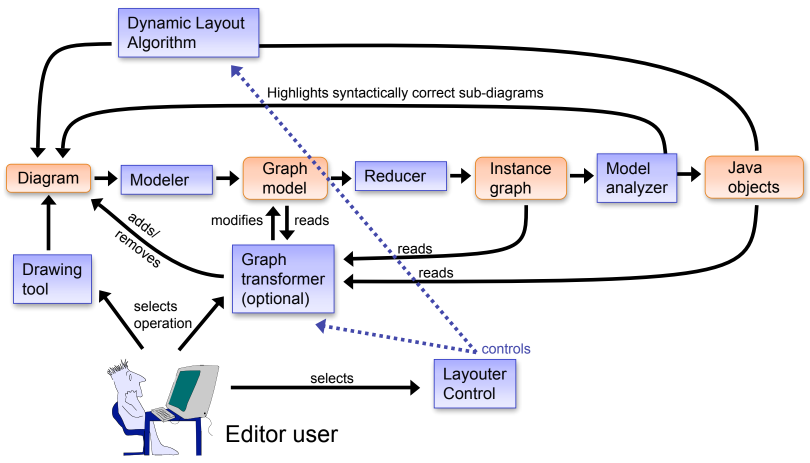

Figure 5: Architecture of DiaMeta

ture 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 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. However, the model analyzer not only checks the diagram's abstract syntax, but also creates the object structure of the diagram's correct subdiagram. The layouter modifies attributes of diagram components and thus the diagram layout is based on the object structure, which allows to access the represented diagram components.
The layouter is optional for free-hand editing, but necessary for structured editing. Structured editing operations modify the graph model by the means of the graph transformer (Sect. 3.3) 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. For our approach, we introduced an additional component that controls the layout, which will be explained in Sect. 3.5.

3.2 DiaMeta Framework

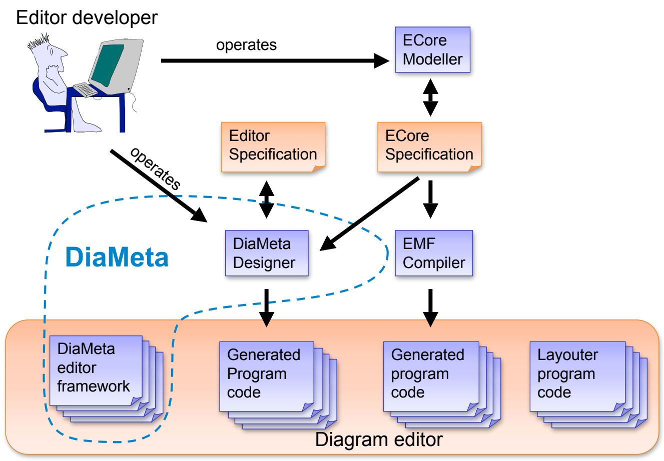

Figure 6: DiaMeta Framework

This paragraph outlines DiaMeta's environment supporting specification and code generation of diagram editors that are tailored to specific diagram languages. The DiaMeta environment shown in Figure 6 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 and the interaction specification. Besides that, he may provide a layout specification, if he wants to define a specific layouter. A language's class diagram is specified as an *EMF* model (ECore specification), created by using the *ECore* modeller. The *EMF* compiler is used to create Java code that represents this model. Figure 3 shows the class diagram of statecharts as an *EMF* model. The editor developer uses the DIAMETA Designer for specifying the concrete syntax and the visual appearance of diagram components, e.g., initial states are drawn as circles. The DIAMETA Designer generates Java code from this specification. In addition, the editor developer can provide a layouter. 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.

3.3 Graph Transformer

DIAMETA offers the possibility to specify structured editing operations via graph transformations. These transformations operate on an internal graph model that is freely modifiable. They are defined in the Designer Specification, by a textual language. A graph transformation is either called by the editor user, or by the *Layouter Control*, which is described in Subsection 3.5.

