Graph Layout for Domain-Specific Modeling

Denis Dubé
Supervisor: Prof. Hans Vangheluwe

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

The aim of this thesis is to investigate automatic graph layout in the context of domain-specific modeling. Inherent in the nature of domain-specific modeling is the creation of new formalisms to solve the current problem as well as the combined use of multiple formalisms. Unfortunately, graph layout algorithms tend to be formalism-specific, thus limiting their applicability.
As a starting point, all major graph drawing techniques and many of their variants are summarized from the literature. Thereafter, several of these graph drawing techniques are chosen and implemented in AToM^3, A Tool for Multi-formalism and Meta-Modeling.
A new means of specifying formalism-specific user-interface behaviour is then described. By fully modeling the reactive behaviour of a formalism-specific modeling environment, including layout, existing graph drawing algorithms can be re-used without modification. The DCharts formalism is modeled to demonstrate the effectiveness of this approach.

Le dessein de cette thèse est d'examiner le dessin de graphe automatique dans le contexte de modelage domaine-spécifique. Inhérent dans la nature du modelage domaine-spécifique est la création de nouveaux formalismes pour résoudre le problème actuel de même que l'usage combiné de formalismes multiples. Malheureusement, les algorithmes de dessin de graphe ont tendance à être formalisme-spécifiques, ainsi limitant leur validité d'application.
Comme un point de départ, tout les techniques majeurs pour le dessin de graphe et beaucoup de leurs variantes sont résumées de la littérature. Par la suite, plusieurs de ces techniques de dessin de graphe sont choisi et sont appliqué dans le logiciel AToM^3.
Un nouveaux moyens de définir le comportement d'interface utilisateur formalisme-spécifique est alors décrit. En modelant entièrement le comportement réactif d'un environnement de modelage formalisme-spécifique, y compris le dessin, les algorithmes de dessin de graphique existants peuvent être remploient sans modification. Le formalisme de DCharts est modelé pour démontrer l'efficacité de cette méthode.

Acknowledgements

I would like to thank my supervisor Hans Vangheluwe. Without his advice and encouragement I would never have realized my potential to make a contribution to domain-specific visual modeling. I love modeling and coding things that can be seen and this has been a great opportunity to do both. A special thanks to Ernesto Posse, who helped me work out some of the mechanics of thesis writing, including introducing me to the LYX word processor used to create this document. Finally, a big thank you to my parents, without whose encouragement and support I might not have studied for so long, let alone completed this thesis.

Contents

List of Figures

List of Algorithms

Introduction

In recent years a trend has emerged whereby systems of increasing complexity are being modeled. This is due in no small part to the effectiveness of models in aiding the design and/or analysis of complex systems such as those found in the software and physical domains. A good model abstracts a complex system into manageable components or areas of concern. A complex system typically has multiple areas of concern, each of which is best modeled in a given formalism. For example, a model of a software system must deal with both static structures and dynamic behaviours. The class diagram formalism is ideal for partitioning system components into classes and linking them with associations. Class diagrams are however inadequate for expressing, for instance, the interaction of a user with the system. Hence, a model of the interaction subsystem requires a different formalism such as statecharts. In brief, modeling with multiple formalisms allows developers to view their complex systems from multiple viewpoints and to choose the best formalism for specific subsystems.

An ideal formalism for a given subsystem maximally constrains models to the specific problem domain. A constrained model yields two key benefits to the modeler: it eliminates the possibility that the modeler will erroneously create a model that is invalid in the problem domain and it ensures the formalism will closely match the modelers mental model of the problem, thus bridging the conceptual gap between problem domain and model. Thus, an ideal formalism for a specific problem domain and for which appropriate model verification and/or execution is available allows a modeler to fully focus on the problem domain.

Visual formalisms provide an important additional means of bridging the conceptual gap between problem domain and model. There are two key differences between visual and non-visual formalisms. The obvious one is that the symbols of the non-visual formalism are replaced by graphical icons and (typically) arrows denoting relationships. The second is that the visual arrangement (layout) of these icons and arrows is very important. Indeed, a model with a good visual layout allows a user to extract information at a glance whereas a poor layout is far more challenging to decipher. As any modeler knows, manually drawing good layouts is very time-consuming. Moreover, when a model is modified, the changes often downgrade the existing good layout to a poor one. Hence modelers need robust visual modeling tool support that can lighten their workload.

The main contribution of this thesis is a new framework for modeling the reactive and layout behaviour of a visual modeling environment that supports multiple formalisms simultaneously. This model-based framework contrasts sharply with the hard-coded and inflexible approaches found in other visual modeling tools. The framework, its implementation, and an example formalism illustrating its usefulness are described in chapter 3.

The framework requires access to automatic layout implementations in order to handle layout considerations. Hence, various means of obtaining automatic layout were implemented. The design, implementation details, complexity analysis, and performance of these layout techniques are described in chapter 2.

The choice and implementation of the layout techniques could only be made with a thorough knowledge of: graph theory, the elements of a good layout, and existing graph drawing techniques. Hence, chapter 1 presents a rather exhaustive review of the graph drawing literature.

1 Graph Drawing

Introduction

In our approach to multi-formalism modeling, every model is in essence a graph. For each such model, an infinite number of visual layouts exist, resulting in anything from a meaningful (from the modeler's point of view) drawing to one that is misleading and error-prone. Given that hand-crafting layout is highly time-consuming, support for automatic layout is crucial in modern visual modeling tools. Unfortunately, no single graph drawing technique exists, nor is likely to ever exist, that can draw any given graph in the most meaningful fashion. Moreover, those graph drawing techniques that come closest to creating ideal drawings do not scale very well with the number of vertices in a graph. On the other hand, it should be noted that in the context of interactive, possibly domain-specific visual modeling, most visual models are rather small. This, since it is easier to understand a complex system rendered as multiple smaller models of domain-specific subsystems and taking advantage of hierarchical (de-)composition than understanding one large monolithic model. In any case, graph drawing algorithms applicable to graphs small and large are both reviewed in this chapter.

The first part of this chapter, section 1.1, makes the link between real-world problems and graphs, and the effects of model constraints on graphs.

A good layout can only be systematically achieved by understanding what makes it good. Hence, section 1.2 delves into the measurable metrics (visual aesthetics) that indicate the quality of a layout.

Ideally, every metric should be optimized to obtain an optimally meaningful drawing. Section 1.3 begins by explaining why this is not possible. Thereafter, a comprehensive overview of graph drawing techniques from the literature is presented, including variant techniques and the specific visual aesthetics optimized by each technique. The primary focus is on techniques applicable to small graphs. However a number of techniques applicable to large graphs are also given.

1.1 Graph Basics

Graph drawing techniques operate directly on the structure of graphs. Therefore knowledge of basic graph theory is a prerequisite to understanding graph drawing techniques. The glossary in chapter 4 describes all the graph terminology used throughout this thesis with concise, mostly natural language definitions. Readers familiar with graph theory terminology are encouraged to at least quickly read the glossary, as it is written specifically from a graph drawing point of view. On the other hand, for more mathematical definitions and a comprehensive overview of graph theory, the following book is recommended [Die05]. In section 1.1.1, the link between problems in the real-world, domain-specific modeling, and graphs is made. Closely related to the previous section, section 1.1.2 discusses how different formalisms constrain models to different graph structures.

Problem	Formalism	Graph type
Modeling vehicle traffic systems. Roads are edges, intersections vertices.	Traffic	Digraph
Modeling and simulation of large scale hierarchical systems. States are vertices, edges are containments and transitions. (Discrete EVent System)	DEVS	Digraph, Compound
Includes modeling and simulating manufacturing and network problems. Vertices are event generators/recorders/etc. and edges are transitions. (General Purpose Simulation System)	GPSS	Digraph
Modeling websites. Web pages are vertices, hyperlinks are edges (but not hyperedges). (World Wide Web visualization)	WWW	Digraph, large, sparse
Highly generic modeling. Entities are vertices, edges are relationships. (Entity-Relationship diagram)	ER	Hypergraph
Modeling of many software engineering problems. UML formalisms include use-case, collaboration, sequence, deployment, and class diagrams. (Unified Modeling Language)	UML	Digraph, mixed-graph
Modeling of analysis, design, and reverse engineering aspects of software.	DataFlow	Digraph
Modeling of scheduling activities with dependencies.	PERT	Digraph, hierarchical
Modeling causal continuous time systems. (Causal Block diagrams)	CBD	Digraph
Modeling and simulation of reactive behaviour.	Statechart	Hypergraph, compound
Modeling and simulation of concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic systems.	Petri-net	Digraph

Table 1.1: Linking real-world problems to graph structures

1.1.1 Modeling problems as graphs

A vast number of problems in the real-world can be mapped to models of a given formalism. A formalism defines constraints on the types of models that can be constructed. This is described in greater detail in 1.1.2. The underlying structure of a model is simply a graph. The following table provides examples of common problems, the formalisms that allow them to be modeled, and a brief description of the resulting graph structure.

1.1.2 Model constraints

A formalism constrains the form of a model and hence of the resulting graph structure. Ideally, the formalism will constrain the model maximally [VdL04]. Such constraints force the user to construct only those models that are both syntactically and semantically[1] correct models. Thus constraints prevent a user from accidentally constructing an incorrect model, as well as provide some guarantee as to the model's correctness.

Model constraints can take many forms. At the abstract level, the graph structure itself is constrained. Models will typically partition the vertex and edge sets. Given these sets, it is possible to constrain whether or not an edge is possible between given vertex types. For example, in a Petri-net diagram it would not make sense to allow edges between two places or two transition states. If an edge is possible, then the types of edges can also be constrained. An example of this is a statechart, where an edge between state A and B can be of two types, containment or transition. However, if A is already contained by B, or vice versa, then the edge can only be a transition from A to B. For each given type of edge, the maximum number allowed per vertex can be constrained, also called

[1]Semantics are more difficult to check than syntax, thus a static analysis of this may be very limited

edge cardinality. Finally, a model can constrain the maximum number of occurrences for a given type of vertex.

Models have other properties that will affect the graph structure, as outlined in table 1.1. For example, a PERT chart yields a highly hierarchical digraph, since edges are between activity vertices, and each activity depends on the completion of some other activity. On the other hand, a class diagram is an example of a mixed-graph. The mixed-graph classification is due to the fact that inheritance and directed class associations yield directed edges, but undirected associations yield undirected edges. Moreover, class diagrams are not necessarily hierarchical, which is discussed in section 1.3.1. Another special case can be found in Entity-Relationship diagrams, where a relationship can be connected to any number of entities. Thus the relationships are hyper-edges.

Model constraints may also occur at the drawing level. In most models, vertices are not merely points, but are instead meaningful icons with labels. Edges, however, are not necessarily drawn as lines or arrows. In DEVS and statechart models, some edges denote that one vertex contains other vertices inside of it. Thus a parent vertex must either be constrained to be large enough to contain its children, the children forced close together, or both. Another possibility is that a certain vertex type must be drawn in a certain region of the drawing. An example of this is an electrical system model, where it is typically required that only a single ground exist and that it be drawn below all other vertices.

1.2 Visual Aesthetics

Visual aesthetics are the measurable qualities of a drawing. The goal of graph drawing is to optimize these visual aesthetics according to the needs of the domain-specific formalism. It is expected that by meeting the visual aesthetic needs of a formalism that the resulting drawing of a model in that formalism taps the full potential of the human visual system in understanding the model's information. As a quick illustration of how this can be, consider two drawings of the same graph and model: one is randomly drawn and the other is drawn such that vertices are visually close in proportion to the length of the edge path between them. The randomly drawn graph reveals no model information at all, certainly not with any guarantee. On the other hand, the second method visualizes graph theoretic distances, thus revealing at a glance which model components are related. This is so fundamental it is not even considered a visual aesthetic and all graph drawing techniques, to a greater or lesser extent, visualize these graph theoretic distances.

Although visual aesthetics are highly domain-specific, some generalization can be made about their relative importance. In [PCJ97], an experimental study of the importance of three visual aesthetics is conducted. The study was conducted on graphs with no inherent meaning and subjects were only asked graph theoretical questions, such as what the shortest path between two vertices was. It showed that both edge crossings and edge bends are quite detrimental to human understanding of a graph, in the sense that both increase the probability of incorrectly assessing the structure of a graph. On the other hand, the study results showed symmetry had no measurable impact on human cognition. A followup study, [WPCM02], reveals a visual aesthetic not previously considered by the graph drawing community. This new aesthetic, continuity, is the measure of the angle formed by the incoming and outgoing edges of a vertex. For the task of finding a shortest path between two vertices, continuity can become even more important than edge crossings. This demonstrates how the ultimate goal of maximizing human cognition of graphs can sometimes differ from optimizing well known visual aesthetics.

In the following subsections, the most important visual aesthetics that are commonly optimized by graph drawing techniques are summarized.

1.2.1 Graph Area

A graph that can fit on one page, yet display all the information contained in similar graph that requires two pages, is far superior since it eliminates the need to flip back and forth between the pages. In a print medium, this can be particularly detrimental to human cognition as it then requires following references rather than simple lines. Similarly, a graph with an aspect ratio that closes matches that of a display device, typically a 4:3 ratio, presents information to us much more efficiently than a very wide but short graph or a very tall but narrow graph. It is a well known fact that humans have a preference for views with an aspect ratio corresponding to the Golden ratio (approximately 1.618). For example, business cards have dimensions corresponding to this ratio, and the 4:3 ratio of standard display devices is somewhat close, although 5:3 would be much closer.

1.2.2 Vertex Placement

When placing vertices, it is best to place them uniformly, avoid overlap, and emphasize symmetries. The uniform distribution of vertices in a graph minimizes the area of the graph, a visual aesthetic already mentioned. Vertices drawn in an overlapping fashion, a common occurrence when vertices are not point-sized, hides information and thus severely reduces the readability of the drawing. A vertex placement that emphasizes symmetries is more pleasing to the eye. For example, binary trees are often drawn in a symmetrical fashion, with the children of a given node equally balanced to the left and right of it. However an experimental study of symmetry has failed to show a statistically significant improvement in the human understanding of graphs drawn to emphasize symmetry [PCJ97]. Note that the scope of this experiment was limited to syntactic tasks such as determining the shortest path between two vertices rather than semantic tasks, tasks that are application-specific and require interpreting the graph.

1.2.3 Edge Crossings

The visual crossing of edges is extremely detrimental to human understanding of graph information. This is experimentally verified in [PCJ97]. The angle formed by edge crossings has an important impact on cognition as well. If the crossing edges form a right angle, then they are very easy to distinguish. If instead they form a very small angle, which edge is which becomes ambiguous. Indeed, in some cases increasing the crossing angles is preferable to minimizing the crossings. A nice illustration of this can be found on page 30 of [BM01]. A final situation to avoid is where an edge crosses a vertex, particularly when the vertex has a visual size greater than that of a point.

1.2.4 Edge Bends

Ideally, an edge should contain no bends at all, since a straight line is far easier to follow than a snaking poly-line. Unfortunately, avoiding overlapping as well as crossing inevitably results in bends, thus one can merely minimize them.

1.2.5 Direction of flow

Whenever applicable, directed edges should move only in one given vertical and/or horizontal direction. For example the majority of the edges could flow from the top to the bottom of the drawing, or from the top-left to the bottom-right. The advantage to drawing a graph in such a fashion is that it is highly revealing of the underling graph structure, particularly in the case of hierarchical graphs. Indeed a hierarchical graph drawn in this fashion makes finding source vertices and paths to sink vertices immensely easier compared to drawings that do not respect direction of flow.

A simple example of this is a tree drawn with the root at the top and the leaves below. In this case, each edge flows from a parent node to a child node below its parent, though the child may be either to the left or the right of its parent.

1.2.6 Edge Length

It is readily apparent that a long edge is far more difficult to trace from origin to destination than a similar short edge. Hence edge lengths should always be as short as possible. A similar aesthetic is that the variation between edge lengths be minimal. However the minimization of this last aesthetic yields does not appear to have much impact on the readability of a drawing. In [PMCC01], the results are so mixed one might even suggest that maximizing edge length variation is useful.

1.2.7 Mental Map

The mental map visual aesthetic differs from the others in that it applies only to dynamic contexts. Consider a user having just applied an automatic layout technique to a model in order to obtain a drawing. Later the user then modifies the model slightly, by adding an edge or a vertex for example, and again applies automatic layout. The user is familiar with the first drawing, having made a mental map of where vertices are located and in what order. Thus if the drawing of the modified model is vastly different, the user will have to spend time building a mental map of the entire model all over again. Satisfying this visual aesthetic in a meaningful fashion can be difficult, since it requires essentially not moving the vertices from their positions in the original drawing, thus conflicting will most of the other visual aesthetics.

1.2.8 Vertex Connections

A large angle between two edges connected to the same vertex, particularly a point vertex, makes them easy to distinguish. Conversely, a small angle between two edges can confuse the viewer when one edge has an arrow head and the other does not. The drawing is ambiguous. This aesthetic is known as angular resolution and merits careful consideration in the edge routing phases of a graph drawing technique.

Most models use icons that are not point sized to represent a vertex. This makes the connection problem even more difficult. Naively connecting edges to the center of such a vertex results in overlap. For rectangular and circular icons, it is possible to efficiently find the intersection of an edge and the boundary of the icon. Otherwise, connections are made to specific ports on the outside of such an icon. Some drawing techniques require the use of edge ports. Moreover, some orthogonal layouts even require the ability to dynamically change the location and number of ports for a given vertex icon. Another concern is that with sufficient edges entering the ports of a vertex, the arrowhead drawings will become unreadable. This can be avoided by grouping incoming and outgoing edges into separate input and output ports.

1.3 Techniques for graph drawing

The goal of graph drawing techniques is to make the visualization of information as readily readable as possible. Thus all graph drawing techniques seek to draw graphs in a fashion that optimizes some subset of the visual aesthetics of the previous section. Although the aesthetics can be translated into mathematical constraints, optimizing all the aesthetics is, in general, an impossibility [BM01]. One problem is that some of these constraints are contradictory with respect to each other. Some examples: minimizing edge bends conflicts with minimize edge length since adding more bends allows shorter edge lengths, avoiding overlap of vertices and edges conflicts directly with area minimization, and avoiding crossings between edges and vertices conflicts with both edge length and area minimization.

A second problem is that solving even seemingly simple sub-problems for a single visual aesthetic is often hard, in terms of theoretical complexity, resulting in prohibitively long computation times for anything but small graphs. For example, finding the minimum number of edge crossings in a k-layer graph has non-polynomial complexity as does the seemingly simpler sub-problem of a 2-layer bipartite graph. This is shown in greater detail in section 1.3.1.

Thus there is no single graph drawing technique that will work for every graph. Each domain-specific visual formalism will prioritize the visual aesthetics differently. The drawing technique will then solve for these aesthetics in order of priority using heuristic algorithms, unless the graph is so small an optimal solution is possible. Different formalisms may have very different graph structures. For example, a UML class diagram can have both directed and undirected associations, yielding a mixed graph containing both directed and undirected edges. Preserving the direction of flow in such a graph requires a different approach than for digraphs, as is shown in [Eig03]. Formalisms may also impose special rules about where and how certain types of vertices and edges may be drawn. Thus one must find the correct graph drawing technique for a given formalism and then customize it to handle formalism specific issues that arise. The following subsections summarize the major, as well as many minor, approaches to automatic graph drawing techniques.

1.3.1 Layered

Layered graph drawing techniques partition vertices into layers and then draw the edges between the layers. According to [BM01], the concept of a layered graph drawing technique first arose in 1977. In 1981, with the publication of Sugiyama's method [ST81], this approach to graph drawing gained a great deal of attention and has since been implemented in many tools.

A layered graph is a digraph with some restrictions. First, the graph should have an overall direction of flow, because this technique will assume it exists regardless. For example, a tree dependency graph flows from what nothing depends on to what depends on many others. On the other hand, a causal block diagram does not possess an oriented flow, and hence does not benefit so much from this technique. A second restriction is that the input digraph be acyclic. In the layer assignment section, preprocessing techniques to make cyclic graphs acyclic are discussed.

Not all graphs can easily be partitioned into layers. As a trivial example, consider the graph of a list structure. Since we do not typically allow edges to be drawn between vertices of the same layer, the resulting drawing is very tall but narrow, severely violating the aspect ratio visual aesthetic. Similarly, a class diagram where inheritance arrows are used for determining layers, will result in a very short but wide graph. This last observation assumes the class diagram designer respected the good design rule of limiting inheritance to a depth of two and at most three.

This drawing technique deals with the following visual aesthetics, when implemented to the fullest extent, given roughly in the order that they are optimized for: direction of flow, area, mental map, vertex overlap and uniform vertex distribution, long edge lengths, edge crossings, edge bends, and symmetry. Surprisingly, given the general complexity of automatic layout, this technique deals explicitly with almost all the visual aesthetic criteria. This broad coverage of aesthetic criteria, combined with its moderate implementation difficulty, is no doubt why it is so widely implemented in graph layout tools. Moreover improvements to the the many algorithms it uses continue to be published to this day. The overall running time of most layered drawing techniques is not well analyzed, since some of the heuristics employed are of the iterate until no further improvement occurs type. Nonetheless, the time complexity can be assumed to be roughly quadratic.

The layered drawing technique is typically broken up into three major steps: layer assignment, crossing reduction, and horizontal placement. Important extensions to this technique include satisfaction of the mental map visual aesthetic and the ability to deal with compound graphs.

Layer assignment

The layer assignment phase of the layered drawing approach assigns each vertex in the input graph to a layer. There are quite a few heuristic methods for accomplishing this task, each of which will work better or worse for certain aesthetic criteria, as well as overcoming the non-polynomial complexity nature of both area minimization and edge length minimization. A method that uses the existing positions of vertices to determine layering not only preserves the user's mental map, but is simple to implement. Usually though, an initial drawing is not available or not very good, thus an automatic layering needs to be generated from just the underlying structure of the graph.

Layering algorithms have some restrictions placed on them. A layering that satisfies these restrictions is known as a proper layered hierarchy. The first restriction is that edges are not permitted between vertices belonging to the same layer. The second is that to preserve direction of flow, vertices of a given layer never have incoming edges from a layer that can be reached by outgoing edges. In other words, the descendants of a layer do not flow back into the parent layer. This restriction implies the third, namely that the input graph must be acyclic. A final restriction is that if an edge occurs between two layers then the two layers are adjacent. For example, one can think of the layers as an ordered list, where adjacent layers are parents/children of each other. Edges that do traverse more than one layer require special handling. This takes the form of dummy vertices, representing the bends in the edge, that are assigned to each intervening layer the edge traverses. These vertices are then treated like real vertices by later phases of the layered drawing algorithm, with all the overhead this entails.

Cyclic graphs are common and one would like to draw them anyway. Thus a preprocessing step must be applied to break cycles. This is done by reversing the directions of some of the edges, from the point of view of the layered drawing algorithm. The edge is ultimately drawn in the correct orientation. Clearly the minimum number of edges should be reversed for a drawing of high quality. This problem, also known as the maximum acyclic subgraph problem, is NP-hard [Kar72]. A variety of heuristic algorithms to solve this problem can be found in [BM01].

A naive layering algorithm, longest-path, is easy to implement by simply assigning each vertex to a layer according to its discovery order by a depth first search starting at the root vertex. Another approach, Coffman-Graham, involves minimizing the width of each layer, so as to improve the area visual aesthetic, by constraining each layer to a parametrized width. The pseudo-code for these algorithms can be found in [BM01]. Sander describes a method that minimizes variations in edge widths using a one-dimensional spring embedder in [San96c], although this only applies if edge orientation is not important. Another method for minimizing area is given by [TNB04], with the advantage that one does not need to manually set the maximum layer width. A powerful method that minimizes edge lengths and bends by minimizing multi-layer edges is formulated as an integer linear programming problem in [GKNV93]. Finally, an experimental comparison of three common layering methods: longest-path, Coffman-Graham, and Ganser ILP, is presented in [HN02].

A new approach to layering is taken in [ESK04]. Instead of requiring a proper layered hierarchy, long edges are represented with only two dummy vertices at both ends of the edge. This requires some adjustments to the later phases of the layered drawing technique which is described in the paper. The advantage of drastically reducing the number of dummy vertices cannot be understated. Every phase of the layered drawing technique takes a performance hit for each additional dummy vertex. The authors claim a running time of $O((|V| + |E|)log|E|)$ and a memory requirement of O(|V|+|E|), both of which are substantially better than previous approaches.

Crossing minimization

The crossing minimization phase of layered drawing reduces edge crossings. It turns out that it is sufficient to re-order the vertices inside each layer without dealing with actual coordinates. The most common approach to reduce crossings in the entire graph, the layer-by-layer sweep, deals with only two layers at a time [BM01]. Using the ordered list of layers from the first phase, the layer-by-layer sweep fixes the order of the first layer and re-orders the second layer. Then the second layer is fixed and the third re-ordered, until the last layer is reached, whereupon the direction of the sweep is reversed. Since layer-by-layer sweep operates on a very local scope, only two layers at once, and yet crossings can arise from global ordering of the layers, this technique requires many iterations until the number of edge crossings ceases to reduce further.

This phase of the algorithm is typically the most computationally expensive and with good reason. As mentioned earlier the bipartite crossing minimization sub-problem is NP-hard [EW94a, EW94b, GJ83]. This is because the only way to find the minimum number of crossings between two layers of vertices is to literally try every single possible ordering of vertices in both layers. Naturally, most graphs have far more than two layers, thus this computationally hard problem must be solved many times. The reader may wonder why one does not consider all layers simultaneously. Indeed, experiments have shown that considering only two layers at once yields less than optimal solutions for the entire graph [JLMO97]. The trouble of course is that if the two-layer problem is so difficult, the multi-layer problem is even more difficult.

Many algorithms for reducing crossings in layered graphs, using the layer-by-layer sweep approach, exist. A very comprehensive overview of them can be found in [BM01]. Only a sampling of the more interesting ones are reproduced here. Note that a layer-by-layer sweep is typically terminated once it is detected that crossings are no longer being minimized. A simple method, using crossing numbers, is described in [BM01], and can be used to find a lower bound on the number of crossings. A more powerful method that gives the exact number of crossings is given in [BJM02]. This method is simple to implement and has $O(|E|\log|V|)$ complexity where $|E|$ is the number of edges between two layers, and $|V|$ is the number of vertices on the smallest of the two layers.