3.4 Dynamic Layout Algorithm

In Figure 7 we can see a birds-eye view of the *dynamic layout algorithm* that has been presented in [MM07a]: The algorithm is based on the idea that a set of declarative constraints is given, assuring 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 are responsible for updating the attributes, i.e., to satisfy the constraints. The *layouter* is either called directly by the user or by the *Layouter Control*. All potentially violated layout constraints are checked, and the rules that were switched on are collected. Thereafter the new values for the attributes are calculated via attribute evaluation. Now, 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 the new attribute values. Otherwise, the *layouter* has to evaluate the rules again. If the *layouter* does not succeed after a certain number of iterations, the *layouter* stops and returns the old values as result.

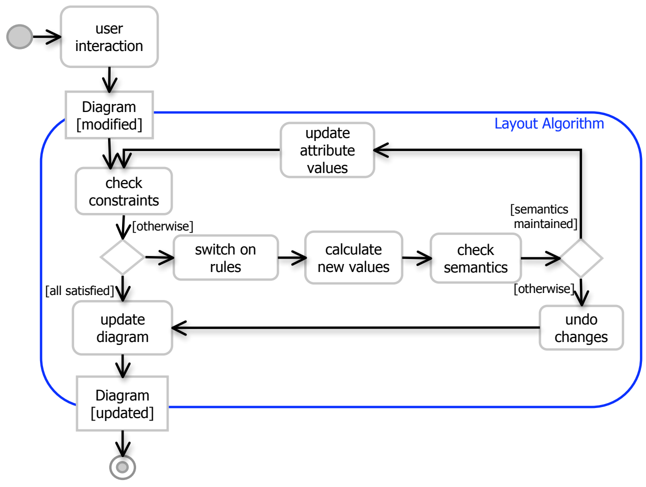

Figure 7: Dynamic Layout Algorithm

3.5 Layouter Control

We offer the editor user two possibilities to apply a layouting strategy. Either the user can choose a layouting strategy, and all phases are applied automatically. This is the ”normal” operating mode. Alternatively, the editor user may apply each phase manually by clicking a button. This is the ”test” operating mode, that turned out to be very helpful during layouter creation.

When applying a strategy automatically, as shown in Figure 5, the editor user has to select the desired layouting strategy (*layouter control*). The *layouter control* then ”controls” the graph transformer and the dynamic layout algorithm. For each phase, the *layouter control* inserts the ”right” graph transformation or dynamic layout algorithm, and initiates the process shown in Figure 5.

When applying a strategy manually, the editor user has to select the desired graph transformation or dynamic layout algorithm. The *layouter control* then inserts this graph transformation or dynamic layout algorithm, and initiates the process shown in Figure 5.

Graph transformations operate on the internal graph model and change this model. Afterwards, the processing steps that are required after a graph transformation are executed, and the diagram as well as the object model are updated. The dynamic layout algorithm operates on the object model and updates the attributes of the object model. Afterwards, the necessary steps are processed and the diagram is updated.

4 Layout Algorithm

The idea of the algorithm is combining graph transformation and the dynamic layout algorithm described in [MM07a]. Therefore, the layouting strategy is separated into different phases. In each phase, either a graph transformation or the dynamic layout algorithm is applied.

The general idea is the following: A graph transformation changes the model. Then the dynamic layout algorithm updates attribute values. Afterwards, a graph transformation undoes intermediate changes in the model. The purpose of intermediate changes in the model is reducing the complexity of the constraints and attribute evaluation rules specified in the dynamic layout algorithm. The separation into different phases splits up the dynamic layout algorithm into small pieces, and again reduces the complexity.

In Section 5 we are going to examine a concrete example, providing the reader with a better understanding of how to separate an algorithm into different phases.

5 Layout Algorithm for Statecharts

The layout algorithm we are going to specify is a simplified version of Sugiyama’s algorithm [STT81]. It has been defined by a combination of graph transformations and the dynamic layout algorithm presented in [MM07a].

Sugiyama’s algorithm is split up into different phases. We have refined this sequence of phases and specify each phase either by graph transformation (GT) or the dynamic layout algorithm (LA) to update the diagram. In the following section we are going to describe these phases in more detail.

01 (GT) Cycle Removal	07 (LA) Calculation of Layer Height and Width
02 (LA) Horizontal Layering	
03 (GT) Dummy Node Insertion	08 (LA) Update Position of States
04 (GT) Connecting of States in Vertical Layer	09 (LA) Update Position of Arrows
05 (LA) Vertical Layering	10 (GT) Dummy Node Replacement
06 (GT) Connecting of States in Horizontal Layer	11 (GT) Undo Cycle Removal

5.1 Phases

01 (GT) Cycle Removal A statechart diagram - when ignoring hierarchical states - is a directed graph that may contain cycles. In the first phase, these cycles are removed by flipping one or more edges.

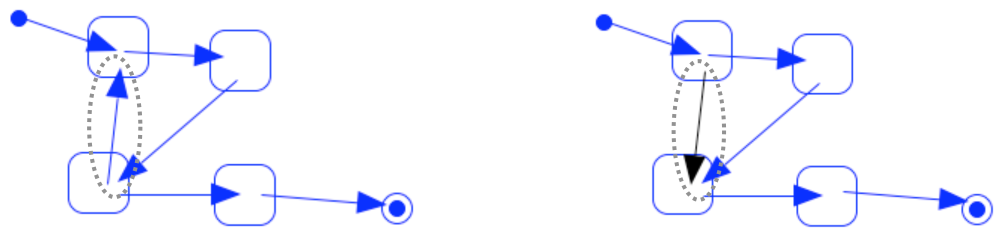

Figure 8: Cycle Removal

```
forall [arrow]=getArrow_() (markUnvisitedArrow_(arrow))
forall [state]=getState_() (markUnvisitedState_(state))
( (
    forall [state]=getStateWithNoOutgoing_()
      (unmarkIngoing_(state)! unmarkState_(state))
    getStateWithNoOutgoing_()
  )!
  flipOutgoing_()
  getStateUnvisited_()
)!
```

We use the following algorithm to do this. First, all states and transitions are marked with a unary hyperedge in the graph model. The mark is removed at all states without any marked outgoing transition arrows. Their ingoing transitions are also unmarked. If all marks are removed, we are done. Otherwise, we know that the diagram contains one or more cycles. If this is the case, the algorithm arbitrarily chooses one marked state, and flips all outgoing edges by changing the value of the attribute *flipped*. Now, the algorithm continues with the first steps and proceeds till all marks were removed.

We implemented this algorithm by the graph transformation shown above, using the hypergraph transformation approach provided by DiaMeta. A statement of the form `forall [a] = getA() (do(a))` calls the rule `getA()` and stores return values in the list `[a]`. Then the rule `do(a)` is applied to all elements `a` of the list. The statement `(do())!` calls the rule `do()` as many times as applicable. Figure 8 shows a statechart before and after cycle removal. In this example, the black arrow (indicated by the ellipse) has been flipped.

02 (LA) Horizontal Layering The attribute evaluation rule that is associated with the transition *trans* takes care of computing the attribute *layerX*. If a state has no incoming edges, *layerX* has the value 1. Otherwise, *layerX* is the maximal distance of a state from the initial state. Figure 9 shows a statechart with the values of the attributes *layerX* and *layerY*[1].

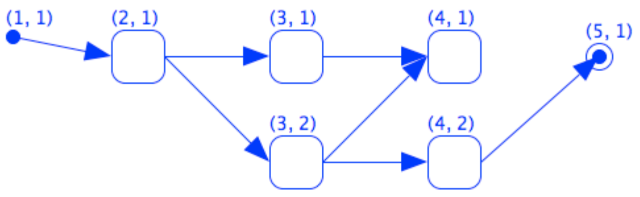

Figure 9: Statechart with Layers

```
trans.to.layerX := max(trans.to.layerX, trans.from.layerX + 1)
```

03 (GT) Dummy Node Insertion As a next step, dummy nodes are inserted, such that arrows only connect states in one layer with states in the next layer (*layerX*).

The algorithm works as follows: For each arrow, it is checked if it connects a state in one layer with a state in the next layer. When this is not the case, e.g., if it connects a state in layer n with a state in layer $n+2$, one or more dummy nodes are inserted.