The most naive algorithm involves permuting every vertex in every layer, thus guaranteeing an optimal solution, albeit with a running time of $O(n!)$. A more practical approach, also yielding optimal solutions, is the branch-and-cut algorithm by [JM97]. Although this algorithm is also non-polynomial in complexity, whenever the layers have at most 60 nodes, the branch-and-cut algorithm runs as fast or faster than most heuristic methods. Also, this method can also be used in a heuristic fashion by simply interrupting the solver after a given amount of time, thus yielding the best solution found at that point, although using it in this fashion yields poorer results than far simpler heuristic methods.

The most commonly used algorithms are the barycenter and mediancenter node weighting heuristics. Both algorithms compute a metric for each node in the layer being re-ordered based on the order of the neighbors in the fixed layer, and then re-order the nodes by sorting them according to that metric. For barycenter, the metric computed for each node is simply the sum of the order of each neighbor in the fixed layer divided by the node's degree. In this case, the degree of the node is the number of edges it has in the neighboring fixed layer, not in the entire graph. If two nodes receive the same metric, a random tie breaker is recommended in [Pat04]. A randomized sorting algorithm is also used in [San94], except that Sander's implementation goes further and keeps the tie breaker in memory to allow backtracking if crossings are not reduced. An additional strategy, mentioned in [San95], is to use the mediancenter heuristic to break the ties. This coupled strategy is called barymedian, as it uses barycenter first, and mediancenter as a fall-back. The reverse is also possible, and is called medianbary.

For mediancenter, the only difference is that instead of averaging, one picks the order of the neighboring vertex that is median. If there are no neighboring vertices, then the median is 0. If there are an even number of neighboring vertices then two medians exist, and the leftmost one is chosen. Both heuristics are intuitively appealing, they position vertices such that they are as close as possible to their neighbors and hence reduce the crossings. However, the barycenter method has the advantage of yielding a drawing with no crossings at all whenever this is possible. On the other hand, [San95] claims that if the average degree of the vertices in the graph is low, mediancenter is usually more appropriate. Furthermore, it is mentioned in [BM01] that mediancenter is guaranteed to give a solution with no more than three times the optimal number of crossings.

Three more variants that improve the results of barycenter and mediancenter are described in [BM01]. The average median heuristic modifies the mediancenter heuristic by using the average of both the leftmost and the rightmost medians whenever the degree of a vertex is even. Similarly, the semi-median heuristic simply uses the barycenter heuristic if the vertex degree is even. The third and highly refined variant of mediancenter is called the weighted median heuristic and was developed by [GKNV93]. As in the previous variants, mediancenter is used for vertices of odd degree. If the vertex degree is two, then barycenter is used. For all other even vertex degrees, an interpolated value is used to bias the vertex toward the side where the neighboring vertices are more densely packed. The formula is as follows:

$$wmed(u) = \frac{\Pi_1(v_{j/2}) \bullet right + \Pi_1(v_{j/2+1}) \bullet left}{left + right}$$

$$left = \Pi_1(v_{j/2}) - \Pi_1(v_1)$$

$$right = \Pi_1(v_j) - \Pi_1(v_{j/2+1})$$

In the formula, the vertex u has j neighbors in the fixed layer. Thus, $\Pi_1(v_j)$ gives the order of the last neighbor in the fixed layer of vertex u.

The running time of barycenter, mediancenter, and their hybrid variants is O(n). The sorting requirement increases this to at least O(n log n) in the worst case, where n is the number of vertices in the re-ordered layer. However, since the layers are sorted repeatedly, the layers are typically very close to the sorted order. Thus the sorting barely requires more than O(n) comparisons of vertices. Ultimately, it is difficult to analyze how many sweeps a graph requires. In particular, downward and upward sweeps tend to undo each others efforts, and thus for most graphs the sweeps could go on forever without converging. Thus the total running time of the crossing reduction is usually bounded by running the layer-by-layer sweeps for at most a constant number of iterations. In practice, this not only yields good results, it is the fastest crossing reduction method.

In [LART86], it is observed that with barycenter at least, upward and downward sweeps of the layers can yield very different metrics. This causes vertices to oscillate between one sweep and the next. To improve convergence, they recommend that a third pass be used where the metric is the average of the upward and downward sweep metrics.

A more sophisticated approach to crossing minimization is given in [ML03]. Here an algorithm based on the GRASP methodology is applied, which stands for greedy randomized adaptive search procedure. This algorithm uses many different strategies, including random vertex orderings, barycenter, and permutations of neighboring vertices. Of course, as the word search indicates, this algorithm has a higher running time than pure barycenter. No running time is given beyond experimental results, however one can safely conclude the running time is non-linear. On the other hand, the results of this algorithm are often optimal or very nearly so, as extensive comparisons with other heuristic algorithms show.

A few algorithms exist for reducing crossings by considering more than two layers at a time. One such method involves planarizing the graph and is given by [Mut97]. Essentially, the method attempts to remove as few edges as possible from the graph until planarity is achieved, and then re-inserts them into the final drawing. The edge re-insertion algorithm attempts to route edges to minimize edge crossings, but crossings are inevitable nonetheless. In practice the resulting crossings tend to yield larger angles between crossing edges than do other methods, thus making the graph more readable. Another, much older method attributed to Tutte and described in [EL89], involves fixing the top and bottom layers, and then choosing the order of each node as the average of the node's indegree and outdegree. This can be formulated as a system of linear equations, and the solution is very similar to doing a layer-by-layer sweep with barycenter.

Horizontal placement

The horizontal placement phase assigns coordinates to each vertex. The name, horizontal placement, is derived from the convention that most layered drawing are drawn in layers starting with a root layer at the top and successive layers moving down. Thus the vertical coordinate assignment is easy to determine from the layering assignment. One does need to keep an appropriate vertical separation between layers to avoid vertex overlapping. Optionally, the vertical separation can be increased to take into account the slope of nearly horizontal edges, making them more readable[GKNV93]. This last situation occurs frequently with long, multi-layer traversing edges, that bend abruptly at their start and/or end. Thus we primarily deal with horizontal coordinate assignment in this phase.

The primary objective of this phase is to minimize edge bends and increase symmetries. Since changing the ordering provided by the second phase would add edge crossings, straightening the edges results in at least some sacrifice of the area aesthetic. A horizontal placement method essentially assigns coordinates such that if any two vertices are neighbors then they have identical coordinates, if such is possible. If more than one child vertex has the same parent, or a parent has more than one child, the vertices are placed as close possible to the horizontal coordinate of the parent or child, thus giving the drawing a symmetrical look. Edges that traverse multiple layers and that were given dummy vertices in the first phase are assigned horizontal coordinates such that the dummy vertices all share the same horizontal coordinate if possible. Typically a long edge will bend at the top and/or bottom, whereas the short edges will not be permitted to bend at all.

A problem formulation that provides the exact solution for horizontal placement exists. It optimizes for straight vertical edges and narrow layer widths. Since it involves solving a quadratic objective function, an NP-hard problem, this method is of little use for problems of reasonable size. A description of it can be found in [BM01, ST81]. An improved formulation of the problem, in [GKNV93], consists of constructing an auxiliary graph. The horizontal placement problem is thus transformed into a layer assignment problem, that is then solved using a network simplex method. This method returns optimal solutions if given enough computation time. Since network simplex is non-polynomial, the fact that approximations of the optimal solution are returned if the solver is halted after a predefined time-limit proves quite useful.

There are also a variety of heuristic approaches to the horizontal placement problem. The first such is the priority method first published by [ST81] and described in detail in [Ste01]. The method is very similar to the barycenter and median heuristics already discussed in the crossing reduction phase. First, each vertex is assigned a down-priority equivalent to its outdegree and an up-priority equal to its indegree. If the vertex is a dummy, an edge bend, and its neighbor is also a dummy, then it receives infinite priority instead to ensure straight long edges. Then the algorithm loops over upward and downward sweeps, until no vertices move. Each sweep uses the barycenter or median methods to calculate the desired grid position of each node, which will be very near the position of the node's children or parents. If the node is not in its desired grid position, it attempts to move there. If another node occupies the desired position, it will attempt to push it into the next grid position, a recursive process. The process succeeds only if each pusher node has greater priority than each pushed node.

In [San94, San96c], Sander presents a two part method consisting of a pendulum and a rubber band analogy from physics. The intuition behind the pendulum method is that each vertex is a ball swinging on edge strings from a fixed layer of vertices. Since gravity imposes a horizontal force to each vertex ball proportional to the angle of the edge string, this method balances out the horizontal positions of the vertices while implicitly respecting the vertex order from the crossing reduction phase. Furthermore, a minimum separation distance is enforced between each ball, which can be visualized as the result of a ball that is larger than the actual vertex. This minimum separation distance becomes important in the rubber band method since each vertex will need some space in which to move to the left or right. The idea behind the rubber band method is to straighten out the edges, something the pendulum method ignores. This is done by treating each edge as a rubber band, so that a vertex having one parent vertex and one child vertex will have force exerted on it such that the difference between the child and parent positions is minimized, and thus yielding a straight edge. A variation of this method appears in [San96b], whereupon long edges with dummy vertices are forced into strictly vertical lines. The only exception is the top and bottom part of a long edge, which are permitted to bend. This is called a Manhattan style layout.

In [BJL01], an alternative method for Manhattan style layout is presented. This involves two essentials steps. The first step deals with the positioning of long edges composed of dummy vertices. Without violating the crossing reduction ordering, a leftmost and rightmost positioning of the long edges is calculated. The long edges are then fixed at the average of the two positions so as to achieve balance. The second step then positions the rest of the vertices and is fairly complicated. The original vertices are treated as sequences sandwiched between the already fixed dummy vertices. The goal of the algorithm in this step is to minimize the length of the edges between the vertices in the sequence currently being positioned and their neighbors in a previously positioned sequence. This method has a total running time of $O(\overline{m}(log\overline{m})^2)$, where $\overline{m}$ is the total number of edges in the k-layer graph, including those edges between dummy vertices.

A last method, given by [BK02], is fairly simple and runs in linear time. This method computes four different alignments for the vertices according to the median position of their neighbors. The four different alignments are the result of an upward/downward traversal through the layered graph and a leftmost/rightmost median conflict resolution strategy. Each of the four alignments picks out a different subset of edges that will be straight. With this, we can define blocks, which are simply every vertex along a straight edge path. A horizontal compaction step is then applied, the backbone of which is a longest path layering algorithm. Longest path layering simply assigns each block a coordinate that is recursively defined to be the maximum coordinate of the preceding blocks plus some minimum separation distance. The final step in this algorithm combines the biases of the four different alignments using the average of the left and right median coordinates of each vertex. This method compares quite well with that of [BJL01], although it does appear to use more area.

Extensions

The layered drawing technique, as described thus far, ignores the mental map visual aesthetic as well as compound graphs. A simple way to satisfy the mental map visual aesthetic, to some extent, is by using a sketch based technique. The sketch based technique uses a very different approach to layer assignment than usual. It directly uses the coordinates of vertices in an already computed drawing to compute the layering. This requires either extra processing to ensure a proper layered hierarchy, or accepting the inevitable crossings that will result if edges are permitted between vertices of the same layer.

Alternatively, one can use an incremental version of a layered drawing technique for the mental map problem. Incrementality, as the name suggests, is about drawing a graph incrementally as insert and delete operations modify the vertices and edges. A constraint based approach that accomplishes this is described in [BP90]. A concise four step summary of a constraint based approach is given in [And98]. Essentially, all vertices are constrained to their positions, save those that have been newly modified and their neighbors. The layout algorithm then attempts to create a new layout without violating any of the constraints.

To handle compound graphs, a layered drawing technique must first distinguish between adjacency and inclusion edges. It is important to do so, since many domain-specific formalisms such as DEVS, Statecharts, and control flow diagrams cannot otherwise be drawn meaningfully with a layered drawing technique. One of the first extensions to handle compound graphs is given in [SM91]. The layer assignment phase is modified to take hierarchy into account. Furthermore, edges between compound components, that is between vertices at different levels of the hierarchy, result in dummy edges with compound dummy vertices. The result is similar to that of long edges traversing multiple layers in the regular layered drawing technique. The crossing minimization phase is extended to keep vertices in the same compound component close together, minimize crossings between edges of vertices in different compound components but on the same layer, and minimize crossings between the edges of vertices on different layers. This is done using a priority based barycenter heuristic. Similarly, the horizontal placement phase is computed using a priority based barycenter heuristic, with special attention given to avoid overlapping rectangle boundaries.

An alternative method for compound graphs is given in [San96a]. In this method, the layer assignment phase has two different layer types. The new layer type consists of dummy vertices representing the top and bottom borders of a compound component, which will ultimately be drawn as the upper and bottom borders of a rectangle. This new layer type is very thin since only a rectangle line is drawn through it, whereas the normal vertex layers are much thicker. The border vertices are used to partition which compound component should go above, below, or on the same layer/s as another compound component. Dummy vertices between compound components are routed either outside compound components as in Sugiyama's method, or inside. The crossing minimization phase uses a barycenter heuristic modified so that on any given layer, vertices of a compound component are grouped together. Moreover, vertices of two compound components on multiple layers are not permitted to intertwine each other. These two modifications prevent boundary rectangles from overlapping each other. In [For02], Sander's crossing minimization method is improved by considering more than one layer at a time, thus avoiding unnecessary crossings. Note that this improvement addresses crossings inherent to the grouping a compound graph requires, not the general multi-layer crossing minimization problem. The final phase, horizontal placement, receives an interesting modification in the form of new dummy vertices along the left and right border of each compound component. These new dummy vertices are then required to form straight lines, thus yielding nicely spaced compound components with enough room to draw rectangular boundaries.

1.3.2 Force-directed

The force-directed class of graph drawing techniques are based on virtual physics models. The first algorithm to use this technique is attributed to [Ead84], which evolved from an earlier technique for VLSI layout called forced-directed placement [FR91]. The intuition behind this class of drawing technique is that since physical objects, such as molecules, tend to settle into highly uniform and balanced states, which are desirable visual aesthetics, then the simulation of vertices as molecules will yield a good layout. The uniformity of the molecules occurs due to the various forces acting upon them. Virtual forces are typically created wherever edges exist, although other possibilities exist, and are not necessarily natural in origin. Indeed natural forces, such as acceleration, tend to result in dynamic equilibria, whereas we desire a static equilibria. Continuing the physical system analogy, the molecules keep moving so long as a net force exists, thus the system becomes balanced only when the the sum of the energy imparted by the forces becomes minimal. With the proper choice of forces, this allows for the creation of force-directed heuristics with an explicit termination condition, that is, once the energy sum reaches some small threshold value.

Once a physical model is chosen a balanced configuration is typically found by discrete simulation. Many time steps are used to calculate the forces acting on each vertex and to update their positions. Doing such, one encounters the possibility of reaching a local minimum state, a state such that the forces are not minimal but that further iterations do not improve. When the simulation terminates, vertices are assigned directly assigned the coordinates of the simulated objects, and edges are typically drawn as straight line segments between them.

This class of algorithms works on both digraphs and regular graphs, and is most effective on sparse graph structures. This latter is in part due to the lack of any explicit crossing reduction strategy, thus a large number of edges are sure to yield an incomprehensible spiderweb. Area minimization is possible with the creation of a force, such as gravity, however this class of algorithms does not, in general, use space as efficiently as other graph drawing techniques.

The most basic implementation of the force-directed approach is called a spring embedder. Essentially, each vertex becomes a repulsive charge that repulses every other vertex, and each edge becomes a spring that pulls the connected vertices together. Many formula can be used for generating the repulsive and attractive forces, but regardless of the one used, the effects are similar. This is very straightforward to implement. However, we have not yet dealt with local minima nor oscillatory and rotational behaviour. A vertex is considered to oscillate if in the current iteration its movement vector is the opposite of the last iteration. Rotation is defined similarly, except the current movement vector is roughly perpendicular to the vector of the last iteration, and this should occur repeatedly in the same direction before being considered rotation.

The GEM [FLM94] force-directed heuristic handles local minima by adding random motion vectors at each simulation iteration. This randomness is controlled by a local temperature associated with each vertex. This temperature is decreased when oscillation and rotation are detected. Once the sum of the vertices temperatures reaches a low threshold value, the algorithm terminates. Another important aspect of this heuristic is that it uses barycenter gravity, that is, it creates an attractional force to the average position of all the vertices, leading to compact layouts. The authors give the estimated running time complexity as O(|V|^3), since they need an average of O(|V|) simulation iterations and the repulsive force calculations are O(|V|^2), although they break it down differently.

In [San96c], two equations are given for computing repulsive and attractive forces respectively:

$$F_{repulsive}(v,w) = -\lambda_{repulsive}\tfrac{\Delta(v,w)}{||\Delta(v,w)||^2}$$

$$F_{attractive}(v,w) = \lambda_{attractive}\Delta(v,w)||\Delta(v,w)||^2$$

The distance vector between the two nodes v and w is given by $\Delta(v,w)$ and the Euclidean distance by $||\Delta(v,w)||$. Thus repulsion is inversely proportional to distance, whereas attraction is directly proportional to a power of the distance. The lambdas allow parametrized control of each force's influence.

Also shown is the idea of computing an extra amount of gravity per vertex, proportional to the degree of a vertex. The following equations give the center position of all the vertices and the vertex specific gravitational force respectively:

$$B_{center} = \tfrac{1}{n}\sum_{i=1}^{n} position(v_i)$$

$$F_{gravity}(v) = \lambda_{gravity}(1 + degree(v))(B_{center} - position(v))$$

In this fashion, the seemingly more important vertices are more likely to be drawn near the center of the drawing. Finally, an extension to springs for the case of digraphs is given. In this extension, the springs are supplemented with directed magnetic fields. Various magnetic fields are possible, such as parallel, concentric, and orthogonal. The main advantage of these fields is that they enforce the direction of flow visual aesthetic.

A contribution in the form of advanced preprocessing is made in [GGK00]. The idea here is to find an initial placement of the vertices that is very close to the final configuration, thereby significantly reducing the time spent in the time expensive simulation loop. This is achieved by first finding a maximal independent set filtration with a size of three vertices. The second phase then places these three vertices according to the length of the shortest path between each pair of vertices, their graph distances. The third phase alternates between adding one vertex according to its graph distance to one of those three initial vertices and refinement using a force-directed model. Although overall time complexity is not given, the system is able to handle graphs on the order of 16000 vertices in under a minute. A different preprocessing technique is shown in [MR02]. The authors specifically target WWW visualization, which involves thousands of vertices and edges. Essentially, their preprocessor allocates the vertices uniformly randomly over a very large area, and then over successive iterations places nodes at the barycenter of their neighbors. This process stops once the edge lengths have reached some reasonable value, at which point the area will be reduced to something reasonable. For some graphs, the initial drawing yield by this preprocessor does not even require further refinement with the force-directed method.

In [FR91], they attack the problem of time complexity. Basically, calculating the repulsive force between each vertex and every other vertex, also known as the n-body problem, is very expensive. They decided that since the magnitude of repulsive forces vary inversely with distance, then vertices that are far apart aught to have distant and thus negligible force contributions simply dropped. They implement this by partitioning the vertices into a grid, and only considering repulsive contributions from vertices within adjacent grid boxes. The running time complexity is given as $O(|V| + |E|)$ assuming that the number of simulation iterations required is a constant 50, which seems somewhat doubtful. A more elaborate scheme for partitioning vertices, using graph clustering techniques, is given in [QE01].

A very different approach to the node overlap visual aesthetic is taken in [GN98]. Instead of creating a repulsive force to prevent overlap, they instead use a three part strategy. First they ignore overlapping by treating vertices as point objects and only calculate attractive forces along edges. In the second part, they calculate the actual boundary for each vertex as well as some additional room for edges. Then they compute a Voronoi diagram, move each vertex toward the centroid of its Voronoi cell, and expand as necessary until none of the vertices overlap. The Voronoi diagram approach is justified by comparing it to the far more primitive approach of simply scaling the graph, which results in a very area inefficient drawing. However, other algorithms specifically designed for dealing with node overlaps exist, such as that by [HL03]. This last algorithm is itself a force-directed method that makes use of orthogonal repulsive forces between overlapping vertices. Finally they route spline edges between each of the vertices, ensuring that no edge overlaps a vertex.

Finally, an experimental comparison of various force-directed strategies is given in [BHR96]. These include the GEM and [FR91] algorithms, as well as a simulated annealing approach that is discussed in section 1.3.5. The simulated annealing algorithm gave the best results although it required considerably more time than any of the other approaches. GEM was considered to be among the fastest in running time. Surprisingly, the algorithm from [FR91] was determined to be too slow for more than 60 vertex graphs. This would suggest the authors of this study did not implement the partitioning component of the algorithm.

1.3.3 Orthogonal

The class of orthogonal graph layout techniques includes a wide variety of methods. The following definition of orthogonal layout is detailed in [PT98]. Whereas force directed methods typically only use straight lines, orthogonal layouts also use poly-line edges with bends. The poly-line edges are drawn as continuous sequence of horizontal and vertical segments. In orthogonal drawings, all vertices and edges are assigned integer coordinates. In other words, the drawing is done on a rectilinear grid and edges correspond to grid paths. Orthogonal drawings are sometimes called rectilinear drawings or Manhattan drawings. Finally, there is an important distinction between pure orthogonal and quasi-orthogonal layout. The former has vertices of degree strictly less than or equal to four, whereas the latter does not.

Orthogonal layout techniques are commonly broken down into three phases: topology, shape, and metrics. No matter which approach is taken, orthogonal layout is particularly sensitive to the input graph structure. The following input graph properties are usually required by orthogonal layout: connected graph, biconnected graph, and maximum vertex degree four. The connected graph property is dealt with by considering each connected subgraph separately or by artificially connecting them with dummy edges. Since most orthogonalization algorithms require the construction of an st-graph, the biconnected property is also required. This is satisfied for general graphs by adding dummy edges as well. These dummy edges add overhead to the algorithms and are detrimental to several visual aesthetics. They are only removed when the graph is finally drawn. The GIOTTO algorithm, [TBB88], appears to be the only orthogonal algorithm for general graphs not requiring bi-connectivity. Finally, the requirement that vertices have maximum degree four, is the result of the fact that in 2D only four horizontal and vertical directions are possible. This limitation is often avoided by splitting a high degree vertex into several connected vertices that form a rectangular face. Unfortunately, the size of this rectangular face is unbounded. An approach that does deal with high degree non-planar graphs gracefully is described in [FK97].

In general, orthogonal drawing techniques produce high quality layouts because they optimize for many visual aesthetics. Depending on the specific implementation, they can optimize all or some of the following aesthetics: vertex overlap, edge crossings, number of bends, edge lengths, and area usage. Recently, the direction of flow aesthetic was also added to this repertoire [EKS03].

Alternative orthogonal layout techniques which do not use the topology, shape, and metric approach are known as draw-and-adjust [BBD00]. Comparisons between both the topology-shape-metric and draw-and-adjust approaches can be found in [BGL95, BGL$^+$97]. These comparisons were done on three and four alternative approaches to orthogonal layout, respectively. The results show that in terms of optimizing visual aesthetics the topology-shape-metric approach is superior, whereas the draw-and-adjust approach has much better time complexity.

Topology

The topology phase of graph layout optimizes both the vertex overlap and the edge crossing visual aesthetics. This is done by computing a planar embedding of the graph. The following explanation of planar graphs and their embeddings is quoted from [EKS03]:

> A graph is planar, if it has a drawing in the plane without edge crossings. Such a drawing divides the plane into regions, called faces. A planar embedding is a combinatorial description of the faces and contains for each face the sequence of edges contouring it. A planar embedding implicitly defines a cyclic ordering of the edges around a vertex.

The first linear-time planarity testing algorithm is attributed to Hopcroft and Tarjan [HT74]. This algorithm can be modified so that it also yields a planar embedding when the input graph is planar.

However, it has been described as complicated and subtle. Alternatively, a very recent and simple planarization algorithm is given in [BCPDB04]. This algorithm, also has linear time complexity. Moreover, an experimental comparison of running times with other planar embedding methods is given. The results of the comparison indicate that computing planar embeddings for graphs with hundreds of thousands of nodes, within seconds, is possible with several of these methods.

For non-planar graphs, the previous methods will halt upon discovering non-planarity. Thus edges must be removed until the graph becomes planar. Then the removed edges are re-inserted and the graph is augmented by adding dummy vertices wherever edge crossings occur. A description of a linear time algorithm to re-insert the edges, in a crossing minimizing fashion, can be found in [GMW01]. A different linear time algorithm for edge insertion and routing is given in [Eig03]. This last algorithm is generalized to work on mixed graphs, that is graphs containing directed and undirected edges.

These dummy vertices are treated as regular vertices by all phases of the algorithm, but are not actually drawn. For non-planar graphs, minimizing the number of dummy vertices required to planarize the graph results in edge crossing reduction. Dummy vertices can be minimized by removing the fewest number of edges possible to make the graph planar. More formally, given a graph G = (V, E), the objective of graph planarization is to find a minimum cardinality subset of edges E' ⊆ E such that the graph G' = (V, E \ E'), resulting from the removal of the edges in E' from G, is planar. This problem is also known as the maximum planar subgraph problem and is NP-hard [Cim95]. An algorithm to find the maximum planar subgraph, using branch and bound techniques, is given in [JM96].

A closely related, and more computationally tractable problem, is to find the maximal planar subgraph. This problem consists of finding a subgraph G' of G such that adding any edge present in G but not in G' results in a non-planar graph. An experimental comparison of the performance of five maximal planar subgraph algorithms is presented in [Cim95]. The heuristic yielding the highest quality in this experiment is the cycle-packing algorithm with $O(|V||E|^2)$ time complexity. An improved version of this algorithm, GRASP, is given in [RR97]. Although the worst case time complexity is $O(|E||V|^6)$, this bound is not reached in practice for several reasons given in the paper. Moreover, if speed is more important than quality, the maximum number of iterations can be bounded. The authors freely provide a Fortran implementation of this algorithm. In [Dji95], a linear time algorithm is given to solve this problem. However no quality comparisons are made with respect to the optimal solution, nor other heuristic algorithms.