We have implemented this algorithm by a graph transformation. The inserted *dummy nodes* are *plain states* that are marked as *dummy* (by changing the value of the attribute *dummy*). These *dummy* nodes also have layer attributes. Figure 10 shows a sample dummy node insertion. Here, three nodes had to be inserted.[2]

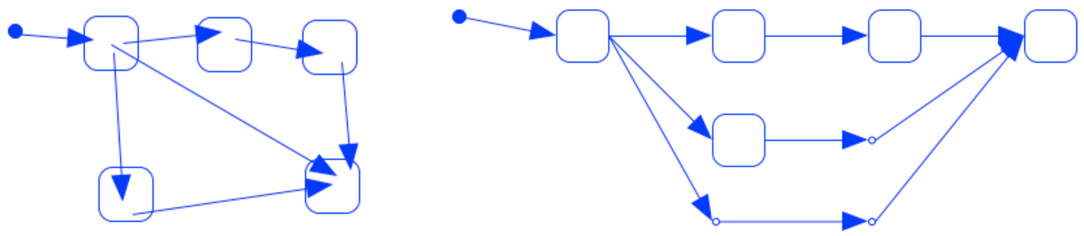

Figure 10: Dummy Node Insertion

04 (GT) Connecting of States in Vertical Layer All states in one vertical layer are connected by the link *prevY*. The elements of a layer can be identified by the attribute *layerX* that was previously set. The sorting of the elements is arbitrary. This connection is realized by a simple graph transformation. The transformation inserts for each link required in the object model an edge in the graph model, which is then translated into the link in the object model. The arbitrary sorting could be replaced by a more sophisticated strategy, e.g., by a strategy that minimizes the number of edge crossings, or by a dynamic layout algorithm enabling the user to change the sorting.

[1] The attribute *layerY* is updated in phase 05.

[2] The diagram in Figure 10 was already layouted. Otherwise, the dummy nodes would appear at the point (0,0).

05 (LA) Vertical Layering All states in one layer are connected by a link (*prevY*). The computation of *layerY* is done by the following attribute evaluation rule, that is associated with the state *state*. If *state.prevY* does not exist, *layerY* is set to 1.

```
state.layerY := state.prevY.layerY + 1
```

06 (GT) Connecting of States in Horizontal Layer After updating the attribute *layerY*, the states in each horizontal layer are connected via the link *prevX*. The elements of a layer are identified by the attribute *layerY*. The sorting of the elements is similar to the value of the attribute *layerX*.[3] This connection is again realized by a graph transformation.

07 (LA) Calculation of Layer Width and Height As stated in the last paragraph, all states in one horizontal layer are connected via the link *prevX*, and all states in one vertical layer are connected by the link *prevY*. The height of a horizontal layer is the height of the highest component in the layer. The width of a vertical layer is the width of the widest component in the layer. The computation is done by a pairwise comparison, i.e., by the following rules, associated with the state *state*. Initially, *state.layerWidth* is set to *state.width* and *state.layerHeight* is set to *state.height*. Figure 11 shows a diagram with states of variable size.

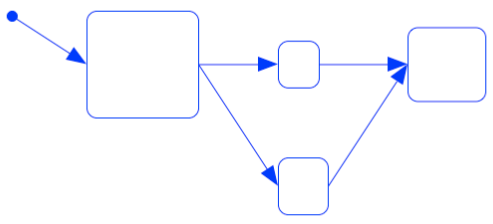
Figure 11: Variable Size

```
state.layerWidth := state.width
state.layerWidth := max(state.prevY.layerWidth, state.layerWidth)

state.layerHeight := state.height
state.layerHeight := max(state.prevX.layerHeight, state.layerHeight)
```

08 (LA) Update Position of States Firstly, the states are layouted. The position of initial states is not updated. This has the consequence that the position of initial states is variable. Labeled states and final states are updated via the following attribute evaluation rules, associated with the state *state*.

```
state.layerWidthComp := state.prevX.layerWidthComp + state.prevX.layerWidth
state.layerHeightComp := state.prevY.layerHeightComp + state.prevY.layerHeight

state.xPos := state.init.xPos + state.layerWidthComp + state.layerX*80
state.yPos := state.init.yPos + state.layerHeightComp + state.layerY*40
              + state.layerHeight/2 - state.height/2
```

state.layerWidthComp and *state.layerHeightComp* is the complete width respectively height of all previous states. *state.init* is the corresponding initial state. *state.layerX*80* and *state.layerY*40* insert spacing between the layers. Components are horizontally centered by adding + *state.layerHeight/2* - *state.height/2*.[4]

[3] It might be the case that one or more numbers are missing, due to the structure of the statechart.