Planar embeddings are not unique. In fact, according to [BBD00], there exist a factorial number of them. From an edge crossing point of view, every planar embedding is equivalent. However, the minimum number of bend numbers will be influenced by the choice of embedding. Thus the distinction between the topology and shape phases becomes blurred. In [DL98], an algorithm that searches through planar embeddings to find an orthogonalization with the minimum number of bends is presented. However, the running time of this algorithm is very poor, $O(6^{n_4}n^4logn)$, where $n_4$represents the number of vertices with degree 4. In [BBD00], a branch and bound algorithm is given to find the optimal number of bends by considering many possible embeddings. This algorithm is only applicable to small graphs. Finally, a linear time algorithm is given in [GM04]. Rather than search through the entire space of potential embeddings, the search is restricted to only those embeddings with qualities that lend themselves to superior layouts.

Shape

The shape phase maps the planar embedding given in the topology phase onto a grid. This grid embedding must have the same combinatorial description of faces as is present in the input planar embedding. The result of this phase is an orthogonal representation. Hence this phase is also called orthogonalization. An orthogonal representation is completely dimensionless. The objective of this phase is to construct the grid embedding while minimizing edge bends.

A relatively simple and linear time complexity approach to this problem is given in [TBB88]. The method is called bend-stretch or fast orthogonalization. This involves constructing an st-graph and then a visibility representation, which consists of vertices drawn with large widths and straight vertical edges. An alternative method for constructing visibility representations without at st-graph is given in [Boy05]. A fast expansion step then chooses an intersection of a vertex and an edge as the position of a unit sized vertex and edges become both vertical and horizontal as a result. Finally, bend-stretching transformations are applied, which are simple rules that match patterns in the embedding of the previous step, in order to reduce the number of bends.

A different and very elegant approach to generating an orthogonal representation is detailed in [Tam87]. In the experimental comparisons of orthogonal layout algorithms, [BGL95, BGL+97], this approach is known as GIOTTO. The approach consists of the transformation of the planar representation into a network flow problem. For example, the flows correspond to the number of bends in the grid embedding. The computation of the minimum cost flow in the network yields the desired orthogonal representation with the minimum number of bends. Recall that the minimum bend number is minimum only for the input planar representation, of which many are possible. The minimum cost network flow problem is solved using an augmentation algorithm in as little as $O(n^2 logn)$ time, due to the specific nature of the problem. An improved version of this algorithm, with $O(n^{7/4}\sqrt{logn})$ time complexity, is given in [GT97].

The Kadinsky algorithm is very similar to the previous approach. They both use network flow techniques, with some important modifications, to find the minimum number of bends. The difference lies in the fact that this approach specifically targets graphs with high degree vertices. The original version treats all vertices as squares of equal size and allows multiple edges per side [FK96]. A later version allows for vertices of high degree to expand in size proportional to their degree [FK97]. This results, at least for some graphs, in a better layout in terms of area usage, edge bends, and edge crossings. A version of this algorithm that can handle vertices of a prescribed size is given in [BDPP99]. This is quite possibly the only algorithm that explicitly deals with vertices of a specific size, something that is particularly useful if the vertices are represented by non-scalable picture icons. The running time for all these algorithms is bounded by the time required to solve the minimum cost network flow problem. In [Eig03], it is revealed that the original version uses an incorrect algorithm. Thus not only will the algorithm fail to yield the minimum cost network flow for some input graphs, it may even fail to yield a feasible solution. A fix for the algorithm is given, however it is not guaranteed to yield the optimal solution to the network flow problem. The run time is complexity is $O(n^{7/4}\sqrt{logn})$.

The basic implementation of these shape phase methods treat all vertices as point objects, with a single edge port in each cardinal direction. In [BNT86], it is shown how a vertex can be expanded into a rectangular skeleton with many potential connection points. This is particularly advantageous in multi-graphs where two vertices have multiple edges between them. By increasing the number of connection ports on a single face of the vertex, all these edges can be drawn as straight lines, thus producing a more compact layout with fewer bends.

Metric

The metric or compaction phase, takes as input a graph embedding and produces the final layout on a rectilinear grid. In [BNT86], this phase is divided into two steps. The first involves the decomposition of each face in the embedding, by matching some basic patterns, into a rectangle. This requires linear time. Thereafter, each rectangle segment is assigned an integer length, using integer linear programming. Interestingly, horizontal and vertical compaction can be solved for independently.

Moreover, the time complexity is polynomial due to the fact that the total unimodularity property holds for the ILP input matrices.

An alternative second step is outlined in [TBB88]. Once again, horizontal and vertical compaction is considered separately. In essence, a network flow technique is applied, where each unit of flow corresponds to one unit of length and each node corresponds to a rectangle (face). Conservation of flow at each node forces opposite sides of a rectangle to have the same lengths, while minimization of the flow cost results in total edge lengths of minimum size. This method has a time complexity of $O(n^2)$.

A variety of different compaction algorithms are experimentally compared in [KKM01]. These algorithms include longest-path compaction, network flow, turn-regularity, and ILP. Note that the first step of decomposition mentioned above is not used by all these algorithms.

Draw-and-adjust

There are two known draw-and-adjust algorithms for orthogonal layout. The first of these is the column approach [BK98]. An s-t ordering is first computed, where s is the source and t is the sink, using depth first search. Then the vertices are consecutively embedded in a grid. Initially two mutually connected vertices are embedded on row 1, with their mutual connection routed along row 0. Each vertex then reserves three columns in the rows ≥ 2, through which they may route their remaining edges. Additional columns are added to the embedding as needed. Also, the choice of column to route edges is made such that bends are minimized. From the rows and columns of each vertex and edge bend, final coordinates are easily derived. This approach works on any biconnected graph and has time complexity $O(|V| + |E|)$.

The second approach is the pairs algorithm [PT97, PT98]. This algorithm is similar to the first in many respects, the major difference being the vertex pairing. An s-t ordering is used to classify each vertex according to its indegree and outdegree. For example, a 1-2 vertex has one incoming edge and two outgoing edges in the s-t ordering. The most important vertex classifications are 2-2, 1-2, and 1-3 since the algorithm can always pair such vertices on a row or column. This is important because two vertices that have been paired to the same row/column reduce the total number of rows/columns and result in fewer edge bends and crossings. The time complexity of this approach is $O((|V| + |E|) \log(|V| + |E|))$. It accepts any biconnected graph where the maximum degree of a vertex is four. Judging from the "typical" layouts of graphs shown in [BGL$^+$97], this method is far superior to the column approach in terms of image quality.

1.3.4 Linear Constraints

Linear constraints provide a declarative approach to layout, but cannot be used without a fairly complex solver. The most common technique for solving linear constraints is Dantzig's simplex algorithm from the 1940's. However, standard implementations of it are not satisfactory for layout purposes [BBS01]. This is in part due to the fact that standard implementations solve one problem and terminate. Layout tends to involve interaction with the user and thus requires a solver that can rapidly solve very similar problems. A similar problem is one where a variable's value is modified or a variable/constraint is added or removed. For example, if the variable representing the horizontal coordinate of an object is increased by one unit by a user, the solver should make use of the existing solution with just the modification of the changed variable. A second problem with constraints for layout purposes is that they are often cyclical in nature, thus making a number of efficient solving techniques unusable. A third problem involves satisfying the mental-map aesthetic during interactive editing of a graph layout. In other words, objects should only move if it is absolutely necessary to satisfy the constraints. A final problem is that while modifying layout, the user may create a conflict whereby the constraints cannot be satisfied by the new change. In this case the conflict needs to be resolved without throwing an error. To overcome these problems, incremental linear constraint solvers were developed, such as the Cassowary and QOCA solvers whose source code is freely available [BBS01, BMSX97].

The primary use for linear constraints lies in the layout of windows for user-interfaces. Many aspects of such are easily represented with linear equality and linear inequality constraints. Inequalities in particular are useful for specifying: insideness, to-above-of, to-below-of, to-left-of, to-right-of, and overlapping constraints. For example, one could give a set of linear constraints such as $X = coordinate$, $X \leq objectA.x$, and $objectA.x + objectA.w \leq objectB.x$, where x is the leftmost coordinate and w is the width, to specify that objectA is to the right of the window boundary X, and the the right side of objectA is left of objectB. The resulting interactive layout behaviour is that attempting move objectA past the window boundary on the left will cause objectA to be frozen at the boundary, whereas moving object to the right of objectB will cause objectB to move right as well.

Although linear constraint solving techniques were primarily developed for the layout of windows in user-interfaces, they have also been applied to graph layout problems. In [CMP99], a Penguins system is developed that uses linear constraints and the QOCA solver for computing the layout of binary trees, state transition diagrams, and mathematical equations. The linear constraints are generated using a grammatical specification of the visual language in question. This means that each graph type, or visual language, receives a specifically designed layout that is best suited to it. Similarly, the GenGED tool uses visual grammars to specify diagram editors or formalisms [Bar98]. The tool also makes it possible to visually add linear constraints to assemble icons that will represent vertices, as well as to describe the layout that edges will impose between various vertices. However, the linear constraints are solved using Parcon, which has very limited documentation and is only available as a binary with limited platform support. Finally, the tool DiaGen uses a textual grammatical specification of the visual language, and optionally that of the layout constraints [MK99]. DiaGen uses QOCA for solving linear constraints, however it does not necessarily use linear constraints for all the visual languages. It does however use them for trees and NSD diagrams. For details on the use of linear constraints in NSD diagrams, see [VM94], where the implementation is done in a tool that is a precursor to DiaGen.

Disjunctive and non-linear constraints occur often in graph layout however. Consider a non-overlapping constraint between vertices. This can only be achieved by disjunctive constraints specifying that either vertex A is to the left of vertex B or vertex A is to the right of vertex B. Attempting to do this with normal conjunctive constraints simply yields conflicting constraints. Thus one needs a solver that can handle such constraints, such as Parcon, whose source is non-freely available, or one needs to implement the modifications to the existing incremental linear constraint solvers, as is proposed in [HMM02, MMSB01]. Disjunctive constraints have a potentially significant impact on running time, as they increase the number of constraints the solver must work on considerably.

Non-linear constraints are not so easily dealt with. A non-linear constraints has the form $x * y = z$. A simple example where this occurs, namely when trying to tightly fit a box around n-characters of text of a certain width, is given in [VM94]. The simplification of constraints in order to avoid non-linearity inevitably results in a poorer layout. Thus for full flexibility, the use of different solver is required, one that can handle non-linear constraints, non-overlap, and even the Euclidean geometric constraints of: perpendicularity, parallelism, and distance equality. Such a solver has been proposed in [Hos01], unfortunately the results were less than satisfactory, in particular the running time was quite poor even for very small graph sizes.

1.3.5 Expensive Methods

This subsection discusses drawing techniques with very high computation costs. This does not mean they are unworthy of discussion; for certain applications these techniques are ideal. For example, consider the MSN Line Drive Maps, which convert road directions from point A to point B into a line map such as one might draw with pen and paper for a friend who does not know your neighborhood [AS01]. The map simplification and labeling of roads is done entirely with a simulated annealing process. In spite of this, the system is quite fast, although it does consider a very restricted graph structure.

Simulated annealing

Simulated annealing is a flexible optimization method, suited for large-scale combinatorial optimization problems, such as the traveling salesman problem, and problems concerning the design and layout of VLSI [DH96]. The intuition behind this method is that in a physical system, cooling a liquid slowly results in a totally ordered configuration, a crystal formation. On the other hand, cooling a system rapidly results in amorphous structures that have higher energy, representing an undesirable local minima. Thus the major difference between simulated annealing and standard iterative improvement methods is that simulated annealing deliberately allows the current and temporary solution to worsen, with a probability proportional to the temperature. This seemingly unreasonable characteristic allows the method to leave an undesirable local minima solution and gives it a much better chance of finding a globally optimum solution.

Applied to graph layout, simulated annealing works as follows. First an initial configuration of the vertices is chosen, such as by random coordinate assignment. Second, the configuration is changed according to some rule, again one possibility is to randomly permute the vertex coordinate. A global evaluation function then determines if this new configuration satisfies the visual aesthetics better than the old. With probability α the better configuration is chosen, and with probability $1 - \alpha$ the other configuration is chosen, where $0 \leq \alpha \leq 1$ and α is chosen such that it decreases with the temperature. This choice of configuration is what allows the method to avoid getting stuck in local minima solutions. Thirdly, the temperature is decreased after the second step repeated some number of times. Finally, the algorithm terminates when no further progress is made or a time limit is reached.

The global evaluation function in [DH96] explicitly optimizes for the following visual aesthetics: uniform node distribution, area, edge lengths, edge crossings, and edge-vertex crossings. However, some of these aesthetics are considered only for the last few fine-tuning iterations. The authors claim a running time of $O(|V|^2|E|)$, and point out that simulated annealing is in general an expensive algorithm. Indeed an experimental comparison of various force-directed methods, including this simulated annealing method, reveals it to be the slowest method by far [BHR96]. The performance of simulated annealing can be drastically improved by using a preprocessing phase to create an initial layout. Useful preprocessing methods include a force-directed method in [San96c] and a planar or nearly planar embedding in [HS94].

Genetic algorithms

Genetic algorithms are very similar to the previously discussed simulated annealing algorithm. In both cases, randomness is used to generate new configurations and a function is used to evaluate the quality of the resulting layout. However, the intuition is quite different, rather than crystallizing liquids, this method simulates biological evolution. Essentially this comprises of searching for a solution while using natural selection to weed out poor configurations.

A genetic algorithm for graph layout, according to [Mas92], has the following pseudo-code:

1. An initial graph configuration is chosen, possibly randomly.
2. The layout is allowed to evolve over a large number of iterations, also known as generations. Evolution is simulated by crossovers and mutations. In natural genetics, a crossover occurs when two genes are randomly selected, cut at two points, and then a portion of each gene is exchanged at the cut points. The graph layout equivalent is performed by placing vertices in a 2D grid, and randomly swapping whatever vertices happen to be in some portion of the grid of one layout with that of another layout. Mutations, in natural genetics are random changes in a letter of a gene sequence, and correspond to a shift in a vertex position in the graph layout problem.
3. A fitness function determines if the crossovers and mutations have improved the visual aesthetics of the graph. Layout configurations with very poor fitness scores are dropped, whereas promising layouts are bred to create new generations of graph layouts.
4. Termination occurs once the population of candidate layouts reaches a predetermined number. The layout with the highest fitness score is returned.

Key issues with this technique lie with selecting a good fitness function and determining how many generations are necessary. The latter problem can be resolved by simply halting the population growth after some amount of time. The selection of a fitness function is not so easily resolved however. A fitness function might consider the following visual aesthetics: the number of edges directed upward, the number of edges shorter than a given constant, the number of crossing edges, and the number of edges forming crossings with other edges at angles smaller than a given constant. An alternative fitness function of much greater complexity is given in [KMS94]. In [Mas94], the author claims that even given the fitness function, finding good weights for each aspect of the function is very difficult, with small changes leading to unpredictable results. This issue is resolved by creating a system that allows the user to provide good and bad layout examples, from which the weights are automatically tuned, until each good example gets a score greater than the bad layout example. The advantage of this system is that it is very intuitive to work with and allows the end user to tune the layout algorithm for specific graph layout requirements.

The computational complexity of genetic algorithms is high. This is symptomatic of all search based techniques, including the previously discussed simulated annealing. This algorithm should be reserved for use only when interactivity is not required, such as VLSI design, or perhaps as a post-process fine-tuning phase.

Rule-based

A rule based approach to layout is proposed in [KMS94]. The intuition behind this technique is that humans construct diagram by applying a finite number of rules to incrementally build the layout, thus an automatic method would encode these rules as heuristics. However this method has serious shortcomings. The search based strategy of applying rules does not guarantee that a valid layout will be found and may require excessive amounts of backtracking from a current invalid layout to a previous valid layout. Layout aesthetics that are global in nature, such as edge crossings, cannot be captured at all. Finally, rules optimizing different aesthetics can conflict. The authors conclude that this technique is not worth pursuing further. A somewhat similar approach can be found in [SYTI92]. The authors improve the system by making it possible to extract layout rules from example layouts with the use of fuzzy logic techniques. Details of the implementation and the running time of their algorithm are sketchy, however one can conclude the system is unsuitable for interactive use.

1.3.6 Other Techniques

This subsection describes graph drawing techniques that are less commonly used or that complement other graph drawing techniques. The former category includes: 3D, circular, competitive learning, multi-dimensional, and graph grammar based layout. The latter category includes large graph visualization methods and techniques for edge routing.

3D layout

3D graph layout is motivated by the fact that there are limits to the amount of information a user can perceive from a given 2D area. Moreover, most of the previously discussed layout algorithms have trivial generalizations to the third dimension. In [Dwy01], the question of whether or not 3D is truly useful in graph layout is studied. The underlying algorithm used is force-directed. Interestingly, the authors cite a study performed on human subjects that showed that even though we perceive 3D drawings as 2D projections, they nonetheless allow for 3 times more information than a regular 2D drawing. This indicates that good comprehension of a graphs structure is possible for larger graphs when drawn in 3D versus 2D.

The results of the study on the effectiveness of 3D in [Dwy01] are not overwhelmingly strong however. Moreover, a control group study using 2D drawings is not used for comparison. In general, 3D techniques should be limited to situations where the vertices are point objects. This makes it possible to understand the structure of complex graphs very easily. When the vertices are large and contain important information in labels, visibility issues significantly degrade the value of this approach.

Circular

Circular layouts are a fast and fairly space efficient way of representing rooted tree graph structures. They have been successfully applied to at least the following domains: network topologies, multimedia documents databases, and virtual reality scene descriptions. Detailed pseudo-code for circular layout algorithms are available in [MH98, YYL03]. The basic idea behind these algorithms is to pick a root vertex, draw the neighbors of the root in a circle around it, and recursively treat the neighbors of the root much like the root vertex. The method of [YYL03] achieves better area efficiency by directing neighboring vertices outward from the root vertex, allowing root and neighbor to be drawn closer together. All these layout methods have linear time complexity.

Competitive learning

Competitive learning using neural networks is applied to graph layout for the first time in [Mey98]. The intuition for this technique comes from the fact that in biological systems the spatial or geometrical arrangement of the cells is important. In other words, it is not just the strength of the neural excitation that convey importance, but also the location. For example, when discussing mammalian brains, the concept of topographic maps often arises, such as in the tonotopic map. The tonotopic map from the ear to the auditory cortex works such that spatially close cells correspond to hearing similar frequencies. Thus a layout algorithm based on neural networks provides us with vertices positioned at their graph theoretic distances, as well as uniformly distributed within a specified area.

In [Mey98], a computationally friendly version of neural networks known as self-organizing maps is used. However unlike in traditional neural network applications, the network is not being trained to do some computation, instead the topology of the trained network is itself the desired solution. The desired network topology is termed an inverted self-organizing map, or ISOM. Essentially, each node in the network is modelled as a vector of weights, corresponding to a 2D grid embedding. The weights are then updated using a stimulus comprising of a random vector. The weights of the node nearest to the stimulus are updated, as well as the adjacent nodes. This means each iteration of the ISOM algorithm is restricted to a small subgraph and that the update procedure is itself not computationally demanding. Although it is difficult to bound the required number of iterations, necessary for complexity analysis, this algorithm has been experimentally verified to have nearly linear time complexity [Mey98, ALPW04].

It is unfortunate that although ISOM is impressively fast, it often yields poor layout results when used on its own. The primary difficulty lies in the fact that it does not explicitly optimize the node overlap aesthetic, resulting in an unreadable layout for some graphs, particularly larger ones. Therefore, ISOM must be hybridized to generate high quality layouts. In [ALPW04], ISOM is used as a preprocess with a directed-force algorithm, yielding a significant 40% time savings. A hybridization of ISOM and simulated annealing or genetic algorithms would likely yield very interesting results, however this does not appear to have been attempted as of yet.

Multi-dimensional

A multi-dimensional graph layout method is proposed in [YKC02]. The intuition here is that vertices should be drawn according to their actual graph distances, but in a highly scalable fashion using algebraic multigrids. Similarly to the previously seen force-directed methods, they find an energy function that yields the desired vertex positions when minimized. This energy function has the property of being both simple and smooth. In order to minimize this energy, they implement a very fast algorithm based on an algebraic multigrid technique. Essentially, this technique involves creating a hierarchy of graphs from the original one, with succeeding levels being increasingly coarse. A coarser level will merge vertices with a small graph distance into a single vertex. In [YKC02], the difference in the number of nodes between levels is approximately a factor of two. This proceeds until the graph is quite small, comprised of approximately 100 vertices. At this point they work backward to the original graph by progressively refining the solution. They do this by computing the eigenprojection of the coarsest level, and use the results in the next coarsest level, until the an eigenprojection is computed for the original graph.

The beauty of this approach is the sheer speed. It can handle 10^5 vertices in a few seconds and10^6 vertices in tens of seconds. This is two orders of magnitude faster than any known algorithm as of at least the year 2000. The clustering of vertices is very similar to the neural network approach. Moreover, this technique easily scales to projections in 3D and can provide alternative projections of the same graph. For example, it is possible to emphasize clustering in one graph layout projection while emphasizing grid structure in another. Unfortunately, it does not deal with the vertex overlap visual aesthetic at all. However this is not important if one only seeks the overall structure of a graph. If it is important, such as when a user seeks to zoom a portion of the graph, alternative layout methods must be applied. Since the overall graph structure has already been computed, a force-directed method requires far less time to improve the layout of the zoomed subgraph.

Graph grammars

Graph grammars are composed of rules and each rule has a left and right hand side. Each side of a rule is graph. A graph re-writing algorithm matches the left hand side of a rule with a host graph. In this case, the host graph is the one requiring a layout. If the re-writing algorithm finds a match for the left hand side of the rule then the right hand side of the rule is applied. The application of the right hand rule side results in a new layout configuration for the corresponding subgraph in the host graph. Layout for an entire graph is achieved by combining the layouts of subgraphs according to the parse tree of the grammar. The graph layout rules may also be specified in a visual fashion, such as in [ZZO01].

Thus graph grammar based layout has the advantage of being intuitive to specify as a series of visual rules. Moreover a graph grammar specification of a visual language can be augmented to also handle layout, as in [ZZO01], thus simplifying the development of domain-specific visual formalisms. According to [Mey98], fast parsing algorithms with polynomial time complexity, are available for context-free layout graph grammars. However, they impose severe limitations on the graph structures that can be handled. For example: grid, planar, and all not strictly hierarchical graphs are excluded from use by such parsing algorithms.

Edge routing

Edge routing post-processes can improve the readability of a layout enormously. Many of the previously discussed layout techniques use simple straight-lines to connect vertices. In [LE98], a high-level approach to the edge routing problem is discussed. The first step consists of ensuring that sufficient space exists between vertices to allow for the edge routing. They restate the details of the force-scan algorithm, which is itself far superior to simply scaling a graph layout. Alternative methods already mentioned are the Voronoi diagram based method by [GN98], and the force-transfer algorithm of [HL03] which is reputed to be the most efficient. The second step works to optimize the following visual aesthetics: number of edge crossings, number of edge bends, and edge length. Naturally, these are conflicting criteria, thus they must be prioritized. Therefore, they apply a shortest-path edge routing algorithm, and then create a dummy vertex for each bend in the resulting edge. Finally, they re-apply the first step of the algorithm, except that rather than dealing with node overlaps, it is now the overlaps caused by the dummy vertices of the edges that are dealt with.

Graph browsing

Browsing a graph structure can, in and of itself, require special graph drawing techniques. Consider any large graph. A detailed view of its vertices, edges, and associated labels is only possible for a very small portion of it. There do exist some simple linear methods for dealing with this. One is to shrink the size of the drawing shown in the main window of the graph viewing tool, and expand the size of selected nodes in a secondary window. The inverse and perhaps more common technique is to provide a small overview window, where the entire graph is shrunk. The overview shows the structure of the graph without any details and the user need only click on the overview to bring up a detailed view in the main window.

Fisheye views of a graph show both a detailed view of the area the user is focusing on and the rest of the graph structure, in the same window. This is done by distorting the picture in a non-uniform fashion. An overview of fisheye view techniques is given in [San96c]. Sander characterizes fisheye techniques as distorting, filtering, and logical. The distorting fisheyes magnify a focus area and distort the rest. Some even use multiple focus areas, such as in [GKN04]. Filtering fisheye views filter out information from distant parts of a graph structure. In other words, they remove vertices and edges, proportional to their distance from the focus area or areas, while attempting to maintain the overall structure of the graph. This makes the distorting fisheye view much more readable. Another fisheye technique is the logical fisheye, where instead of using the coordinates of an existing layout, a new layout is computed according to graph distances from the focus node. The advantage of this is that it is highly revealing of the graph structure. Moreover, the computation of a new layout after distortion is faster since it is done after filtering away distant vertices and edges. It also uses space more efficiently. Unfortunately, the recalculation of layout with each change of focus is very bad for a user's mental-map. This is due to the fact that a drastically different layout may be shown for only a slight change in focus.

2 Graph Drawing Technique Implementations

The usefulness of visual modeling is dependent on how elements of a model are visually arranged. Hence, any tool supporting visual modeling should provide some mechanisms to reduce the burden of drawing models with good layouts. One such tool is AToM^3, into which all automatic layout techniques in this thesis are implemented. This tool is described in section 2.1.

Prior to the implementation of graph drawing techniques in AToM^3, an attempt was made to leverage the capabilities of existing graph drawing tools. This was done by exporting models as bare-bones graph descriptions, and for one drawing tool in particular (yED), re-importing the graph after layout was performed. As section 2.2 explains, the attempt proved largely unsuccessful in meeting model layout needs.

A number of graph drawing techniques were then chosen and implemented in the AToM^3 tool. Since the Python programming language is most suitable for rapid prototyping and AToM^3 is itself coded in it, all implementation was done in this language. However, to avoid the creation of drawing technique implementations that are useable only in AToM^3, as well as provide the possibility of easily plugging in other algorithms potentially written in a more efficient programming language, an abstraction layer was designed. As described in section 2.3, all layout algorithms were completely isolated from the internal data structures of the AToM^3 tool and forced to work through the abstraction layer's own graph drawing optimized data-structure.

The most complex automatic layout algorithm implemented is the layered drawing technique. As previously described in section 1.3, it optimizes quite a few visual aesthetics and is applicable to a wide range of graph structures with good results. The design, extensive pseudo-code, complexity analysis, and case-studies are shown in section 1.3.1.