[4] Initial states are not layouted, and hence they are not centered, e.g., as can been seen in Figure 11.

09 (LA) Update Position of Arrows Arrows are also layouted. Attribute updating is performed via the following attribute evaluation rules, associated with the transition *trans*. The transition starts in the middle right of a state, and ends in the middle left of the next state.

```
trans.xStart := trans.from.xPos + trans.from.width
trans.yStart := trans.from.yPos + trans.from.height / 2

trans.xEnd := trans.to.xPos
trans.yEnd := trans.to.yPos + trans.to.height / 2
```

10 (GT) Dummy Node Replacement In this step, all dummy nodes are replaced by bends. This is done by a simple graph transformation. For each dummy node, the incoming and the outgoing arrow[5] are merged, and a new arrow is created. The dummy node is removed. Figure 12 shows the diagram of Figure 10 after replacing dummy nodes by bends. Here, three dummy nodes were replaced.

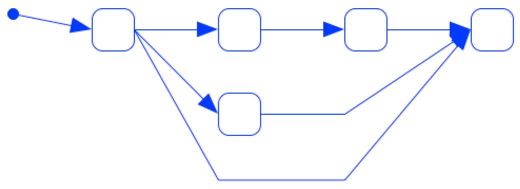

Figure 12: Dummy Node Replacement

11 (GT) Undo Cycle Removal As a last step, the previously flipped edges need to be turned around again. E.g., in Figure 8 (left diagram) we can see the result of undoing the cycle removal as it was done in Figure 8 (right diagram). In this case, the result is the initial diagram. This is not always the case, as there could have been changes in the last steps of the algorithm.

Statecharts with Nested States It is also possible to include nested states. Therefore, the two statecharts are layouted and the size of the containing state is computed from the size of the layouted contained statechart. Figure 13 shows a sample statechart with nested states. Right now, the size of the containing state and the position of the contained initial must be changed by the user, but may be specified via the layout algorithm, e.g., an additional phase.

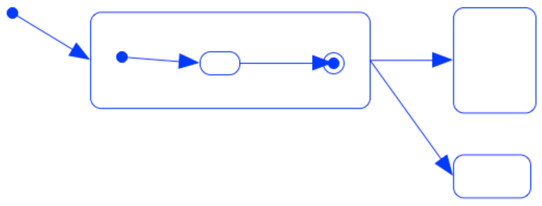

Figure 13: Nested States

General Remarks Besides automatic layouting of nested states, many other enhancements, like space saturation by floor planning, are imaginable. In our example we only used attribute evaluation rules, not the whole functionality of the dynamic layout algorithm presented in [MM07a]. Using this, a static layout algorithm is defined. A dynamic layout is also possible when using the whole functionality of the dynamic layout algorithm. E.g., one could offer the user the possibility to change the order of elements in a layer (*layerY*).

[5] A dummy node has exactly one incoming and one outgoing edge, due to the way it had been created.

6 Related Work

Many comparable tools, like AToM3, GMF, or Tiger, offer the possibility to use a standard layout algorithm, such as FlowLayout. Besides that, some tools, like DiaGen, offer the possibility to use constraints for layout specification. Most tools also allow the developer to write the layouter by hand, as only a small subset of layouter's can be realized by the mechanisms provided. With the approach presented we try to extend this subset.

Many tools support graph transformation, e.g., Fujaba, AToM3 or Tiger, but only rarely use them in the context of layout specification. Guerra et al. presented 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, the graph model is changed via graph transformations, and then the attributes are updated via the dynamic layout algorithm. This gives us more freedom and reduces the size of a layout specification, especially when considering dynamic layout.