The spring-embedder layout algorithm, which uses multiple physical model simulations, is less complicated and faster than the layered layout engine. A key benefit to this layout is its iterative nature, thus making modifications to a model and then re-applying the spring-embedder to it usually results in only minor layout changes. Pseudo-code, complexity analysis, and a case-study can all be found in section 2.5.

The force-transfer layout algorithm, uses a simple physical simulation to eliminate overlapping graph vertices. This algorithm can be combined with random layout in some situations for a complete layout solution. More generally, it is a useful and minimally intrusive layout aid when automatically applied during model creation/modification. It is used in exactly this fashion in chapter 3. Section 2.6 describes the design, pseudo-code, complexity, and a case-study of this algorithm.

The simplest algorithms implemented are tree-like and circle. Both of these are very fast, although correspondingly modest in layout quality. As the name may suggest, the tree-like algorithm is quite effective at drawing tree structures, although otherwise weak. The circle layout is not generally interesting in its own right, however it proves to be highly effective as a pre-processor for spring-embedder layout. Section 2.7 provides the pseudo-code, analysis, and case-studies for both of these algorithms.

Thus far, five different graph drawing techniques have been implemented and analyzed. To determine their actual time-performance as well as validate the computation complexity analysis', each algorithm is benchmarked on a series of randomly generated graphs in section 2.8. This section also discusses how performance may be improved, in particular, by simply re-implemented the algorithms in a higher-efficiency language.

Finally, a different approach to automatic layout is taken in the form of linear constraints. Linear constraint based layout has the advantage of being straightforward to customize by formalism developers to meet formalism-specific layout requirements. Unfortunately this layout approach has limited expressivity and does not hybridize (easily) with other layout techniques. The design and a discussion of how linear constraints can be woven into the modeling process, with the use of a new Pac-Man formalism as an example, is described in section 2.9.

2.1 AToM^3

AToM^3 is an acronym for A Tool for Multi-formalism Meta-Modeling [dLV02a]. It is described in a broader context in section 3.1.1. The following, shorter version, describes only what the tool actually does and motivates the need for automatic layout algorithms.

AToM^3 is used for for modeling, meta-modeling, and transforming models with graph grammars. Moreover, it can be extended to handle the simulation and/or generation of code from models. Models are not merely drawings, they are constrained by domain-specific rules encoded into a formalism specification. The formalism is automatically generated from a meta-model, which is simply a model that generates the specification for the creation of a specific kind of model. Formalisms are specified as meta-models in AToM^3 since it is faster and less error prone than manually hard-coding all the rules. For example, an Entity-Relationship model can specify a Class Diagram model which can in turn specify virtually any other kind of model, such as Statecharts. In this last example, both the Entity-Relationship and Class Diagram models are considered meta-models. To avoid confusion, Entity-Relationship is not termed a meta-meta-model.

The multi-formalism aspect of AToM^3 refers to the ability of AToM^3 to transform a model in one formalism to another formalism using visual graph grammars[1]. Graph grammar rules are themselves models, typically containing elements of both the source formalism and the target formalism of the transformation, as well as a generic "glue" formalism. An interesting example of this is of a model in the domain-specific Traffic formalism being transformed to the Petri-net formalism and then transformed again to a reachability graph to determine if deadlock is inevitable. Note that once the graph grammars are created, any traffic model can be automatically transformed.

At the modeling level, AToM^3 is very similar to other tools of this sort. Each formalism provides buttons for drawing entities on a canvas. The entities have visual icons defined in the formalism. These icons can contain any number of textual attributes, such as names and tokens, and can be changed dynamically by a simulator. Since a formalism is just a meta-model, creating a meta-model means setting an attribute that is the icon to be. A special drawing editor, resembling a paint program, exists for this purpose. Indeed the only difference with a regular paint program is that it also includes dynamic textual attributes and connection ports, to which edges are connected to. Modeling links between entities on the canvas is done by simply clicking on the two entities. Usually tools force you to place a link on the canvas and then hook it up. Oftentimes, only one possible type of link is possible between the types of the entities, such as an arrow, and AToM^3 automatically chooses the correct type of link to establish. Otherwise, the user is asked to choose among only those link types possible between the two entities.

[1] AToM^3 is not limited to transforming one formalism to another. A model in N-formalisms may be transformed to one in M-formalisms, so long as N and M are positive non-zero integers.

AToM^3 has a variety of formalisms already implemented in it. The distribution of AToM^3 found at `moncs.cs.mcgill.ca/people/denis/`, currently contains the following formalisms: Entity-Relationship, Class Diagrams, DCharts (similar to statecharts), Deterministic Finite state Automata, Non-Deterministic Finite state Automata, DEVS, GPSS, Petri-nets (and two variants thereof), Causal Block Diagrams, and Sequence Diagrams.

During the course of this thesis, a number of features were added to AToM^3 to make modeling easier. These include a grid, making alignment of model entities easy by snapping them in place. Also, visible links between entities, such as arrows, were made more user friendly. They automatically choose the connection port on the model entity the link to that is closest to the arrow. Moreover, the arrows can be automatically be drawn as either a straight line or an arbitrarily curved spline between two entities. Another improvement lies with the visual graph grammars. Since a typical grammar consists of many rules, each with a model for the left and right hand sides, the advantage of easy documentation is lost to the needle in a haystack effect. Hence an automatic LaTeXdocumentation generator for graph grammars was added. Finally, it is possible to scale or stretch the coordinates of all the entities in a model at once. Unfortunately, all these additions still leave the burden of layout on the shoulders of the modeler. Even if one does not consider the layout needs of a model created from scratch by a human modeler, the models generated by transformation from one formalism to another have no layout information at all. Hence the need for automatic drawing techniques for graphs that can deal with these models.

Summarizing AToM^3, it is a tool that provides a fairly standard modeling interface. Differentiating AToM^3 from other tools are its meta-modeling and model-transforming capabilities. The former makes it easy to create new formalisms, formalisms that restrict the modeling process to valid rather than arbitrary models. The latter allows for domain-specific modeling without sacrificing the simulation or analytical capabilities of more generic formalisms. AToM^3 is also highly extensible, making the addition of a button to a formalism that does a graph traversal of a model to either simulate it or generate code from it very easy.

2.2 Graph exports and imports

Since several free existing graph layout tools exist, the ability to export graphs from AToM^3 was added. This means that model entities become simple point vertices, except for the GML format, which allows specification of height and width. The vertices are permitted just one string label, versus the multiple attributes possible in AToM^3 icons. Links between model entities are simply exported as an identifier to the end point entities of a link and an arbitrary label. Hence the transformation from model to simple graph structure loses a considerable amount of information. However once exported to GML, GXL, or DOT format the graph layout tools yED, JGraphPad, or GraphViz, respectively, can be used. These tools are available freely available at: `www.yworks.com/en/products_yed_about.htm`, `freshmeat.net/projects/jgraphpad/`, and `www.graphviz.org/`. Provided the labels are descriptive enough, the resulting layout can make understanding properties of the model the graph represents quite obvious.

A simple export to obtain automatic layout is far too limiting however. Therefore, a mechanism to export to GML, wait for a layout engine to modify the GML file, and then re-import the GML file was implemented. This too is problematic however. On the one hand, it is quite a tedious process for the user to follow, especially since yED requires manual user input through a graphical interface. On the other, the notion of ports in different tools is different, hence coordinates for straight arrows in one tool are quite crooked in another. Hence simply off-loading graph layout to other freely available tools is an inadequate measure.

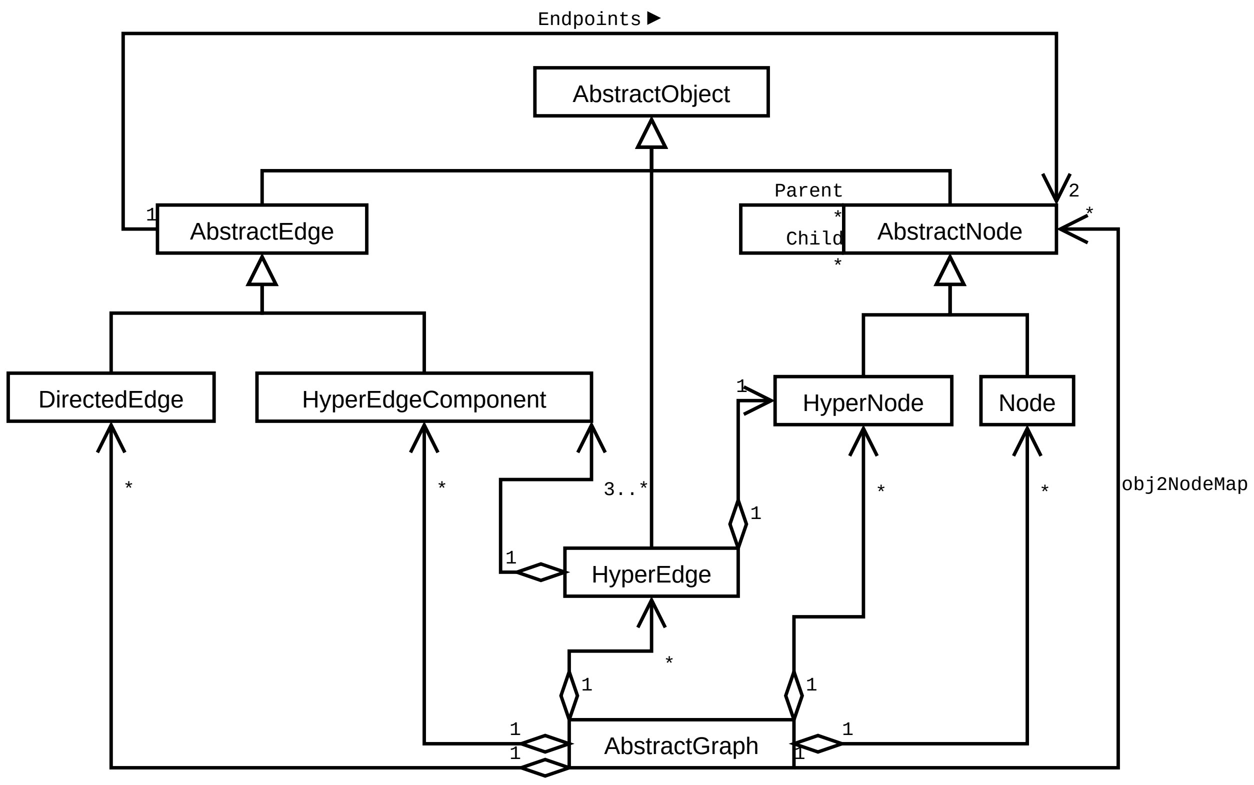


Figure 2.1: Abstraction layer class diagram

2.3 Abstraction layer design

An abstraction layer was built between the AToM^3 tool and the automatic layout methods. It was deliberately constructed so that automatic layout implementations are denied all access to attributes and methods found in the AToM^3 tool. This has two important benefits. One, it reduces the implementation complexity of the automatic layout methods considerably. Two, it enables the automatic layout methods to be used as a portable graph layout library. In particular, the next generation version of AToM^3 will re-use this library.

The abstraction layer itself communicates with both AToM^3 and the automatic layout methods. At the most basic level, it extracts the position and size of vertices, and the coordinates of edges from AToM^3. The automatic layout method is run, and the new positions, sizes, and coordinates are sent back to the original AToM^3 entities. A few other possibilities exist. One is that a layout might request that edges be drawn straight or curved, with no coordinate information provided. Another possibility is that a layout might allow the user to pick a certain vertex, such as to select a root when more than one are possible. These requests are relayed to AToM^3 which has the low level functionality to deal with them.

The class diagram structure of the abstraction layer is shown in figure 2.1, with class details in figures 2.2 and 2.3. The complexity of the class diagram is due mainly to the presence of hyper-edges in AToM^3. Since dealing with hyper-edges in each layout method is far too time-consuming, the abstraction layer automatically converts hyper-edges into virtual directed edges. Thus instantiating AbstractGraph creates DirectedEdge, Node, and HyperEdge objects from the objects that were on the AToM^3 canvas. The HyperEdge objects, which connect three or more vertices, are then broken down into HyperEdgeComponent and HyperNode objects. The HyperNode is the center point of the

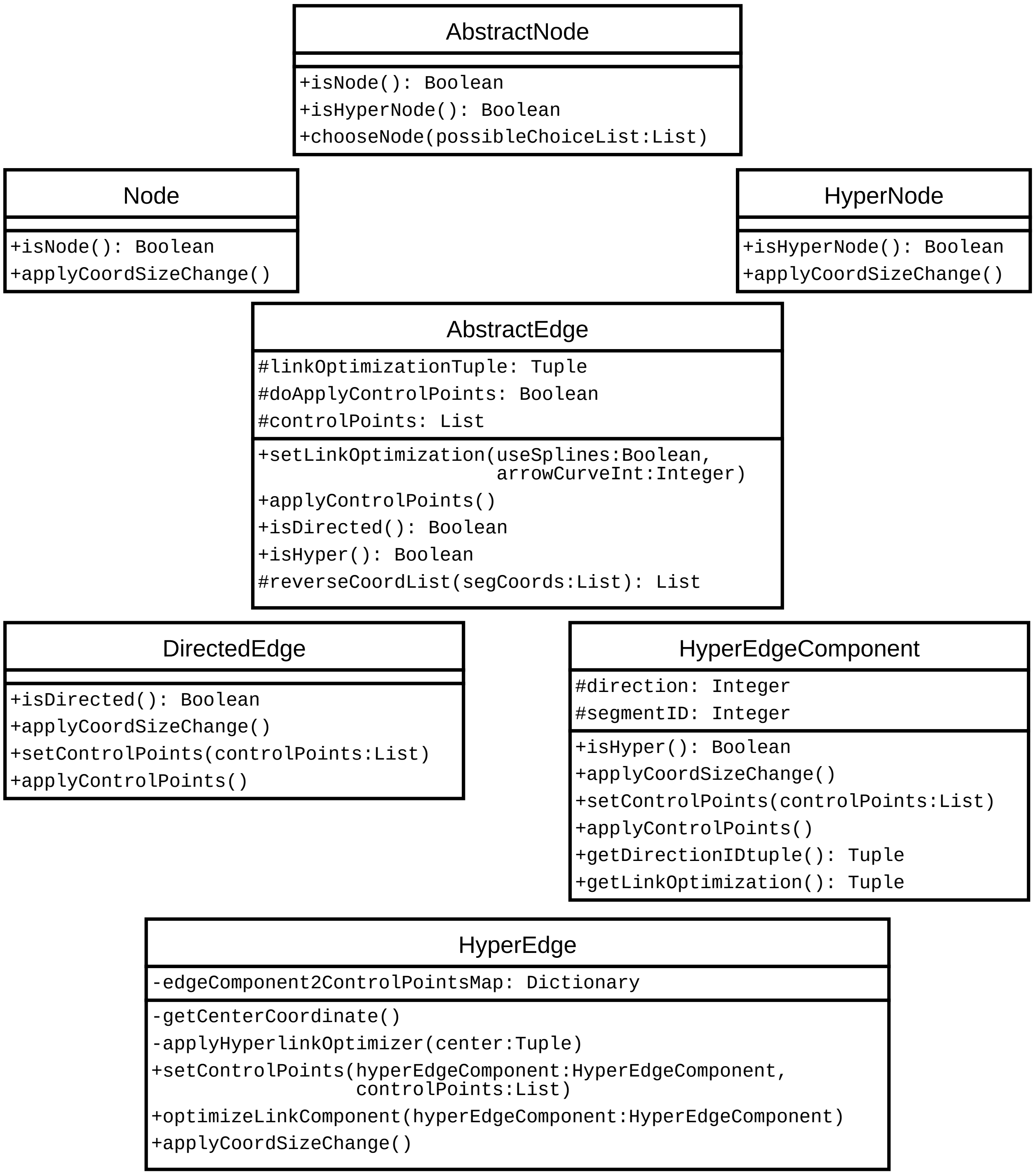

AbstractNode
+isNode(): Boolean
+isHyperNode(): Boolean
+chooseNode(possibleChoiceList:List)
Node
+isNode(): Boolean
+applyCoordSizeChange()
HyperNode
+isHyperNode(): Boolean
+applyCoordSizeChange()
AbstractEdge
#linkOptimizationTuple: Tuple
#doApplyControlPoints: Boolean
#controlPoints: List
+setLinkOptimization(useSplines:Boolean, arrowCurveInt:Integer)
+applyControlPoints()
+isDirected(): Boolean
+isHyper(): Boolean
#reverseCoordList(segCoords:List): List
DirectedEdge
+isDirected(): Boolean
+applyCoordSizeChange()
+setControlPoints(controlPoints:List)
+applyControlPoints()
HyperEdgeComponent
#direction: Integer
#segmentID: Integer
+isHyper(): Boolean
+applyCoordSizeChange()
+setControlPoints(controlPoints:List)
+applyControlPoints()
+getDirectionIDtuple(): Tuple
+getLinkOptimization(): Tuple
HyperEdge
-edgeComponent2ControlPointsMap: Dictionary
-getCenterCoordinate()
-applyHyperlinkOptimizer(center:Tuple)
+setControlPoints(hyperEdgeComponent:HyperEdgeComponent, controlPoints:List)
+optimizeLinkComponent(hyperEdgeComponent:HyperEdgeComponent)
+applyCoordSizeChange()

Figure 2.2: Abstraction layer class diagram details

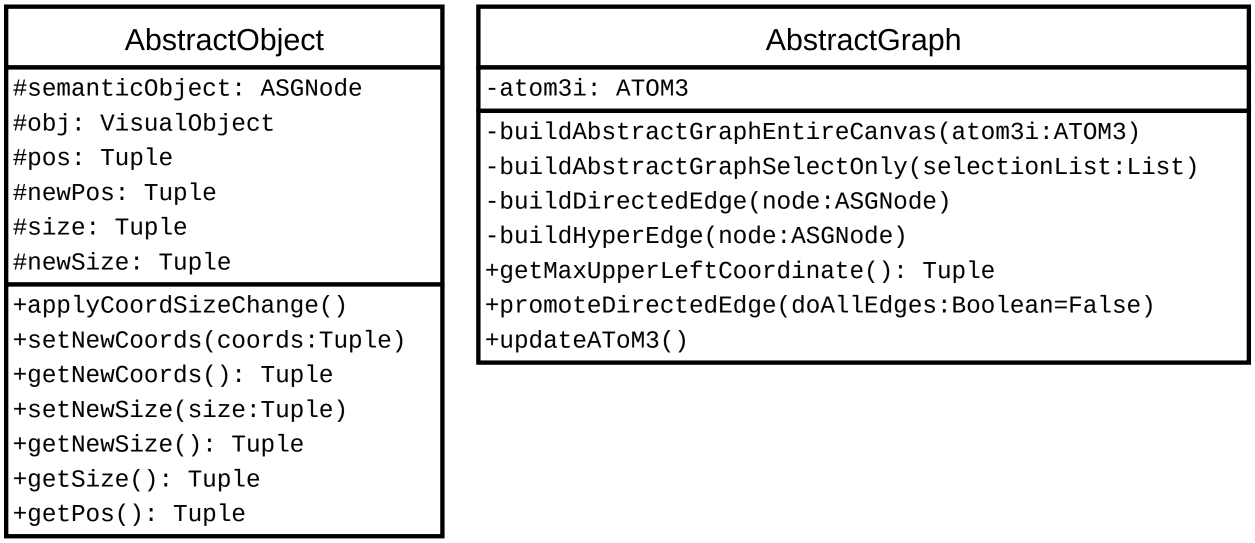


Figure 2.3: Abstraction layer class diagram details

hyper-edge, and the HyperEdgeComponent are the directed edges going from the center point to the other vertices. For the automatic layout algorithm, there is no difference between a DirectedEdge and a HyperEdgeComponent, or between a Node and a HyperNode. Note that in this framework, it is possible for a layout method to process hyper-edges in some special fashion, by directly accessing HyperEdge objects and thus ignore the virtual directed edges.

The meaning of selected methods in the above class diagrams follows:

updateAToM3() In the AbstractGraph, this calls the applyCoordSize() methods of each vertex and applyControlPoints() for each edge.

applyCoordSize() Sets the coordinate and size of the corresponding AToM^3 entity.

applyControlPoints() Sets the control points of an edge in the corresponding AToM^3 relationship.

promoteDirectedEdge() Converts each AToM^3 relationship into a vertex and two edges. By default this method only converts relationships having a center icon.

chooseNode() Requests that a node, from a given list, be chosen by the user. This is handled by AToM^3 and the result is returned to the abstraction layer and hence to the layout algorithm.

getMaxUpperLeftCoordinate() Simply returns the top left coordinate of all the vertices in the abstract graph.

buildAbstractGraphEntireCanvas() Constructs an abstract graph by extracting all the entities and relationships found in the AToM^3 canvas.

AToM^3 graph representation

The internal representation of graphs in AToM^3 is discussed here for comparison with the abstraction layer. Each model in AToM^3 has an associated ASG, abstract syntax graph, object. This object has a dictionary (hash table) with keys for every type of model entity and link in the model. The values of this dictionary are list structures containing every instance of model entity or link occurring in the graph. Both the links and entity objects have attributes indicating whom they are connected too. Hence to find if entity E_a is directly connected to entity E_b, one finds the links connected to E_a and then check if those links connect to E_b. Note that a link may connect to only one vertex, a self-loop, or to more than two vertices, a hyper-edge.

Abstracted graph representation

The graph representation provided by the abstraction layer provides two separate lists, one of vertices and one of directed edges. The model entity/link type information is discarded as irrelevant to layout. Each edge is directed and assigned a single source and target vertex. The source and target vertex are permitted to be the same. Optionally, each vertex is assigned a list of source and target vertices. Using this option, given vertex V_a we can determine if V_b is directly connected to it by simply searching the source and target lists of V_a.

Note that for efficiency reasons, an edge matrix such as the one below is often used to indicate if vertices are connected. This representation makes it possible to find the directed edge from V_a to V_b by simply plugging in the indices for each vertex into the matrix. However, since virtually all graphs arising in practice will have low edge density, this representation is quite wasteful of memory, particularly for large graphs. Furthermore, Python has no built-in support for matrices at this time.

	V_a	V_b
V_a	0	1
V_b	0	0

2.4 Layered

The layered graph drawing technique partitions vertices into layers and then finds good positions for the vertices within those layers. Edges are then drawn in between the vertices. Edges traversing multiple layers are broken down into smaller components, yielding more complex edges with bends in the final drawing. This technique is excellent for visualizing graphs with a hierarchical structure. Graphs that are not truly hierarchical, such as those containing cycles, can sometimes be visualized effectively as well with this technique. Thus it is important that a mechanism for dealing with cycles be implemented.

2.4.1 Design

The implementation of a layered drawing technique does not really require much state information. However the NodeWrapper class is quite convenient and abstracts the difference between a regular vertex, which is in fact an instance of AbstractObject, and a dummy vertex used to represent a portion of a multi-layer traversing edge. The remaining "classes" in the class diagram of figure 2.4 refer to files containing methods and modules containing multiple files. The modules group the core code related to the three phases of layer assignment, crossing minimization, and horizontal positioning. The HierarchicalLayout "class", handles the initialization of the NodeWrappers from the AbstractGraph input, coordinates the activities of the modules, and handles the final assignment of coordinates to the vertices and edges.

2.4.2 Pseudo-code

The layered drawing technique is somewhat complex to implement, as the hierarchical layout class-diagram in figure 2.4 might suggest. This subsection describes the core algorithms inside each of the three modules. There are two exceptions however. The CrossingCounter, an efficient algorithm for counting all the crossings in a layered graph, is very well described in [BJM02] and needs not be reproduced here. Also, due to its elementary nature, the algorithm responsible for determining the final positions of vertices and edges in various orientations is omitted.

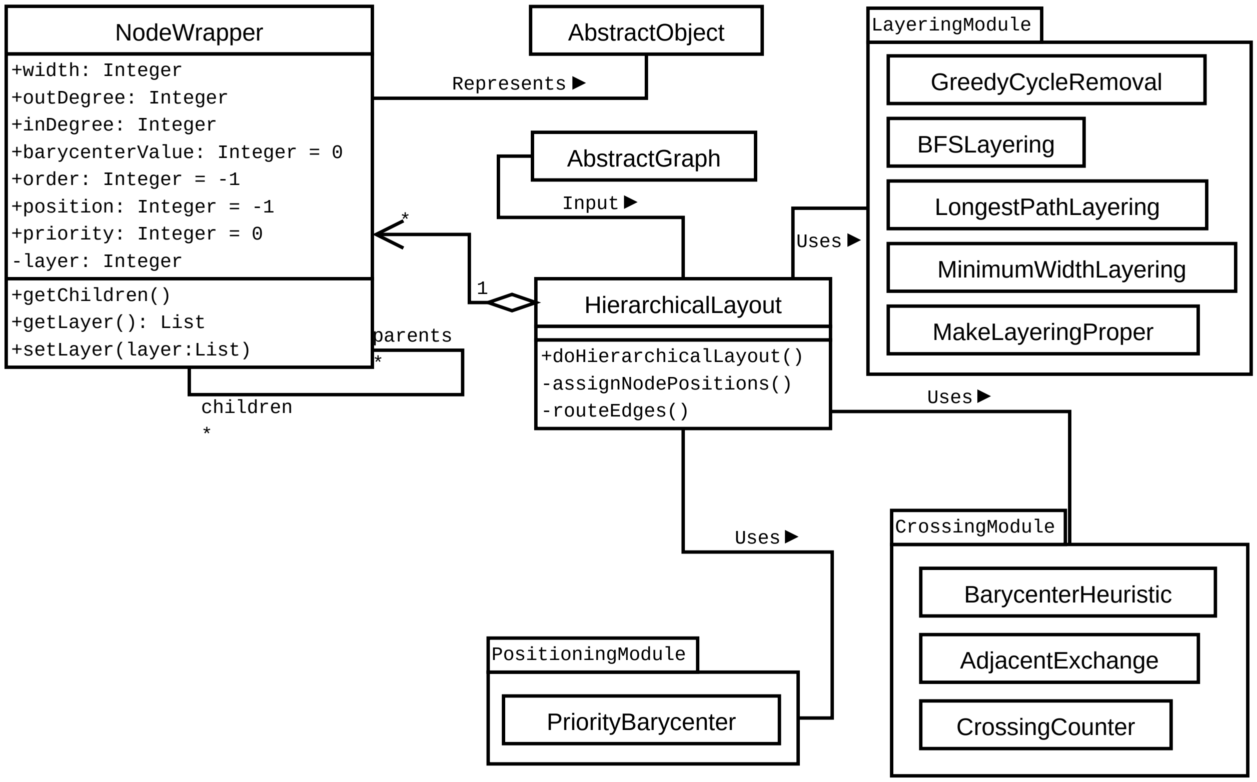


Figure 2.4: Class diagram for the layered drawing technique implementation

Layer assignment

Layer assignment begins with the *greedy cycle removal* algorithm 1. The topological sort simply orders the vertices according to depth first search discovery order. The symbol $\Pi(v)$ stands for the topological ordering of the vertex v. The effect of the topological sort is that each edge will be directed from $\Pi(v_1)$ towards $\Pi(v_{|V|})$, with the exception of cyclical edges. Hence the algorithm visits each vertex in topological order and if the child vertex has an order less than its parent, it is involved in a cycle. All edges thus found to participate in a cycle are reversed. Reversed edges are given a special flag so they can be drawn in their original direction in the final stage of layered drawing method. This algorithm is quite naive and provides no performance guarantees. Better methods can be found in [BM01].