UML diagrams, such as activity and statechart diagrams are basically directed graphs. Most approaches for drawing directed graphs used in practice are based on Sugiyama's algorithm [STT81], e.g., an efficient implementation is presented in [ESK05]. Visual language specific layout algorithms based on Sugiyama's algorithm are for example described in [SK06] (activity diagrams) or in [CMT02] (statechart diagrams). Also some work had been performed to take user interaction into account, and to preserve the "mental map". When we take a look at these algorithms, we examine that they are always hand coded. With our approach, this is done by visual programming. In order to create a new layout algorithm, you have to provide graph transformation rules, constraints and attribute evaluation rules instead of plain Java code. This has the consequence that we can benefit from the advantages of visual programming: the creation and adaption of layouting strategies is easier, and hence experiments in this context are made possible.

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 graph aesthetics. With our approach we created a platform for performing this kind of studies easily, not only for graphs but for all kinds of visual languages.

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, we do not only have automatic time-slices, but also time-slices triggered by user interaction. Besides that, freehand editing provides an initial layout that needs to be considered. In DiaMeta, many degrees of freedom are available, that may be considered when creating a new layout algorithm.

7 Conclusions and Future Work

In this paper we presented a layouting approach, that is especially well suited for a *static layout*. It is based on the layout algorithm described in [MM07a] that uses the two concepts constraint satisfaction and attribute evaluation. This algorithm is combined with graph transformations, and the result is a natural way of describing the layout of visual languages. As an example a simplified version of *Sugiyama's algorithm* is used, applied to *statechart* diagrams. We integrated and tested our approach in DiaMeta. The possibility to define a static layout with our algorithm was shown, and it turned out to be the very elegant and intuitive. When Sugiyama's algorithm is explained in literature, it is described step by step. We translated each of these steps into one or more phases. It was then possible to specify the phases independently, each of them either by a simple graph transformation or a simple dynamic layout algorithm. In contrast to other approaches, our layout specification is similar to the initial idea of the algorithm and, and hence the creation is more convenient for the developer.

In the current implementation, graph transformation operates on the graph model and attribute evaluation on the instance of the meta model. For future implementations, we will investigate the possibility to offer "graph transformation" that operates on the instance of the meta model. E.g., we will investigate the EMF Transformer presented in [BEK$^+$06]. This would reduce the complexity of the approach presented. In the current implementation, this is not possible, as DiaMeta is based on a hypergraph approach: Java objects can be created from a diagram, but a diagram cannot be created from Java objects directly.

Right now, only the parts of the diagram that are recognized as correct, are layouted. Other parts are not changed. Consequently, it might happen that visual components are moved by the layouter on top of other components, not recognized as correct. In future implementations we will need to consider parts of the diagram not recognized as correct in our layouting strategy.

For DiaMeta, an enhanced language for graph transformations and the dynamic layout algorithm is planned. For the dynamic layout algorithm, has been introduced already a pattern concept [MM07b] that simplifies the specification.

Future work has to investigate how layouters interfere with user interactions. When creating a diagram, we recognized that users create (one or more) states first. Then they create the transitions between them. The layouter has complicated the process of diagram creation, as it moves components away. In consequence, users turned off the layouter during creation, and turned it on again afterwards. During user interaction, a dynamic layouting strategy that only performs minor changes would be more adequate. Providing users with more freedom, e.g., offering the possibility of rearranging layers, would be an enhancement. These requirements can be realized by using the whole functionality the dynamic layout algorithm offers.

In [MM07b] we focused on dynamic layout, in this paper we focused on static layout. The most important next step will be experiments about the combination of static and dynamic layout in the context of structured and freehand editing. 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.

Bibliography

[BEK$^+$06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Pp. 425–439. 2006.

[CMT02] R. Castelló, R. Mili, I. G. Tollis. A Framework for the Static and Interactive Visualization of Statecharts. 2002.

[ESK05] M. Eiglsperger, M. Siebenhaller, M. Kaufmann. An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing. In *Graph Drawing, New York, 2005*. Springer, 2005.

[GL07] E. Guerra, J. de Lara. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and Systems Modeling* 6:317–347, 2007.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools, Natal, Brazil, 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.

[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, Karlsruhe, Germany*. Springer, 2007.

[SK06] M. Siebenhaller, M. Kaufmann. Drawing Activity Diagrams. In *Proceedings of the 2006 ACM symposium on Software visualization*. ACM, New York, USA, 2006.

[STT81] K. Sugiyama, S. Tagawa, M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 1981.

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