Algorithm 1 Greedy cycle removal

```
Input: A graph G=(V,E)
Output: An acyclic and topologically sorted graph G
1: V ← topological sort of V
2: for all v in V do
3:   for all child in v.getChildren() do
4:     if Π(child) < Π(v) then
5:       reverse edge between v and child
6:     end if
7:   end for
8: end for
```

Three different layering algorithms were implemented: BFS layering, longest-path layering, and minimum width layering. The first two algorithms yield optimal height but unbounded width. Hence on certain graphs the output is unreadable. The minimum width layering is excellent at yielding drawings with good aspect ratios. Unfortunately, it also tends to create longer edges that traverse multiple layers, which in extreme cases can be too long to make any sense of.

The first strategy, BFS layering, is very simple. Since the graph is acyclic, finding the root vertices is trivial. These roots are then labeled as layer 1. Then the breadth first search algorithm is applied to the root vertices and all the immediately discovered vertices are labeled layer 2. This proceeds recursively from the previously discovered vertices, until no more vertices remain. This layering method will place vertices at their graph theoretical distances from their root vertices. Hence if drawn from top to bottom, vertices near the top have a short path from the roots, and vertices near the bottom have long paths from the roots.

The second strategy, *longest-path layering* 2, is somewhat similar to BFS layering. It starts by layering the leaf (sink) vertices at layer 1. Vertices are added to successive layers if all their children are in layers below them. The resulting layering tends to be bottom heavy, in the sense that most vertices will be near layer 1. Note that the order of the layering, from leaf to root, is the exact opposite from the BFS layering algorithm. The pseudo-code given is reproduced from [TNB04].

The longest-path algorithm works as follows. Two initially empty vertex sets, U and Z, are used to store all the vertices assigned to any layer and any layer except the current layer respectively. The algorithm loops until all vertices have been assigned to a layer, hence when |U| = |V|. A selector attempts to choose a vertex v such that v is not already assigned to any layer and $N_G^+(v) \subseteq Z$. The latter condition requires that all the successors of the vertex v lie in the set Z, which contains all vertices assigned in layers below the current layer. If such a vertex exists, it is assigned to the current layer and added to the set U. If no such vertex exists, the current layer is incremented, and the set Z is updated to include U.

Algorithm 2 Longest-path layering

Input: A directed acyclic graph G=(V,E)

Output: An ordered list of layers each containing a list of vertices

```
 1: U ← $\phi$                                         {Vertices assigned to any layer}
 2: Z ← $\phi$                                   {Vertices assigned to previous layers}
 3: currentLayer ← 1
 4: while |U| $\neq$ |V| do
 5:    select a vertex v $\epsilon$ V \ U with $N_G^+$(v) $\subseteq Z$
 6:    if a vertex v is selected then
 7:       assign v to the layer number currentLayer
 8:       U ← U ∪ {v}
 9:    else
10:       currentLayer ← currentLayer + 1
11:       Z ← Z ∪ U
12:    end if
13: end while
```

The last layering algorithm implemented, *minimum width layering* 3, is based on the *longest-path layering* algorithm 2. This algorithm is described in [TNB04]. Unlike the previous two algorithms, this one actively limits the width, or number of vertices, in each layer. For graphs containing unconnected vertices, it was discovered that an additional preprocessing step is necessary. This step simply collects all the unconnected vertices and places them in the first layer/s or last layer/s. If this step is not performed, vertices with zero degree can be drawn on any layer and in any position on that layer. This not only makes such vertices hard to find, it also increases the workload on the horizontal positioning phase when straightening long edges.

The initialization phase of minimum width layering is identical to that of longest-path, save that widthCurrent and widthUp variables are added. These are responsible for storing the width of the current layer and the estimated width of layers above the current layer respectively. An attempt is then made to select a vertex and add it to the current layer as in longest-path. However, rather than choosing the first vertex that belongs to V \ U with $N_G^+(v) \subseteq Z$, the vertex with the maximum out-degree is chosen. If a vertex is chosen, then widthCurrent and widthUp are updated. In the pseudo-code and the actual implementation, dummyWidth refers to the width of a dummy vertex in an edge traversing multiple layers, and is constant for the entire graph. If the dummyWidth was variable, it would not be difficult to incorporate the information herein.

Algorithm 3 Minimum width layering

```
Input: A directed acyclic graph G=(V,E)
Output: An ordered list of layers each containing a list of vertices
1:  U ← $\phi$                                    {Vertices assigned to any layer}
2:  Z ← $\phi$                                    {Vertices assigned to previous layers}
3:  currentLayer ← 1
4:  widthCurrent ← 0
5:  widthUp ← 0
6:  while |U| ≠ |V| do
7:    for all vertices v $\epsilon$ V \ U with $N_G^+(v) \subseteq Z$ do
8:      select the vertex v having the maximum out-degree
9:    end for
10:   if a vertex v is selected then
11:     assign v to the layer number currentLayer
12:     U ← U ∪ {v}
13:     widthCurrent ← widthCurrent - dummyWidth * v.outDegree + v.width
14:     widthUp ← widthUp + dummyWidth * v.inDegree
15:   end if
16:   if minWidthCondition(v) then
17:     currentLayer ← currentLayer + 1
18:     Z ← Z ∪ U
19:     widthCurrent ← widthUp
20:     widthUp ← 0
21:   end if
22: end while
```

The final part of minimum width layering is triggered by a complex condition, the *minWidthStop-Condition,* shown seperately as algorithm 4. This condition is satisfied by one of three things. The first is as in longest-path layering and is satisfied if no vertex was selected. The second condition captures the intuition that if the width of the current layer is greater than some constant, we should stop adding vertices to the current layer. If the vertex has out-degree $\geq$1, then adding the vertex to the layer either decreases or does not change the width size, since long edges with dummy vertices are eliminated. The last condition is concerned with limiting the size of the layer above the current layer. Unlike widthCurrent which is directly compared to UBW, widthUp is compared to UBW * c. This is because we only estimate the size of the layer above in widthUp and need to give ourselves some leeway in choosing the cutoff point for adding vertices to the current layer. Both UBW and c are hard-coded constants rather than parameters. In [TNB04], an extensive parameter study suggests that 1 $\leq$UBW $\leq$4 and 1 $\leq$c $\leq$2.

Algorithm 4 minWidthStopCondition

```
Input: A vertex v
Output: Boolean value
 1: if no vertex v selected then
 2:    return True
 3: else if widthCurrent $\geq$ UBW and v.outDegree $<$ 1 then
 4:    return True
 5: else if widthUp $\geq$ UBW * c then
 6:    return True
 7: end if
 8: return False
```

The final layering step, *proper layering* 5, deals with long edges. Long edges are any edge that traverses more than a single layer. A proper layered hierarchy cannot have such edges. In particular, such edges would cause many crossings in the final drawing. Hence this algorithm replaces each long edge with a series of dummy vertices and connective edges that traverse only a single layer. In the final drawing, these dummy vertices are represented as edge bends.

Algorithm 5 Proper layering

```
Input: A layering of a DAG, L
Output: A proper layering of a DAG
 1: for currentLayer in 1...|L|-1 do
 2:    for all vertices v in $L_{currentLayer}$ do
 3:       for all child vertices of v do
 4:          if abs(child.getLayer() - currentLayer) $>$ 1 then
 5:             for all layers between v and child do
 6:                add a dummy vertex and single layer traversing edges
 7:             end for
 8:          end if
 9:       end for
10:    end for
11: end for
```

Crossing Minimization

The second phase of layered layout is crossing reduction. A variety of techniques to reduce crossings are described in section 1.3.1. Actual implementations were made of a variation of the *layer-by-layer sweep* 6, and two different heuristics to re-order vertices within the layer-by-layer sweep framework. These heuristics are the *barycenter heuristic* 7, and the *adjacent exchange heuristic* 8. In the literature, layer-by-layer sweep is described with little detail, yet it is very important for both the quality and running time of crossing reduction phase. The actual implementation of layer-by-layer sweep has gone through many re-writes, before reaching the form shown in algorithm 6. It resembles most closely the algorithm described in [Pat04].

The initialization phase sets roundsWithoutProgress to zero and bestCrossings to the current number of crossings inherent in the current order of the vertices in the graph layering. The former variable tracks how many consecutive rounds, iterations of the outermost loop, occur without reductions in the number of crossings. The latter variable is, naturally, used to determine if the round has reduced crossings or not.

The downward and upward sweeps are symmetric. Note that the downward sweep does not re-order the last layer, and the upward sweep does not re-order the first layer. The pseudo-code suggests that any heuristic can be used in the key step of the sweep. However since the stopping condition of the loop requires that no further re-ordering occurs, only a deterministic and converging heuristic may be used. Barycenter, mediancenter, and most variations thereof meet this requirement. Indeed, barycenter will require at most O(|L|), where |L| is the number of layers in the layering, sweep iterations to converge.

However, permutation heuristics or even a barycenter that uses a random tie breaking strategy do not converge, and hence are not compatible as is with the *layer-by-layer sweep* algorithm 6. Recall that multiple vertices can receive the same barycenter value, in which case the heuristic has no notion of the best ordering to assign to them, hence the usefulness of random tie breaking. Unfortunately, randomness inside the sweep loops means we need a more complicated termination condition, such as counting the number of crossings to determine if a reduction occurred. This then creates a need for saving promising vertex orderings and backtracking, increasing both the complexity of the algorithm, and potentially the running time.

The layer-by-layer sweep uses a rather lengthy convergence test. Naturally, it stores the current best ordering of the vertices, as compared to the best number of crossings seen thus far. This is necessary because successive rounds can have more crossings than the best found, a particularly common occurrence if random vertex orderings are used. The algorithm halts on one of three possible conditions. The simplest of these is a hard limit on the number of rounds or iterations of the outermost loop. It can be experimentally verified that the first few iterations significantly reduce crossings, whereas succeeding iterations suffer from the law of diminishing returns, hence justifying the use of this limit. It is also quite useful in analyzing the run-time of the algorithm. The second termination condition is triggered if the algorithm has not reduced the crossings for a certain number of consecutive rounds. This reduces the run-time considerably without reducing the final quality by identifying “hopeless” cases. The final termination condition is the trivial case where no crossings remain in the layered graph. Note that although barycenter is guaranteed to yield zero crossings if this is possible, that only applies for bipartite graphs.

A final consideration is randomness. Although the pseudo-code of layer-by-layer sweep, as is, does not allow for random heuristics, one can nonetheless randomize the order of the vertices outside the sweeps. Hence when the layer-by-layer sweep detects that no reduction is being made in the number of crossings, the order of each vertex is randomized. This form of randomization almost always allows the algorithm to further reduce crossings, although it can easily double or even triple the time needed for the crossing minimization phase. Recall that the first few regular iterations of layer-by-layer sweep eliminate most of the crossings, thus the use of randomization should only be used when quality is of the highest importance or the graph size is very small.

The randomization must be used with some care. The effect of a randomization on a graph with two layers requires only one round of sweeping, both downward and upward, to fully converge. However adding more layers, in conjunction with randomization, will not immediately converge. Hence it is recommended to increase the number of rounds without progress that do not trigger a randomization proportionally to the number of layers. The rational for making this proportional to the number of layers rather than the size of the graph comes from the fact that the sweeps require O(|L|) time to converge. Furthermore, the downward and upward sweeps tend to undo each other's work and can thus be expected to require more time to converge when the number of layers increases. Increasing the number of vertices, on the other hand, does not incur this penalty.

The pseudo-code for the *barycenter heuristic* 7, used by the layer-by-layer sweep, is reproduced here for convenience. The notation $N_{fixedLayer}(v)$ refers to the neighbors of the vertex v in the layer fixedLayer. The algorithm is very simple, for each vertex in the moveableLayer a barycenter value

Algorithm 6 Layer-by-layer sweep

```
Input: A proper layering of a DAG, L
Output: A global ordering of vertices within layers that reduces crossings
 1: roundsWithoutProgress ← 0
 2: bestCrossings ← countAllCrossings(L)
 3: repeat
 4:   repeat                                                    {Downward sweep}
 5:     for all currentLayer in 1...|L|-1 do
 6:       moveableLayer, fixedLayer ← $L_{currentLayer}$, $L_{currentLayer+1}$
 7:       Apply heuristic to re-order vertices inside moveableLayer with respect to fixedLayer
 8:     end for
 9:   until The heuristic no longer changes the order of any vertex
10:   repeat                                                      {Upward sweep}
11:     for all currentLayer in |L|...2 do
12:       moveableLayer, fixedLayer ← $L_{currentLayer}$, $L_{currentLayer-1}$
13:       Apply heuristic to re-order vertices inside moveableLayer with respect to fixedLayer
14:     end for
15:   until The heuristic no longer changes the order of any vertex
16:   currentCrossings ← countAllCrossings(L)                 {Convergence testing}
17:   if currentCrossings = 0 then
18:     return L
19:   else if currentCrossings < bestCrossings then
20:     bestCrossings, bestOrdering = currentCrossings, L
21:     roundsWithoutProgress ← 0
22:   else
23:     roundsWithoutProgress ← roundsWithoutProgress + 1
24:     if roundsWithoutProgress ≥ maximum rounds without progress then
25:       return bestOrdering
26:     else if max(1, |L|) rounds since last randomization then
27:       Randomize the order of each vertex in each layer of L
28:     end if
29:   end if
30: until Maximum number of rounds is reached
```

is computed as the average of the orders of that vertex's neighbors. If the vertex has no neighbors, the barycenter is given the value of the order the vertex occupies in moveableLayer. Finally, the vertices of moveableLayer are re-ordered using a sorting algorithm. If the heuristic is being used with the *layer-by-layer sweep* algorithm 6, then the sorting algorithm must be stable. A stable sorting algorithm will not change the order of two vertices having the same barycenter value. In other words, a non-stable sorting algorithm inherently randomizes the order of vertices with the same barycenter value. Also worth noting, is that this heuristic is run repeatedly by the layer-by-layer sweeper, so the vertices are either already sorted or very nearly sorted in all but the initial run and runs following an order randomization.

Algorithm 7 Barycenter heuristic

Input: A moveableLayer and a fixedLayer

Output: A potentially re-ordered moveableLayer, returns True if progress is made

```
1: for all v $\epsilon$ moveableLayer do
2:     if $|N_{fixedLayer}(v)| = 0$ then v.baryCenterValue ← order of v in moveableLayer; continue
3:     v.baryCenterValue ← 0
4:     for all n $\epsilon$ $N_{fixedLayer}$(v) do
5:         v.baryCenterValue ← v.baryCenterValue + order of n in fixedLayer
6:     end for
7:     v.baryCenterValue ← $\frac{v.baryCenterValue}{|N_{fixedLayer}(v)|}$
8: end for
9: Sort moveableLayer according to baryCenterValue
10: If sort changes order, return True, else return False
```

Like the barycenter heuristic, the *adjacent exchange heuristic* algorithm 8, sorts the vertices in the movable layer. The heuristic is essentially the bubblesort algorithm applied to crossing numbers. Crossing numbers can give a tight lower bound on the number of crossings in a graph, but cannot be used to exactly compute the number of crossings. However, when two vertices are exchanged, the number of crossings is reduced exactly by the crossing number of the first vertex minus the crossing number of the second. Although this sounds great, removing a crossing on one layer of a graph usually creates a crossing on another layer, hence the need for the layer-by-layer sweeping.

The crossing numbers are computed as a matrix. The diagonal of the matrix is always zero. A non-zero value would indicate that somehow two or more edges starting at the same vertex were crossing with each other. Assuming for simplicity that edges start at the center of a vertex, this is impossible. A crossing $C_{u,v}$ is defined as occurring when a vertex V_u has order less than a vertex V_v in the movable layer and in the fixed layer, the neighbors of V_u have order greater than the neighbors of V_v. The inverse crossing, $C_{v,u}$ is defined as occurring symmetrically to $C_{u,v}$. Instead of the neighbors of V_u having order greater than the neighbors of V_v, it is just the opposite for the inverse crossing number.

Horizontal positioning

A priority based barycenter method is implemented to determine the final horizontal positions of the vertices, and is described in [ST81]. This method is fast and has a very similar implementation to that of the crossing reduction phase. Unfortunately, it is very difficult, if not impossible, to improve the quality of the final drawing beyond some point with this technique. Ideally, one would like the long edges to be straight, but priority-barycenter tends to yield long edges that look like spaghettis. This particularly annoying considering that most horizontal positioning methods can guarantee at most two bends at the extreme ends of the long edge. Also, the method is implemented to work on an integer grid. Final coordinates are computed as a post-process by considering the size of the

Algorithm 8 Adjacent exchange heuristic

```
Input: A moveableLayer M and a fixedLayer F
Output: A potentially re-ordered moveableLayer, returns True if progress is made
 1: C ← Initilize crossing number matrix, size |M|×|M|, to zero
 2: for all i in 1...|M| - 1 do
 3:    for all j in i + 1...|M| do
 4:       for all neighborA of $M_i$ and neighborB of $M_j$ do
 5:          if neighborA.order > neighborB.order then
 6:             $C_{i,j} \leftarrow C_{i,j} + 1$
 7:          else if neighborA.order < neighborB.order then
 8:             $C_{j,i} \leftarrow C_{j,i} + 1$
 9:          end if
10:       end for
11:    end for
12: end for
13: repeat
14:    for all j in 0...|M| - 1 do
15:       if $C_{j,j+1} > C_{j+1,j}$ then
16:          Exchange vertex $M_j$ with $M_{j+1}$
17:       end if
18:    end for
19: until No exchanges occur
20: If at least one exchange, return True, else return False
```

Algorithm 9 Priority-barycenter heuristic

```
Input: A moveableLayer M, and a fixedLayer F
Output: Sets the grid positions of vertices in M towards their barycenters
 1: Apply the Barycenter heuristic to M and F, but do not sort
 2: isMakingProgress ← False
 3: for all i in 1...|M| do
 4:    if $M_i$.baryCenterValue > $M_i$.position then
 5:       isMakingProgress ← pushMove(M, i, +1)
 6:    else if $M_i$.baryCenterValue < $M_i$.position then
 7:       isMakingProgress ← pushMove(M, i, -1)
 8:    end if
 9: end for
10: return isMakingProgress
```

largest vertex on a layer and setting the grid spacing accordingly. A more space efficient method could take advantage of the fact that long edges need far less space than vertices.

The initialization step consists of computing starting horizontal positions and vertex priorities. The initial horizontal positions of the vertices is simply the order numbers obtained in the crossing reduction phase. Applying creative strategies to the initial positions, such as varying the spacing between vertices, does not improve the quality of the final drawing. Indeed, it can have a negative impact on the area efficiency of the drawing. The priority of a regular vertex is simply the degree of the vertex. For the dummy vertices, of which long edges are composed, the priority is set to infinity. This gives the long edges the maximum chance of being straight.

The next step is simply the *layer-by-layer sweep* 6, previously shown. However, the *barycenter heuristic* 7 is replaced with the *Priority-barycenter heuristic* 9. This new heuristic uses the old barycenter heuristic as a sub-algorithm, although the sorting phase is very different. Clearly, the usual concept of sorting at this stage would simply increase the crossings which the crossing minimization phase put so much effort into reducing. Instead, the vertices are moved toward their barycenter values, if not already there, using the recursive *pushMove* algorithm 10.

Algorithm 10 pushMove

```
Input: A moveableLayer M, an index i$\epsilon$\{1...|M|\}, and direction d$\epsilon$\{-1,+1\}
Output: ...
 1: canMove ← False
 2: if (d < 0 and i = 1) or (d > 0 and i = |M|) then                {Margin case}
 3:     canMove ← True
 4: else if $M_i$.position + d ≠ $M_{i+d}$.position then            {Free spot case}
 5:     canMove ← True
 6: else if $M_i$.priority > $M_{i+d}$.priority then                {Recursive push case}
 7:     canMove ← pushMove(M, i + d, d)
 8: end if
 9: if canMove then
10:     $M_i$.position ← $M_i$.position + d
11:     return True
12: end if
13: return False
```

The pushMove algorithm attempts to move a vertex one grid unit in a given direction. The direction is set by the Priority-barycenter heuristic, it is the direction the vertex must move in order to reach its barycenter value. If the vertex is at the extreme ends of the layer it resides on then it is free to move further towards those ends. Note that a vertex can acquire a negative grid position which will need correction by a post-process. Alternatively, the position the vertex moves to has no vertex in it, in which case it is free to move there. Finally, the vertex attempts to move into a position with a blocking vertex in it. If the vertex has greater priority than the blocking vertex, a recursive call attempts to move the blocking vertex. If all the recursive calls succeed, then all the vertices involved move by one unit.

Layer-by-layer sweep must also be modified in its convergence testing. There is no metric equivalent to counting crossings to terminate this sweep early. However, since the barycenter method is deterministic, the downward and upward sweeps will completely cancel each other out at some point. Hence convergence can be defined as the positions of the vertices before layer-by-layer sweep being equal to the positions afterwards. On some inputs, this convergence test is insufficient since the position of each vertex can be shifted by one unit to the left or right every iteration. A simple additional test can deal with this. For each layer M, if either $M_1.position > 1$ or $M_{|M|}.position <$

$|M|$ occur, then all the vertices must have shifted left or right. This additional test is correct since the pushMove algorithm, given an initial positioning between 1 and |M|, can only push vertices outside the range 1 to |M|. Finally, a hard limit on the maximum number of rounds is used, as in the original version of layer-by-layer sweep. This test is probably quite redundant since convergence typically occurs in less than a handful of rounds.

2.4.3 Analysis

The analysis of the overall run-time complexity of a layered drawing technique is generally not given. This is due, in particular, to the difficulty of analyzing the running time of the crossing minimization phase. Indeed, the second phase is considered the greatest performance bottleneck to this drawing technique. It is unfortunate that edge crossings have a large impact on the quality of the final drawing when crossing minimization is already an NP-hard problem with just two layers of vertices. The problem is made even worse by the requirement of proper layering, which introduces a large number of dummy vertices where multi-layer traversing edges occur. Hence in the following analysis, whenever crossing minimization and horizontal placement consider vertices and edges, they include those vertices and edges introduced by the proper layering. Significant improvements to the running time can thus be made by following the approach of [ESK04], where no more than two dummy vertices are generated per multi-layer edge.

Layer Assignment

Layer assignment requires an acyclic directed graph, hence the first step is to convert cyclic graphs to an acyclic form if necessary. The greedy cycle removal algorithm uses a topological sort and then visits each vertex and edge just once. This yields a run-time of O(|V|+|E|).

The first layer assignment strategy uses breadth-first search to determine the layering. This well known algorithm uses O(|V|+|E|) time. The second strategy, longest-path-layering, has a linear time according to [BM01]. This requires careful use of the properties of a topological ordering. However the current implementation does not do this. Instead, the “select a vertex” step uses a loop over all unassigned vertices. This results in quadratic time complexity. The third strategy, minimum-width-layering, is a refinement of the longest-path-layering strategy. According to its authors, [TNB04], the run-time is “polynomial”. In the worst case, the additional stop-condition will cause a new layer to be generated after every vertex is assigned, hence running the outer loop one more time than the longest-path layering algorithm. Hence if longest-path-layering is linear then minimum-width-layering is quadratic. Therefore the current implementation of minimum-width-layering has worst-case cubic time complexity.

The final layer assignment step, proper layering, ensures edges cross only one layer. This requires visiting every vertex and edge once. Hence this algorithm is O(|V|+|E|).

Crossing minimization

The crossing minimization phase consists of a layer-by-layer sweep and a barycenter heuristic. The outer loop of layer-by-layer sweep is bounded by a constant c. The inner loops consist of downward and upward sweeps. A single sweep requires |L| iterations to re-order all the layers, where |L| is the number of layers generated by the layer assignment phase. Moreover, the downward/upward sweeps require at most |L| iterations before they converge. Thus the sequence of downward and upward sweeps require $O(c * 2 * |L|^2)$ time.

A single sweep runs the barycenter heuristic on a single layer L_i. The barycenter heuristic must visit each vertex in L_i and each edge between L_i and L_{i+1} (or L_{i-1} for upward sweeps). Barycenter then sorts each vertex in L_i. Hence barycenter requires O(|V| log |V| + |E|), where V and E are the vertices and edges of layer L_i respectively. Thus the entire algorithm is $O(c * |L|^2 * (|V_{LMax}|$ log

$|V_{LMax}| + |E_{LMax}|)$) where V_{LMax} and E_{LMax} are the maximum number of vertices and edges found on any layer. This indicates that the implementation of the layer-by-layer sweep and barycenter algorithms are highly sensitive to the height of the layering.

Alternatively, a single sweep runs the adjacent exchange heuristic on a single layer L_i. In the literature, adjacent exchange is considered a $O(|L_i|^2)$ algorithm. Indeed the sorting portion of adjacent exchange is equivalent to bubblesort and does indeed have a $O(|L_i|^2)$ running time. However, barring the existence of a very clever crossing number update scheme, the crossing numbers must be recomputed for every single sweep. This requires visiting every vertex in the layer, comparing it with every other vertex of greater order, and visiting all the edges of each vertex pair. Therefore, if V are the vertices of layer L_i and E the edges between L_i and L_{i+1}(or L_{i-1} for upward sweeps), then adjacent exchange is really $O(\frac{1}{2}|V|^2|E|)$.

Horizontal positioning

The horizontal positioning phase also uses layer-by-layer sweep. Once again the outer loop is bounded by a constant c and the sequence of downward and upward sweeps yield a total of $O(c * 2 * |L|^2)$ time complexity. The priority-barycenter heuristic uses the regular barycenter heuristic as a subroutine but without the sorting. Simply calculating the barycenter values for a layer L_i requires $O(|V| + |E|)$ time, where V and E are the vertices and edges of layer L_i respectively.

Once the barycenter values are computed, priority-barycenter then loops over all the vertices of the a layer L_i, and makes $|L_i|$ calls to the pushMove algorithm. In the worst-case for pushMove, a vertex on the extreme left would succeed in pushing all the vertices in the layer to the right. This yields $|L_i|$ time complexity for pushMove, where L_i is the layer pushMove received as input. Therefore priority-barycenter requires $O(|L_i|^2)$ time to consider each vertex in the layer and move it to its barycenter with pushMove. Hence this last step dominates the run-time for priority-barycenter.

Combining the outer and inner loops, the running time of horizontal positioning is $O(c * |L|^2 * (|V_{LMax}|^2))$. Once again, V_{LMax} represents the maximum number of vertices found on any layer, and |L| the total number of layers. This time complexity is misleading however. Consider that the previous crossing minimization phase has already ordered the vertices by their barycenter values. It stands to reason that each vertex is fairly close to where it "wants" to be and will not require anywhere near the worst case number of pushes in the pushMove algorithm. Also, this algorithm tends to converge much faster than the crossing minimization one, particularly when the latter uses randomization rounds, so the constant c is much smaller.

Overall running-time

The layer assignment phase varies between $O(|V| + |E|)$ and cubic time complexity depending on the layering strategy. If we make the assumption that the number of layers and the size of a layer are roughly half the number of vertices in a graph then the time complexity of the last two phases can be simplified. Thus crossing minimization requires $O(|V|^3 \log |V| + |V|^2|E|)$ time and horizontal positioning requires $O(|V|^4)$ time. Hence the overall time complexity for this algorithm is quartic.

2.4.4 Case-study

The layered drawing technique gives excellent results on a wide range of formalisms. Some examples of models drawn with this implementation of layered layout are shown in figures 2.5, 2.6, and 2.7. In the first figure, which shows a model in the Causal Block Diagram formalism, a nice effect of the layered layout is that constants are grouped on the left, whereas the plotting visualization is on the far right. This occurs in every model in that formalism. The second figure is of a telephone model in the GPSS formalism and is drawn from top to bottom rather than from left to right. The last model, a reachability graph, was generated with a graph grammar transformation from a Petri Net.

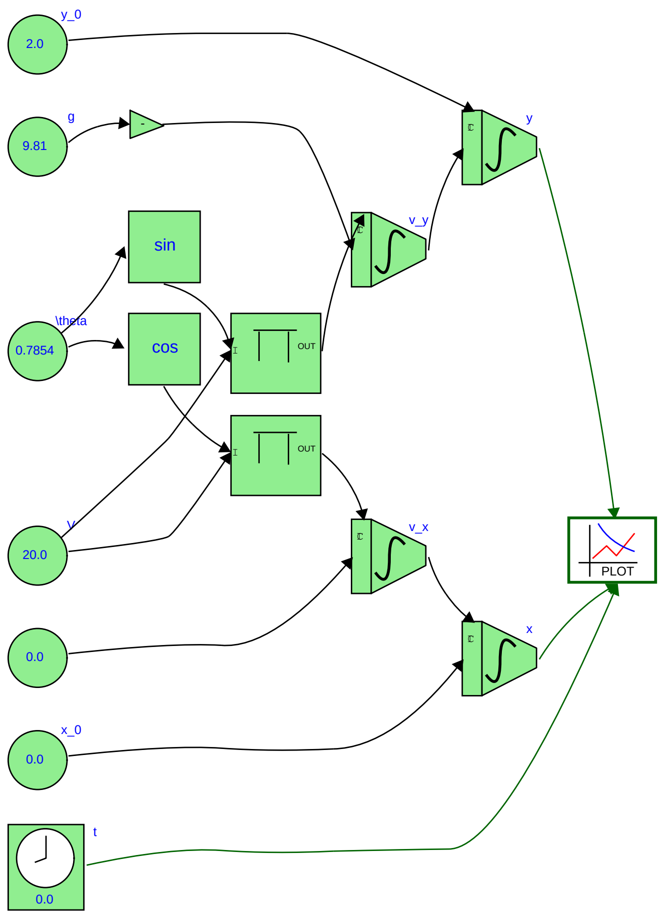


Figure 2.5: Ballistic model in Causal Block Diagram formalism

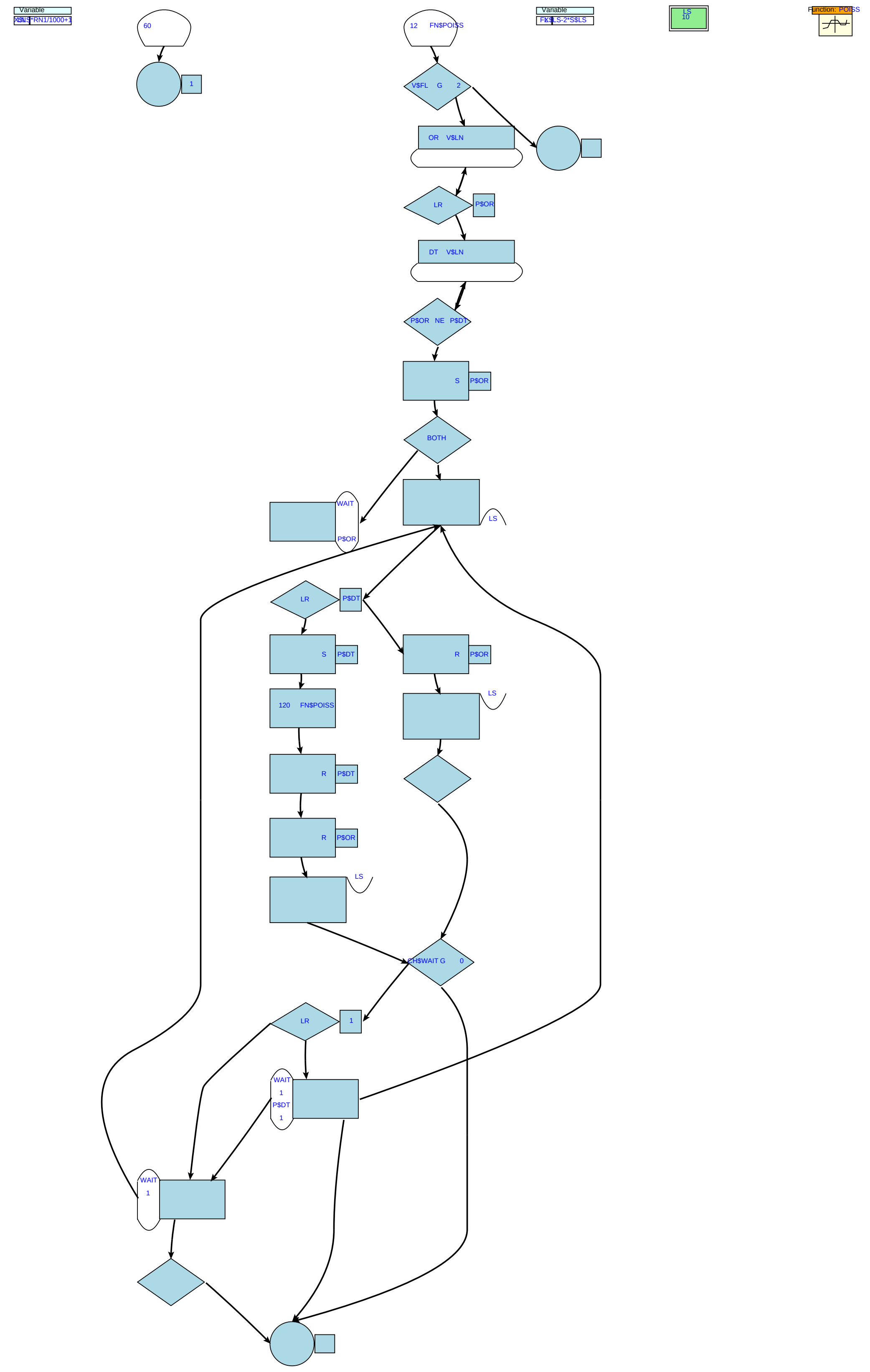


Figure 2.6: Telephone model in the GPSS formalism

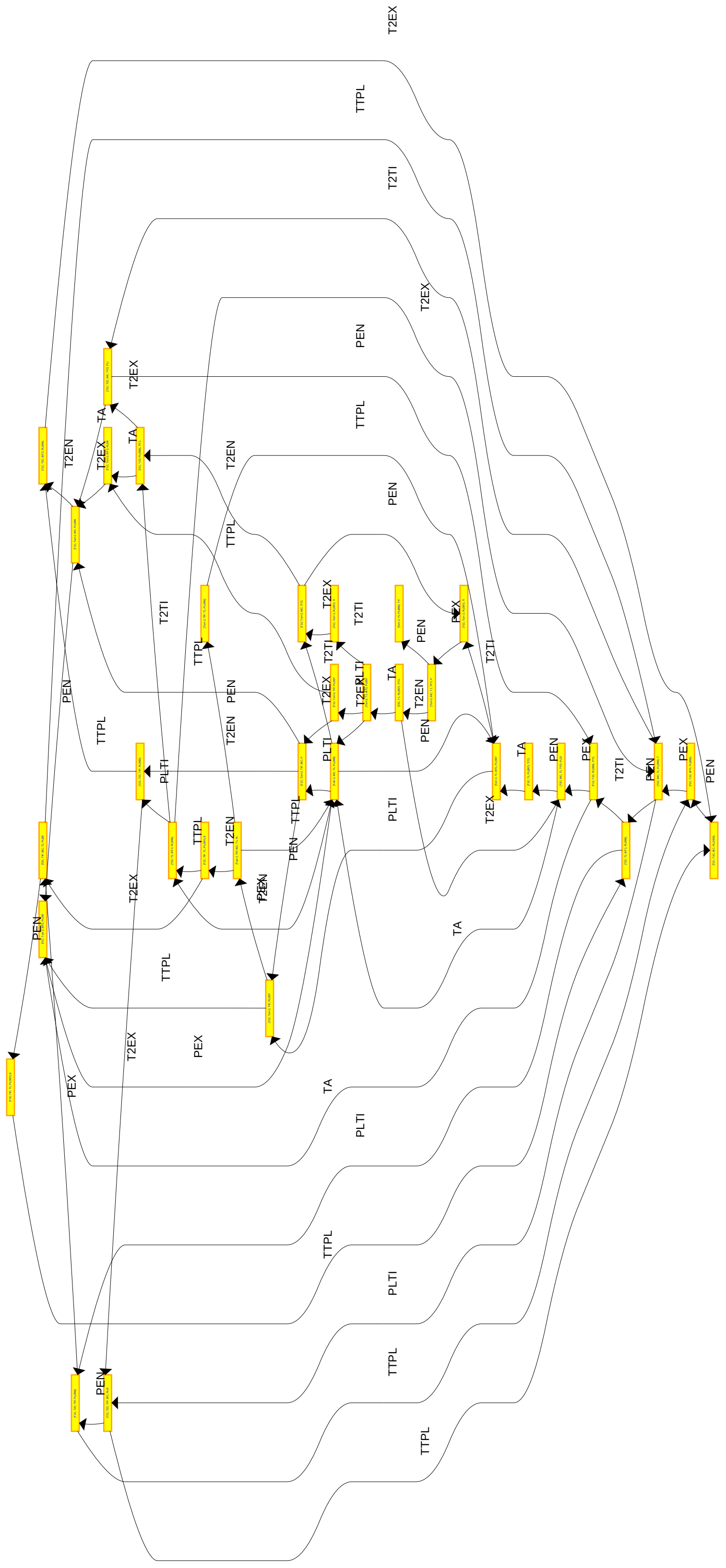


Figure 2.7: Model generated from a Petri Net in the Reachability Graph formalism

It puts particular strain on this drawing technique, because there is no net direction of flow (Note that some Petri Net reachability graphs do have such a flow). Recall from section 1.3.1 that the layered drawing technique assumes that there exists a direction of flow. In any case, the generated model, in the Reachability Graph formalism, reveals all possible states that the original Petri Net could reach. In this particular model, it can be seen in the center-right that transition PEN reaches a state from which no other states can be reached.

2.5 Spring-embedder

The spring-embedder is a form of force-directed layout. Edges are simulated as springs and vertices as rings to which the springs are attached to. These are the attractive forces. In this implementation, electrical repulsion charges as well as gravitational forces are added. The repulsion forces, generated by each vertex, prevent vertices from overlapping. The gravitational force drastically increases the area efficiency of the final drawing.

The main advantage of this graph drawing algorithm is that it is simple to implement. Moreover, it is customizable for different tasks by adding or removing forces. The running time is quadratic with the number of vertices since calculation of repulsive charges is done between every pair of vertices. The simulation of forces is run for a fixed number of iterations, by default 100 iterations. This has proven to be sufficient in tests, particularly when the graph is first preprocessed. The circle layout algorithm, in section 2.7, serves as a linear time preprocessing phase.

2.5.1 Pseudo-code

The pseudo-code for the simulation loop of the *spring-embedder* is given in algorithm 11. A pre-processing step of either circle layout or even a random layout algorithm is recommended. Doing so can improve the convergence speed and quality of the final drawing. The initialization step then consists of acquiring the center coordinates of the vertices, setting 2D force vectors to zero, and setting the repulsion charges. The repulsion charges are set to the diagonal length of each vertex. The motivation for doing so is that the charges will be at least strong enough that any other vertex entering the bounding circle of the vertex is strongly repulsed, thus eliminating vertex overlaps. The simulation step repeatedly calculates the forces acting upon the vertices. Once all the forces are calculated, the positions of the vertices are modified, and another simulation run begins. The algorithm terminates after 100 iterations, which is typically enough, or if a convergence threshold is triggered.

The convergence test simply checks if the maximum force acting on a vertex is below a parametrized threshold. A value of 10.0 was experimentally found to work well for graphs with 4, 30, and 126 vertices. Forgiveness rounds are used to ensure that a given simulation run that made little progress in terms of maximum force does not trigger convergence too early. Thus if two forgiveness rounds are used, the maximum force must be less than the threshold force for two consecutive rounds for convergence detection. The convergence test considerably improves running time, particularly if the input graph is already drawn similarly to the final result, such as when the circle preprocessing step is used. The only disadvantage, particularly noticeable with a handful of vertices, is that symmetries are not fully realized. This is due to the fact that the repulsion forces between relatively distant vertices are quite slight and take many simulation runs to add up sufficiently to realize perfectly symmetrical shapes. An example of this is a vertex with three vertices connected to it, where the desired shape of three equidistant vertices around the center vertex does not occur with the convergence test.

The *repulsion* algorithm 12, is responsible for both avoiding vertex overlaps and revealing symmetries. Overlaps are avoided by simply generating large repulsive forces whenever two vertices overlap. The symmetries are the result of the repulsive charge extending well beyond a vertex,

Algorithm 11 Spring-Embedder

```
Input: A graph G=(V,E)
Output: An embedding of G
 1: for all v in V do                                              {Initilize}
 2:    v.forceVector ← 0
 3:    v.pos ← center coordinate of v
 4:    v.charge ← chargeStrength * diagonal length of v
 5: end for
 6: for all i such that $0 \leq i \leq 100$ do                    {Simulation}
 7:    Repulsion(V)
 8:    Attractive(E)
 9:    Gravity(V)
10:    for all v in V do                                            {Update}
11:       v.pos ← v.pos + v.forceVector
12:       v.forceVector ← 0
13:    end for
14:    Test for convergence
15: end for
```

albeit weakly, thus causing vertices to space out nicely. Note that the given pseudo-code is very naive, it calculates the vertex pair ($\mathrm{v_a}$, $\mathrm{v_b}$) separately from the pair ($\mathrm{v_b}$, v_a), rather than doing both at once. This is for simplicity of presentation, please see the *Force Transfer* algorithm 15, for a more efficient method that takes advantage of the fact that the force from $\mathrm{v_a}$to $\mathrm{v_b}$ is simply the negative of the force from $\mathrm{v_b}$ to $\mathrm{v_a}$.

The first step in calculating repulsion forces is to find the Manhattan and Euclidean distances between the pair of vertices. If the Euclidean distance is greater than some threshold, the impact of the force is very slight and can be ignored for the sake of efficiency. For maximum efficiency, one would partition the vertices beforehand to determine which have impact on which others. This was not implemented however, since the speed of the algorithm was satisfactory for the given test graphs.

The final repulsion step consists of calculating a scalar force proportional to the charges of the vertices and inversely proportional to the square of the distance separating the vertices. Recall that the repulsive charge of each vertex is proportional to its size. This scalar force is then multiplied by the 2D Manhattan distance vector, yielding an increment to the 2D force vector. In the case where the Euclidean distance between the pair of vertices is really small, less than 0.1, then the scalar force is calculated as just value of the charge. The motivation for this lies in the fact that the repulsion charge divided by a very small value yields a very large value, potentially creating a force large enough to launch a vertex into orbit.

The *attractive* algorithm 13, treats edges as physical springs. The first step consists of finding the Manhattan and Euclidean distances between the pair of vertices connected to the edge. If the Euclidean distance between the two vertices is near zero, then artificially setting the distance to a minimum value avoids precision and divide by zero issues. A minimum distance of 0.1 works well.

The final step in the algorithm calculates the spring force using the physical equation for springs. The spring constant determines how violently the spring expands/contracts when not at its ideal length. Too low a value results in a sluggish spring that does not try very hard to achieve its ideal length. Too high a value results in a spring that oscillates above and below its ideal length. Fortunately, a value of 0.1 for the spring constant seems to work across a wide range of graphs.

Algorithm 12 Repulsion

```
Input: A set of vertices V
Output: Update force vectors for V
 1: for all $v_a$ in V do
 2:   for all $v_b$ in V do
 3:     if $v_a \neq v_b$ then
 4:       calculate Manhattan distance vector and Euclidean distance between $v_a$ and $v_b$
 5:       if abs(Euclidean distance) $>$ threshold then
 6:         charge $= v_a$.charge $+ v_b$.charge
 7:         if abs(Euclidean distance) $> 0.1$ then
 8:           force $= \frac{charge}{(Euclidean\,distance)^2}$
 9:           v.forceVector $\leftarrow$ v.forceVector $+$ (Manhattan distance vector) * force
10:         else
11:           v.forceVector $\leftarrow$ v.forceVector $+$ sign(Manhattan distance vector) * charge
12:         end if
13:       end if
14:     end if
15:   end for
16: end for
```

The ideal length is user definable graph wide parameter. A default value of 100 pixels is used. A more sophisticated strategy might vary the ideal length according to the total degree of the vertices connected to the edge. The intuition is that high degree vertices are densely packed together, so increasing the ideal length will give them more room to avoid overlapping. Finally, the computed spring force is multiplied by the 2D Manhattan distance vector and added to the force vector of one vertex and subtracted from the other.

Algorithm 13 Attractive

```
Input: A set of edges E
Output: Updated force vectors for vertices linked to E
 1: for all e in E do
 2:   $v_s \leftarrow$ e.getSource()
 3:   $v_t \leftarrow$ e.getTarget()
 4:   if $v_s \neq v_t$ then                                    {Avoid loop edge}
 5:     calculate Manhattan distance vector and Euclidean distance between $v_s$ and $v_t$
 6:     if Euclidean distance $<$ minDistance then
 7:       Euclidean distance $\leftarrow$ minDistance * sign(Euclidean distance)
 8:       Manhattan distance vector $\leftarrow$ minDistance * sign(Manhattan distance vector)
 9:     end if
10:     force $\leftarrow$ springConstant * $\frac{(Euclidean\,distance) - idealSpringLength}{Euclidean\,distance}$
11:     $v_s$.forceVector $\leftarrow v_s$.forceVector $+$ (Manhattan distance vector) * force
12:     $v_t$.forceVector $\leftarrow v_t$.forceVector - (Manhattan distance vector) * force
13:   end if
14: end for
```

The *gravity* algorithm 14, attempts to increase the area usage efficiency. It does not really simulate gravity. True gravity would require each vertex to have mass and accelerate the vertices towards some strong gravitational field source. Instead, a pseudo-gravity imparts upon each vertex a velocity towards the gravitational field source. Masses are ignored, which is equivalent to considering each vertex to have a unit mass. The gravitational field source is determined to be the barycenter of all the vertices. The intuition for this is that the barycenter will be the densest part of the graph drawing, hence vertices should attempt to get as close to this point as possible.

The force vector imparted on each vertex is calculated as the unit vector between the vertex and the barycenter and the strength of the gravity field. If the strength of the gravity field is negative, the resulting drawing will be quite spread out, but not area efficient. A value of 10, which by some coincidence is close to the value of gravity on the surface of Earth, works well, at least for small sparse graphs. The trade-off with high gravities is that though they make good use of area, they increase the number of edge crossings, making the drawing less readable. Since the gravity is circular in nature for two dimensions, it also yields drawings with a circular perimeter, rather than a rectangular drawing that is better suited for a viewing device.

Algorithm 14 Gravity

```
Input: A set of vertices V
Output: Updated force vectors for V
1: barycenter ← $\frac{\sum_{v \varepsilon V} v.pos}{|V|}$
2: for all v in V do
3:     calculate unit vector between v.pos and barycenter
4:     v.forceVector ← v.forceVector + unit vector * gravityStrength
5: end for
```

2.5.2 Analysis

The simulation loop for spring layout terminates in at most a constant 100 iterations. For small graphs with vertices and edges on the order of 100, the layout usually converges before this maximum is reached. It is not clear from the literature whether the required number of iterations is constant or not. In [FR91], the authors lament this lack and then go on to say that their own efforts to pin the number of iterations as a function of the graph size failed. Hence they too used a constant number of iterations, in their case just 50. Indeed, the use of a fixed number of simulation iterations can be justified since in those rare situations where a greater number of iterations are needed, the algorithm can be run a second time using the previous result as a starting point.

The repulsion algorithm dominates the time complexity for each simulation iteration. It requires O(|V|^2) time since each vertex repulses every other vertex. Using a partitioning scheme, this can be reduced to O(|V| log |V|) at best, [QE01]. The attractive and gravity algorithms use only O(|E|) and O(|V|) time respectively. Hence the overall time complexity for this implementation is O(|V|^2).

2.5.3 Case-study

The spring-embedder drawing technique is not applicable to as many formalisms as is the layered. The key issues with this layout are the lack of crossing minimization and the unstructured appearance of the final drawing. Formalisms that generally work well with this type of layout include Finite State Automatons and Petri Nets. A transmitter model in the Petri Net formalism drawn with the spring-embedder is shown in figure 2.8.

2.6 Force-transfer

The force-transfer drawing technique is another example of the force directed approach. It consists of a simulation whereby each vertex exerts forces on overlapping neighboring vertices. The simulation terminates once the forces have pushed all vertices apart such that no overlap remains. This

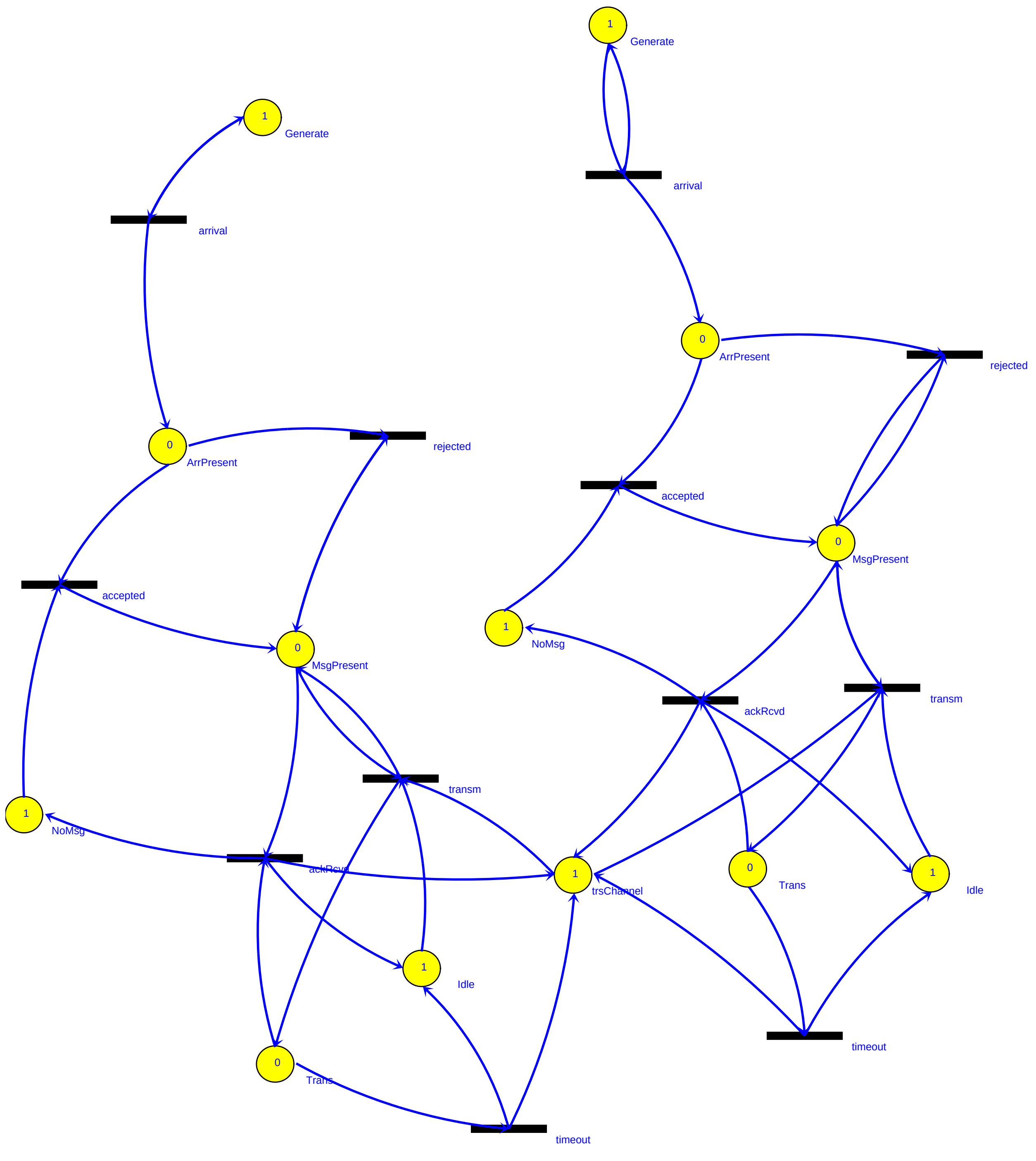


Figure 2.8: Transmitter model in the Petri Net formalism

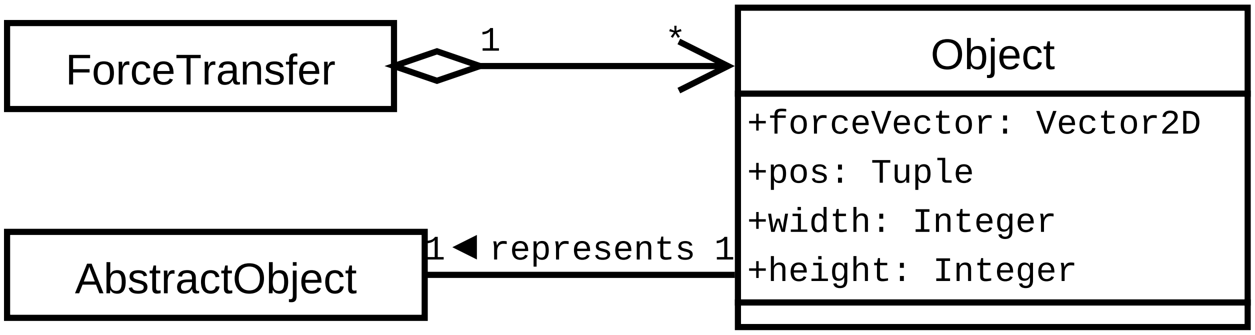


Figure 2.9: Force-transfer class diagram

technique is useful either as a post-process to another layout technique that does not adequately address overlap issues or on its own.

2.6.1 Design

The design for the force-transfer algorithm is shown in figure 2.9. This design is very basic, consisting of ForceTransfer which does the force simulation and Object. Object is a wrapper around the AbstractObject. Object doesn't necessarily represent just vertices, it can also represent hyper-edge centers or even edge bends. It extends the functionality of AbstractObject by allowing querying for center coordinates rather than top-left vertex coordinates and by tracking the forces acting on the object.

2.6.2 Pseudo-code

The *force transfer* algorithm 15 consists of an initialization and simulation phases. The initialization phase sets forces the forces acting on each vertex to zero and the position of the vertex to its center coordinate. The simulation loop then iterates over every pair of vertices. The simulation of the vertex pair ($\mathrm{v_a}$, $\mathrm{v_b}$) is equivalent to that of the pair ($\mathrm{v_b}$, $\mathrm{v_a}$), hence only one vertex pair is used per simulation run. Once all the forces have been calculated between the vertices, the positions of the vertices are updated accordingly.

Simulation ends when convergence is detected, that is when no vertex overlaps remain. However, the simulation is also terminated by a fixed number of iterations, to avoid running the force transfer algorithm for a very long time on certain inputs. For example, a hundred vertex input graph where the vertices are embedded such that they all have the same coordinate, would not converge rapidly. In such cases, a better strategy is to apply a different layout method first. The layout method need not be sophisticated, even a completely random layout technique can give good results.

The core of the force transfer algorithm is *calculateForce* 16. The first step consists of elementary calculations of Manhattan and Euclidean distances and the unit vector distance between the pair of input vertices. With these, a scalar force magnitude is computed. The force magnitude must lie between zero and negative one to have any impact on the positions of vertices. Thus if it is outside this range, the vertices involved are not overlapping.

The force is only applied horizontally or vertically and not both at once. The direction of the force is determined by the greatest separating distance between the vertices. The motivation for this is to move the vertices apart as little as possible such that they no longer overlap. This reduces the chance that a vertex will force another vertex away only to create a new overlap. Moreover, by moving vertically or horizontally only, area efficiency is increased. This is particularly noticeable when compared to a naive alternative to force transfer, scaling.

Algorithm 15 Force Transfer

```
Input: A graph G=(V,E)
Output: An embedding of G
1: for all v in V do                                            {Initilize}
2:   v.forceVector ← 0
3:   v.pos ← center coordinate of v
4: end for
5: for all i such that $0 \leq i \leq 50$ do                        {Simulation loop}
6:   isMoving ← False
7:   i ← 0
8:   j ← 0
9:   while i < |V| do
10:    while j < |V| do
11:      if i ≠ j and calculateForce($v_i$, $v_j$) then
12:        isMoving ← True
13:      end if
14:      j ← j + 1
15:    end while
16:    i ← i + 1
17:    j ← i
18:  end while
19:  if not isMoving then                                   {Convergence test}
20:    break
21:  end if
22:  for all v in V do                                          {Update}
23:    v.pos ← v.pos + v.forceVector
24:    v.forceVector ← 0
25:  end for
26: end for
```

Algorithm 16 calculateForce

```
Input: A pair of vertices, $v_a$ and $v_b$
Output: Updated force vectors for $v_a$ and $v_b$V
1: calculate Manhattan distance vector and Euclidean distance between $v_a$ and $v_b$
2: $u_x$, $u_y$ ← calculate the unit vector between $v_a$ and $v_b$
3: $d_x \leftarrow u_x^{-1} * (\frac{v_a.width+v_b.width}{2.0} + minSeperation)$
4: $d_y \leftarrow u_y^{-1} * (\frac{v_a.height+v_b.height}{2.0} + minSeperation)$
5: forceMagnitude ← seperationForce * (Euclidean distance - min(abs($d_x$), abs($d_y$)))
6: if forceMagnitude < -1 then
7:   if abs($u_x$) > abs($u_y$) then
8:     $v_a$.forceVector.x ← $v_a$.forceVector.x + ($u_x$ * forceMagnitude)
9:     $v_b$.forceVector.x ← $v_b$.forceVector.x - ($u_x$ * forceMagnitude)
10:  else
11:    $v_a$.forceVector.y ← $v_a$.forceVector.y + ($u_y$ * forceMagnitude)
12:    $v_b$.forceVector.y ← $v_b$.forceVector.y - ($u_y$ * forceMagnitude)
13:  end if
14:  return True
15: end if
16: return False
```

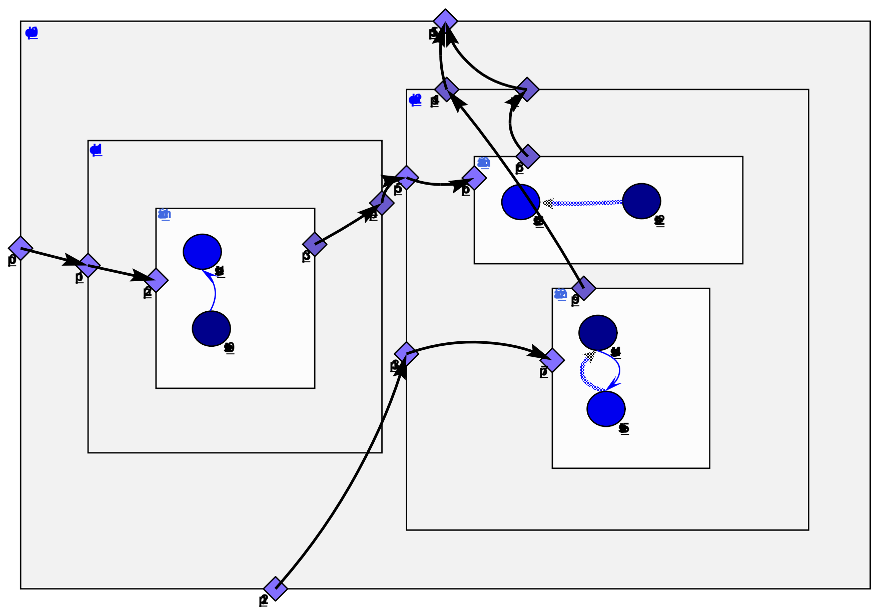


Figure 2.10: Sample model in the DEVS formalism

2.6.3 Analysis

The *force-transfer* algorithm 15, has few loops. The first and last are trivially bounded by O(|V|) and O(50 * |V|) respectively. The innermost loop, which calls the *calculateForce* algorithm 16, is bounded by O(50 * $|V|^2$). Since the *calculateForce* algorithm does not itself have any loops, overall time complexity is simply O($|V|^2$).

2.6.4 Case-study

This algorithm was originally conceived of to “fix” layouts by other drawing techniques that neglected to consider the size of the vertices. However it proves useful when automatically activated whenever a user interactively modifies a model, such as in a class diagram formalism, hence avoiding overlap. Moreover, force-transfer was integrated into a hierarchical DEVS formalism, thus providing overlap avoidance at each hierarchical level. Indeed, as figure 2.10 shows, force-transfer can even be used when no previous layout exists. This typically occurs when a DEVS model is automatically generated. The layout in the figure was produced by a method that successively applied a combination of random, force-transfer, and simple arrow layouts to each level of the hierarchy.

2.7 Tree-like and circle

The tree-like and circle layout algorithms are both fairly simple to implement. The tree-like layout gives good results on graph structures that really are trees. For graph structures that are not trees, that is structures with cycles in them, the layout is often unreadable. This is due to the fact that the algorithm breaks cycles as a preprocessing step. Thus when the edges removed to break the cycles are re-inserted as straight edges between vertices in the drawing phase, they often cross many other edges and overlap vertices.

The circle layout algorithm is particularly inefficient in area usage and thus is best used on subgraphs or small graphs. Alternatively, it makes an excellent preprocessing step for a force directed method such as a spring-embedder. As in the tree-like algorithm, large numbers of edge crossings and edge-vertex overlaps are possible. Since this layout method is not really meant for use on its own, no edge routing techniques were implemented to eliminate the edge-vertex overlaps and reduce the edge crossings.

2.7.1 Pseudo-code

The first step in tree-like layout is to find all the root vertices in the graph. The children of each root vertex is assigned coordinates before the root vertex itself. This recursive process allows each root vertex to be positioned precisely centered above all its children.

Algorithm 17 Tree-like

```
Input: A graph G=(V,E)
Output: An embedding of G
 1: R ← findRootVertices(V)
 2: maxHeight ← maximum height of all root vertices
 3: $x_{pos}$, $y_{pos}$ ← 0
 4: for all r in R do
 5:    w ← 0
 6:    for all v in r.getChildren() do
 7:       w ← layoutNode(v, $x_{pos}$ + w, $y_{pos}$ + $y_{offset}$ + maxHeight) + $x_{offset}$
 8:    end for
 9:    r.pos.x ← $x_{pos} + \frac{w}{2} - \frac{r.width}{2}$
10:    r.pos.y ← $y_{pos}$
11: end for
```

In pure tree layout, only vertices with no parents are roots. In tree-like layout, roots can also be the result of breaking up a cycle. Hence algorithm *findRootVertices* 18 finds all true roots with no parents first, and marks all their children as visited using breadth first search. Note that the children marker routine, pseudo-code not shown, also sets the children returned by the getChildren() method. Any vertices not thus marked must be in a cycle. Such cycles are broken by picking one of the vertices to be a root and then marking the rest of the vertices from this root using breadth first search. Root vertex picking is done either manually by the user with a mouse click on the canvas or automatically. The automatic technique sorts vertices by decreasing out-degree. It then greedily selects vertices until all the vertices have been marked.

The final step of tree-like layout is shown in algorithm *layoutNode* 19. This is very similar to the root node layout. Vertices with no children are assigned coordinates immediately since they have no dependencies. Vertices with children first make a recursive call that assigns coordinates to the children, and only then are assigned coordinates themselves. This allows for parent vertices to always be centered and above their children.

The entire pseudo-code for circle layout is given in algorithm *circle* 20. As a preprocessing step, all vertices are sorted topologically. This sorting is constructed using depth first search in linear time. This step is important since it forces edges along the perimeter of the circle. By comparison, a random vertex sort or vertices sorted by degree, both result in a large number of confusing edge crossings near the center of the circle.

The first step after preprocessing is calculating the perimeter of the circle[2]. The perimeter should be large enough to fit all the vertices in the graph. Therefore, assuming vertices have rectangular bounding boxes, calculating the diagonal distance of a vertex is equivalent to its bounding circle diameter. Since space must be left for edges to be drawn between vertices, an extra amount of

[2]The perimeter of a circle is usually called its circumference

Algorithm 18 findRootVertices

```
Input: A set of vertices V
Output: A set of root vertices R
 1: R ← {}
 2: for all v in V do
 3:     if v.indegree = 0 then
 4:         R ← v
 5:         markAllChildrenBFS(v)
 6:     end if
 7: end for
 8: cycleVertices ← {}
 9: for all v in V do
10:     if not v.marked then
11:         cycleVertices ← v
12:     end if
13: end for
14: while cycleVertices not empty do
15:     R ← chooseRootVertex(cycleVertices)
16: end while
```

Algorithm 19 layoutNode

```
Input: A vertex v, $x_{pos}$ coordinate, $y_{pos}$ coordinate
Output: New position for v
 1: if not v.hasChildren() then
 2:     v.pos.x ← $x_{pos} + \frac{v.width + x_{offset}}{2}$ - $\frac{v.width}{2}$
 3:     v.pos.y ← $y_{pos}$
 4:     return v.width
 5: else
 6:     w ← 0
 7:     h ← v.height + yOffset
 8:     for all $v_{child}$ in v.getChildren() do
 9:         w ← layoutNode($v_{child}$, $x_{pos}$ + w, $y_{pos}$ + h) + $x_{offset}$
10:     end for
11:     v.pos.x ← $x_{pos} + \frac{w}{2}$ - $\frac{v.width}{2}$
12:     v.pos.y ← $y_{pos}$
13:     return w - $x_{offset}$
14: end if
```

space, offset, is added to this bounding circle diameter. Adding this up for every vertex gives the perimeter distance of the circle drawing.

The final step is to calculate an interval fraction between 0 and 1. This interval, when multiplied by 2Π, becomes the radian angle used to calculate the vertex positions on the circle. Initially, the interval is based on the last vertex. This ensures the first and last vertices do not overlap. Successive intervals are then calculated according to the current and previous vertices. Thus no overlaps occur over the entire circumference of the circle.

Algorithm 20 Circle

```
Input: A graph G=(V,E)
Output: An embedding of G
 1: obtain a topological sort of V
 2: perimeter ← 0.0
 3: for all v in V do                                                    {Perimeter}
 4:     v.boundingCircleDiameter ← $\sqrt{v.width^2 + v.height^2}$ + offset
 5:     perimeter ← perimeter + v.boundingCircleDiameter
 6: end for
 7: diameter ← $\frac{perimeter}{\Pi}$
 8: interval ← $\frac{v_{|V|-1}.boundingCircleDiameter}{2.0*perimeter}$                  {0 ≤ interval ≤ 1}
 9: for i = 1,...,|V|-1 do                                        {Assign coordinates}
10:     x ← diameter * (1 - sin(interval * 2Π))
11:     y ← diameter * (1 - cos(interval * 2Π))
12:     $v_i$.pos ← x, y
13:     interval ← interval + $\frac{v_i.boundingCircleDiameter+v_{i+1}.boundingCircleDiameter}{2.0*perimeter}$
14: end for
```

2.7.2 Analysis

The *tree-like* algorithm 17 iterates over the root vertices, a subset of R of V. The *layoutNode* algorithm 19 is called |R| times. Recursive calls are made |V|-|R| times to layout children and ultimately the leaf vertices. Finally, the *findRootVertices* algorithm 18 iterates over all the vertices in the first loop. If the vertex has no parents, a depth first search marker is run on the vertex. The total amount of depth first searching done is bounded by |V|. The second loop is also clearly bounded by |V|. Finally the chooseRootVertex() method, automatic version, uses a greedy strategy that is linear in the number of vertices. Hence the overall run-time of Tree-like layout is linear.

For the *circle* algorithm 20, the topological sort is done in linear time using depth first search. The perimeter calculation and coordinate assignment loops clearly run |V| times. Therefore circle layout is also a linear time algorithm.

2.7.3 Case-study

The applicability of the tree-like drawing technique is limited to models of trees or graphs that are very nearly trees. On the other hand, circle layout is useful in any formalism, during interactive editing, when applied to a subset of the vertices that form cycle. Otherwise, circle layout is simply the preprocessing step for the spring-embedder. Figure 2.11, shows the same dependency model in the Generic formalism drawn with tree-like and circle layout.

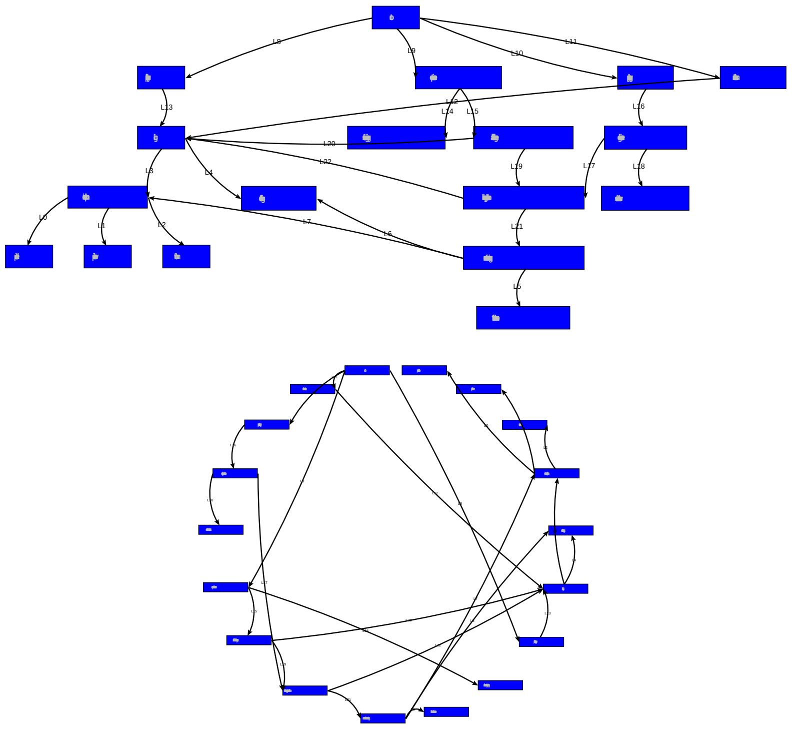


Figure 2.11: Dependency model in Generic formalism

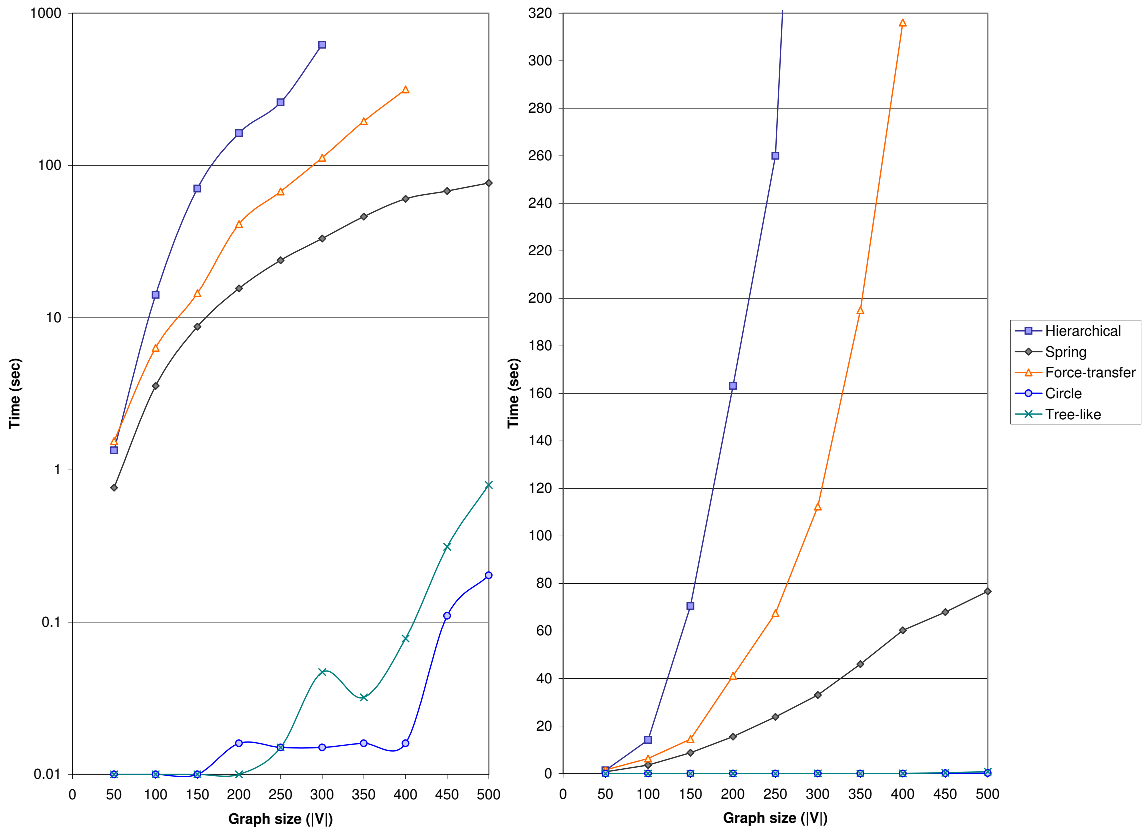


Figure 2.12: Time performance (logarithmic and normal time)

2.8 Experimental time performance

The time performance of all the drawing method implementations, save that of linear constraints, are shown in figure 2.12. Note that on the left, the vertical axis represents log time to effectively capture the wide variations in time performance, whereas time is incremented normally on the right to make linear behaviour more obvious. The tests were performed on randomly generated connected graphs ranging in size from small to large. The graph size is equal to the number of vertices and to one and a half times the number of edges. The graphs were constructed thus on the assumption that a model created by a user would not have many more edges since they would quickly clutter the graph and make it difficult to understand the underlying problem. Conversely, a graph with fewer edges is easier to compute a layout for.

The times shown in the graph represent only the real-time elapsed while the layout algorithm was executing. In more detail, time was measured only while the layout algorithm was working on the abstract graph representation. Thus time required to generate the abstraction or redraw the canvas is not included. To avoid misrepresenting the running-time with a random graph that just happens to be particularly good or bad for a given algorithm, five random graphs were generated for each size. Therefore, the running-times shown in figure 2.12 are actually the median time of the five trials. All tests were conducted on a 3.2 GHz P4 processor with hyper-threading enabled, thus CPU utilization by the layout algorithms was at most fifty percent.

The results for circle and tree-like layout are gratifying. They indicate that even for large graphs, only a fraction of a second is necessary to compute the entire layout. This is particularly useful in the case of circle layout, since it is highly recommended as a preprocessing phase for the spring-embedder.

It is far more difficult to judge the performance of the force-transfer algorithm. This algorithm is highly dependent on the initial layout. In the experiments, the initial layout was arbitrarily constructed as a straight-line of vertices oriented to the south-east. Each successive vertex in the line overlapped its predecessor by approximately ninety percent. As expected, the algorithm yielded quadratic asymptotic behaviour. Thus the most important improvement to this algorithm algorithm would be to augment it to avoid computing the forces between each pair of vertices by eliminating distant vertices from consideration.

The spring-embedder results are quite surprising in that they are linear. Yet the algorithm is clearly quadratic. The explanation for this lies in the preprocessing step of circle layout. Without this step, the time results are indeed a quadratic curve. Hence the use of the preprocessing step is justified for three reasons: circle layout is fast, it improves the final drawing quality, and it improves the time performance of spring layout. This algorithm would greatly benefit from a re-implementation in a non-interpreted language. This might well make the algorithm fast enough that the delay is nearly imperceptible to the user throughout the range of medium sized graphs.

Finally, the results for the layered drawing technique, hierarchical layout, are puzzling. Initially, the time-usage indicates very poor asymptotic behaviour, as one might expect from an algorithm with a worst-case time complexity of $O(|V|^4)$. Yet between graph sizes of 150 to 250, the asymptotic behaviour is strictly linear. It is not clear at this time why this might be and deserves closer investigation. One important factor is the Python language. Indeed, with a graph size of 300, fifty seconds were required to compute the breadth-first search layering, but with 350 vertices no result was forthcoming after ten minutes. Hence this drawing technique would definitely benefit from a re-implementation in a more appropriate language.

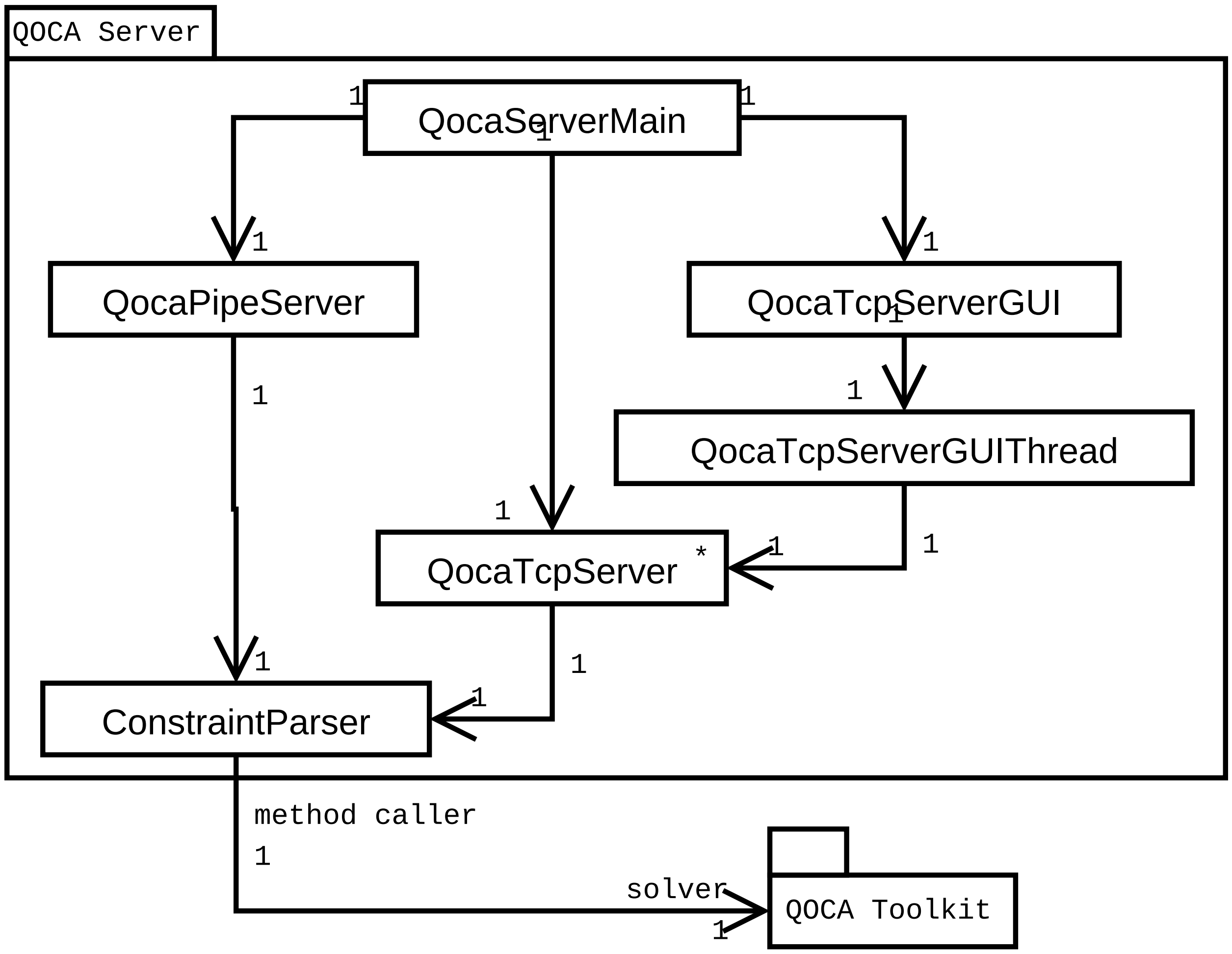


Figure 2.13: QOCA server class diagram

2.9 Linear Constraints

Linear constraints provide a declarative approach to layout. On the one hand, this allows developers with no special background in layout algorithms to craft specific layout behaviours. On the other, layout constraints are rather inflexible, limited in what the layout aesthetics they can satisfy, and very difficult to hybridize with other drawing techniques.

Implementing a constraint solver is a very large undertaking in its own right. Hence the readily available QOCA constraint solving toolkit was integrated into AToM^3. The following subsection describes the integration of the toolkit. Thereafter, experiences with the use of linear constraints in AToM^3 are presented in subsection 2.9.2.

2.9.1 Design

The integration of the QOCA constraint solving toolkit into AToM^3 is composed of a server and client module. The server module, written in Java, directly interfaces with the QOCA solver. In brief, the server module send commands to the QOCA library on behalf of the client and returns results when applicable.

The server is started by executing the QocaServerMain class. Depending on the command-line arguments, either a GUI is shown to the user or a pipe based or TCP/IP based server is started. The GUI is equivalent to using command-line arguments. The pipe based server, QocaPipeServer,

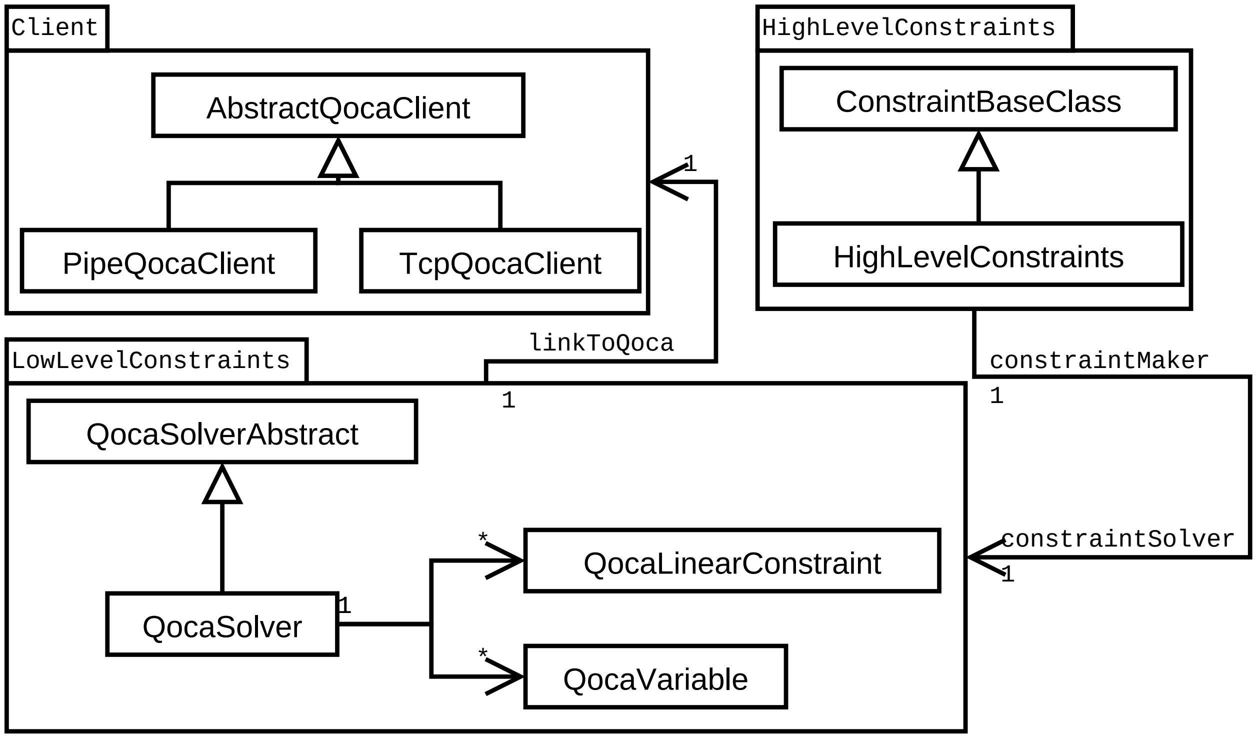


Figure 2.14: QOCA client class diagram

reads standard input (stdin) and uses the ConstraintParser to convert the textual commands into method calls. If the method call is a solve, then it writes out the change variables and their values to standard output (stdout). The TCP/IP server, QocaTcpServer is similar to the pipe version, except that it is multi-threaded to handle as many clients as necessary. Unit-tests verifying the correct function of both server types were performed.

The client portion is written in Python to smoothly interface with AToM^3. The AbstractQocaClient ensures that both the pipe and TCP/IP clients support the same low-level functionality, including: connect, read, write, and disconnect. An internal representation of the solver, QocaSolver, is used to keep track of the variables and constraints. It also generates/parses textual commands that are written/read by the low-level client. The ConstraintBaseClass provides six high-level constraints, such as a one dimensional offset constraints. This is accomplished by creating all the necessary low-level constraints, composed of QocaLinearConstraint and QocaVariable instances. The HighLevelConstraints class further specializes the ConstraintBaseClass, by allowing for the creation of of over fifty high-level constraints including insideness and overlap. Hence linear constraints need not be explicitly created, instead a method call to the appropriate high-level constraint is often all that is necessary. As with the server, each component of the client was thoroughly unit-tested independently of the AToM^3 application.

2.9.2 Linear constraints and AToM^3

To verify the effectiveness of linear constraints in the AToM^3 tool, a toy formalism was created. This formalism, as figure 2.15 reveals, is inspired by PacMan. The formalism has 5 different nodes/vertices: the scoreboard, square grid blocks, PacMan, ghost, and food. Relationships are defined between blocks and PacMan, ghost, and food, between blocks and other blocks, and between scoreboards and a block. For each relationship, there is at least one high-level constraint generated.

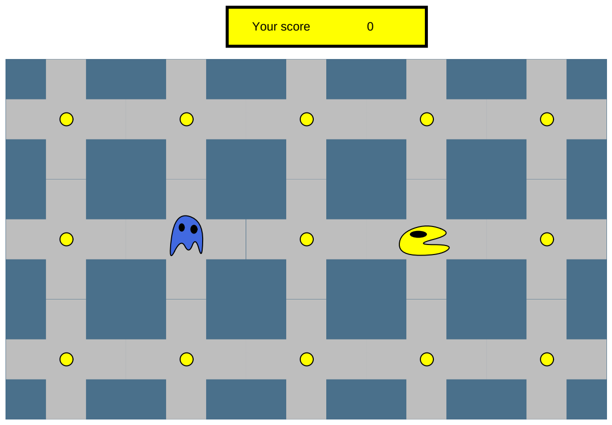


Figure 2.15: Example model in the PacMan formalism

The simplest of the constraints is that between a block and the PacMan, ghost, or food upon it. This constraint forces the two entities to be centered with respect to each other. The constraint between blocks and other blocks is one of four possibilities: to the left of, to the right of, to the top of, and to the bottom of. Finally, the scoreboard is constrained within at least a few pixels above the top-center block, at most several dozen pixels above it. It is similarly constrained horizontally.

A simulation graph grammar was also constructed. It provides a working demonstration of a graph transformation with correct layout behaviour in the host graph. For example, when a rule in the graph grammar alters PacMan's relation to the block he currently sits on to another block, PacMan's icon actually moves to the center of that other block.

Beyond the world of PacMan, linear constraints could be very useful for assembling icons. For example, in a UML class diagram, a class is composed of three boxes, one for each of: class name, attributes, and methods. Linear constraints would be effective in keeping those boxes aligned and appropriately scaled. Unfortunately, at this time the integration of linear constraints with AToM^3 has not reached that component dealing with visual icons.

3 Formalism-Specific UI and Layout

A key component of visual modeling, the visual modeling environment, has been neglected in the literature to date. Yet the construction of new domain-specific formalisms and multi-formalism environments are important for modeling today's complex systems. The status quo of hard-coding the environment is inadequate due to the inflexibility and bug-prone nature of this approach. Hence a new framework for explicitly modeling the reactive behaviour of the visual modeling environment, including formalism-specific behaviours and layout considerations, is developed.

At a high level of abstraction, the new framework works as follows. First, a model of generic user-interface reactive behaviour is constructed. The code generated from this model is generally applicable to most formalisms (*i.e.*, it is formalism independent). Thereafter, each formalism provides additional models that refine the generic UI behaviours with more specific behaviours. Using visual cues (bounding boxes), the correct formalism-specific or generic behaviour is chosen even in the presence of multiple simultaneous formalisms. Finally, the formalism-specific behaviours themselves include automatic graph layout method invocations in specific sequences and appropriate times.

Section 3.1 provides the motivation and background for this approach. Section 3.2, describes the architecture of this approach. This includes a generic user-interface behaviour model, in the statechart formalism, that can be further refined by formalisms. Section 3.3 then presents a case-study for this approach, in the form of the DCharts formalism. DCharts is an existing formalism, developed in Thomas Feng's M.Sc. thesis [Fen04]. Finally, section 3.2 describes all the DCharts formalism-specific user-interface behavioural models. Ultimately, these models are used to precisely determine when, how, and to what the automatic layout methods described in chapter 2 are applied.

3.1 Motivation and background

This section provides motivation for using multiple formalisms. Indeed, multiple formalisms create user-interface behaviour demands that are hard to satisfy with traditional approaches. At the same time, the concepts of meta-modeling and domain-specific formalisms are summarized below. Finally, the existing concepts that the proposed solution to formalism-specific UI reactive behaviour combines are described. These concepts are: nested and zooming user-interfaces, nested events in graphical user-interface libraries, and scoping in programming languages. Descriptions of them and exactly how they relate to formalism-specific UI behaviour can be found in subsections 3.1.2, 3.1.3, and 3.1.4 respectively.

3.1.1 Domains, Formalisms, and Meta-models

In recent years, various software applications were developed to support modeling of complex systems, particularly in the software and physical systems domains. Modeling is essential to analyse and design such complex systems. A model provides abstraction, which can dramatically increase the understanding of the represented system. Often, a problem can be modeled with multiple formalisms, depending on the viewpoint taken or on the aspect of interest. For instance, in the software domain, the structure of an application is better viewed as a UML class diagram whereas the dynamics of object interaction are better modeled as a UML sequence diagram.

Having the possibility to view the solution of a problem from different angles, some more abstract than others, helps the developers understand, modify and possibly re-use problem solutions. Moreover, specific formalisms are better suited to model specific systems.

Modeling a problem with multiple formalisms is good, but it is not enough to use only existing formalisms. It should be possible to design formalisms which are maximally constrained to model and solve problems in a specific domain. Such an approach has several advantages. First of all, the modeler has a specific mental model of the problem, and the closer the formalism is to this model, the easier model development will be (from a cognitive point of view). Also, if the formalism is very close to the domain, the human modeller is constrained to construct only models in that domain and is hence less likely to make errors. More importantly, the modeler can *abstract* away from *how* the model is executed and verified by the computer and focus on the domain-specific problem at hand.

Domain-specific modeling does not only bring abstraction, but also verification and execution. If a system is represented in a formalism with known formal properties, one can automatically infer the system behaviour, at least up to a certain point. For example, in [VdL04], a simple road system was modeled in a domain-specific formalism, which was transformed into Petri Nets. Since Petri Nets have known formal semantics, the transformation specifies the behaviour of the traffic system formalism. Then, from the Petri Nets model, a reachability graph was generated to assert the non-occurrence of deadlocks in the traffic system.

AToM^3, A Tool for Multi-formalism and Meta-Modeling, was developed in the Modeling, Design and Simulation Lab (MSDL) of McGill University by Juan de Lara and Hans Vangheluwe [dLV02a]. This tool enables domain-specific multi-formalism modeling by means of meta-modeling and graph transformation. Complete software applications for creating models in a domain-specific formalism are synthesized from meta-model specifications.

The AToM^3 tool was proven to be very powerful, allowing the meta-modeling of known formalisms such as DEVS [PB03], Statecharts [BV03, Fen03], UML Class Diagrams and Activity Diagrams [dLV05], Finite State Automata [VdL02], Petri Nets [dLV02b], GPSS [dLV02d], Process Interaction Networks [dLV04], Hybrid Systems [LJVdLM04, dLGV04, dLVAM03], Causal Block Diagrams [PdLV02, dLVA04] (a subset of Simulink), Dataflow Diagrams [dLV02c] and many others. More importantly, many new domain-specific formalisms were constructed using the tool, such as the Traffic formalism [JdLM04].

The philosophy of AToM^3 is to *model everything* explicitly. Hence, all four aspects of a formalism are explicitly modeled in AToM^3. The first aspect is the abstract syntax of a formalism. For example, abstract syntax is what specifies that a UML class diagram formalism is composed of classes and relationships in the form of associations and inheritance. The second aspect is the concrete, possibly visual, syntax. For example, a UML class diagram is represented by a rectangular box. These first two aspects are both static in nature. Thus, they are meta-modeled using either the *Entity Relationship* or *Class Diagram* formalism.

The last two aspects of a formalism, the semantics and the reactive behaviour of the visual modeling environment, are dynamic in nature. In AToM^3, graph transformations are often used to explicitly model the operational or denotational semantics of a formalism.

The most crucial aspect of a formalism in the context of this thesis, is the reactive behaviour of its modeling environment. The reactive behaviour specifies how a given sequence of input events, such as from the mouse and keyboard, influence the state and future behaviour of the modeling environment. For example, simultaneously pressing shift, control, and c in a UML class diagram formalism might create a new class. In most existing visual modeling tools, this is dealt with by having a single hard-coded behaviour. It is thus possible to manually construct a user-interface with the appropriate reactive behaviour without too much difficulty. However, in applications that support multiple formalisms the complexity increases, particularly if the different formalisms are dissimilar. Even worse, from the application developer's point of view, is support for multiple formalisms at once in a single diagram. As shall be shown in subsequent sections, the use of formalism-specific behaviour statechart models instead of hard-coding can greatly improve the situation.

3.1.2 Nested and zooming user-interfaces

A nested graphical user-interface is one where widgets, reactive visual components such as buttons and windows, are recursively nested. A very familiar example of a nested GUI is that of a menu system. Menu commands are placed inside broad categories, such as File and Edit, and potentially refined further by hierarchical sub-menus. In the case of menus however, the displayed widgets are restricted to a simple text label.

A zooming user-interface is one where the density of the information shown to the user can be scaled. A nice example of this from real-life is reading a newspaper. When searching for an interesting article one pushes the newspaper away, zooming out to get an overall view. After finding an interesting article, one pulls the newspaper in closer to read it more easily.

These two concepts of nested and zooming user-interfaces are combined in [PM99]. The system makes it possible to use general widgets, including directly user-modifiable canvas widgets, in a zooming, nested hierarchy. Hence, at the highest hierarchical level, the widgets provide controls that are few and very coarse in effect. These high level widgets do have the advantage that they require very little area on the screen. Successive levels of the hierarchical widgets, provide more and finer controls. They appear in the same general area as their higher level parents, but they use up more screen space, so fewer of them can be seen at once. Thus this system is very advantageous in that it can present very large and layered control problems to the user in a cohesive and readily navigable visual interface.

Both of these concepts prove quite useful in formalism-specific visual user-interfaces. Simply replace GUI widgets with formalism entities that are hierarchical in nature.

3.1.3 Nested events in GUI libraries

A graphical user-interface library provides widgets for interacting with a user. In order to avoid making incorrect generalizations, the following is true at least for the Tk/Tcl GUI library [Ous94]. Widgets range from windows and canvases to simple buttons and labels. Widgets are hierarchically organized and the highest level is called the root, which is simply an instance of the GUI library. Moreover, it is possible to bind method invocations to some triggering event to each widget.

Interesting behaviour occurs when both a parent and child widget, according to the widget hierarchy, bind to the same trigger event. In this case, the child widget is considered the most tightly binding and receives the event first. Thereafter the parent receives it as well. Sometimes this is not the desired behaviour at all. For example, when one presses tab in a text widget, one means to actually insert a tab, not insert a TAB and then change focus to the next widget. Thus, Tk/Tcl allows a widget to exclusively handle an event, halting it from propagating higher in the hierarchy of widgets. A description of exactly how this is done can be found at `www.pythonware.com/library/tkinter/introduction/events-and-bindings.htm`.

Both the idea of binding events at multiple hierarchical levels and explicitly handling events prove useful in formalism-specific user-interfaces. Again, GUI widgets are replaced by their hierarchical formalism entity counterparts.

3.1.4 Variable scoping in programming languages

Most high-level programming languages provide means of specifying the scope of a variable. The scope of a variable is usually denoted using a pair of braces. Furthermore scopes are inherently hierarchical. The scope of a variable is used by the compiler to bind it to its declaration. If the compiler finds an occurrence of the variable in a given scope, it searches that scope first for its declaration. If it fails to find it, it searches successively higher levels in the scope hierarchy until reaching the global variable space.

The link to formalism-specific user-interfaces in this case requires using the bounding boxes of visual formalism entities as scope delimiters. The variable becomes an event and the building process the main event loop which passes the event to each hierarchical scope level in succession.

3.2 Formalism-specific UI

This section describes the architecture of our new approach to modeling the reactive behaviour of formalism-specific user-interfaces. The goal is not to create an entirely new specification of user-interface behaviour for each formalism, but rather to modify it to suit special requirements. Hence, it makes sense to have a single, generic, application-wide specification of the user-interface behaviour. This is described in the following subsection. The formalism-specific modifications to this generic specification are then made through the entities of the formalism. That is, for each entity of the formalism that has specific user-interface requirements, a small UI specification is created. This entity may be quite artificial, such as an entity created for the sole purpose of enclosing the usual formalism entities. This is described in subsection 3.2.2. Finally, since some events affect formalism entities but don't occur within the enclosure of an entity with a UI specification, pre/post UI observers are used to bridge the gap, and are described in subsection 3.2.3.

3.2.1 Generic UI behaviour

The generic user-interface behaviour for the entire AToM^3 application is modeled using statecharts [Har87]. This model is shown in figure 3.1. The advantage of modeling the UI behaviour cannot be understated. The model may look complex, but it is easy to trace what happens given a certain sequence of events. If the user-interface were hard-coded however, it would be far more difficult, even with just a small subset of the hundreds of events this model handles. Moreover, a small change in the behaviour specification may require drastic modification of code, but only a minimal change in the statechart model.

The generic UI behaviour statechart works as follows. When instantiated, the "Initial" state in the top-left is active. It is then initialized with a "Start" event that places it in the "Active Event Loop" composite state. Within this composite state, the "Main" state is the default, so this is the one that becomes active. Notice how many of the events can be directly handled from this "Main" state. Although only a few arrows are shown doing this, there are in fact many arrows super-imposed upon one another to reduce visual clutter, and the labels were then drawn above and below the arrows.

The more complex event sequences dominate the right side of the model. Most of these sequences are similar, so only two of them shall be described. These are the event sequences necessary for selecting entities and for creating a new arrow interactively.

An additive selection consists of holding down a key and a mouse button, moving the mouse around, and then releasing the mouse button. While the selection is in progress, it would not make sense to do something else, such as delete everything on the canvas. Indeed, this is impossible in the model, since the very first action, the key-button combination, puts us in the state "Add To Selection". Only two events are now accepted: mouse movement and the mouse button release. Hence there can be no confusion as to the user-interface's behaviour in this situation.

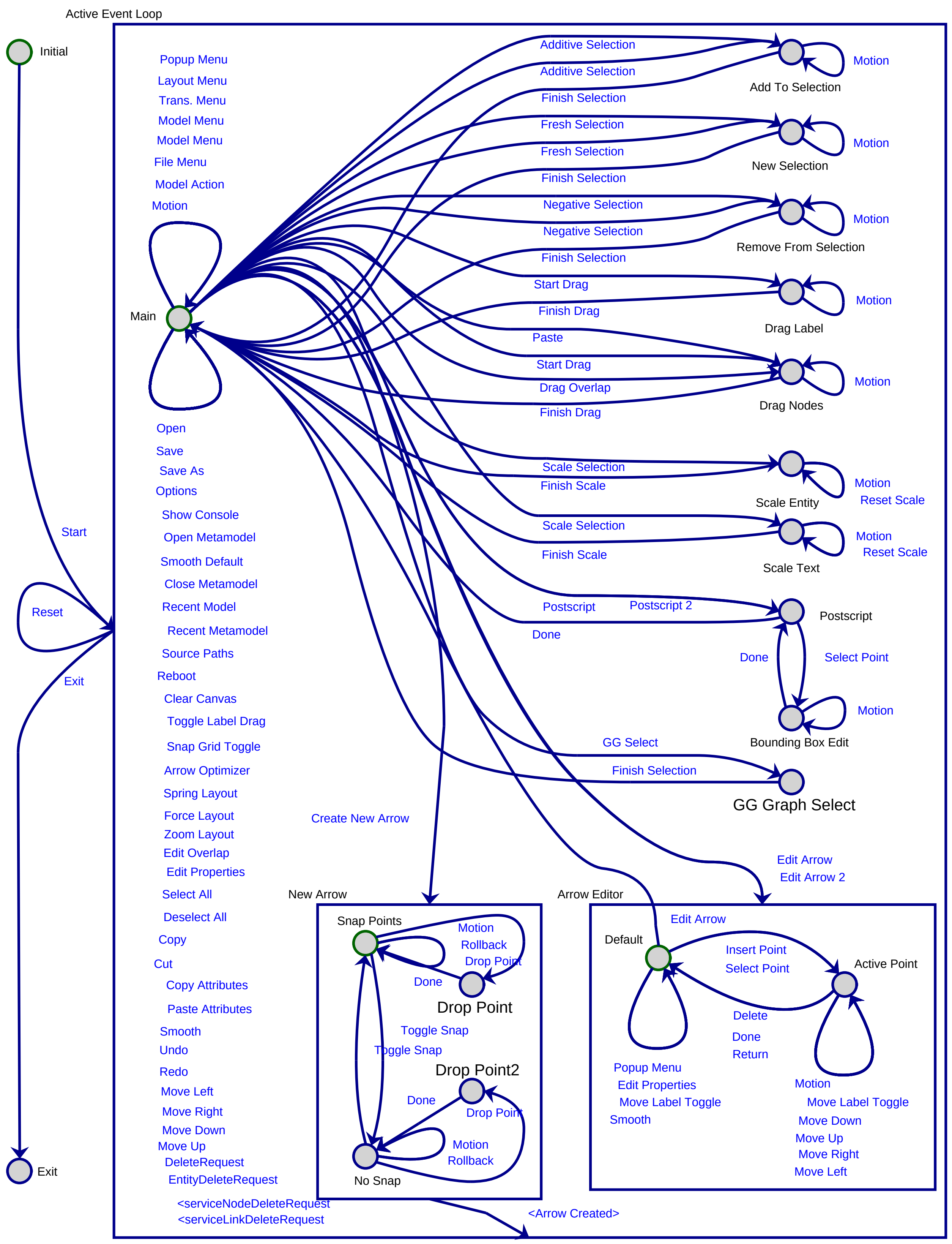


Figure 3.1: Generic user-interface behaviour statechart

Creation of a new arrow occurs when the user holds down the control key while pressing the left mouse button to generate the “Create New Arrow” event. The “New Arrow” composite state and the composite’s internal default state “Snap Points”, then become active. The user can now generate four events: “Motion”, “Drop Point”, "Rollback", and “Toggle Snap”. The first event simply triggers code to redraw the arrow being created whenever the user moves the mouse. The second event temporarily enters the “Drop Point” state. Entering this state triggers code to either complete the arrow, if an entity is currently under the mouse cursor, or to add a control point to the arrow. If the arrow was completed then either the “<Arrow Created>” or “Reset” event is generated, depending on whether the target entity was valid, and the active state becomes “Main” again. Alternatively, if a control point was added, then the event “Done” is generated. This returns “Snap Points” to the active state.

The third event, “Rollback”, allows the user to remove previous control points until the new arrow itself is removed. The latter situation also results in the generation of the “Reset” event and a return to the “Main” state. Finally, the “Toggle Snap” event simply allows the user to switch between dropping control points on grid points or at arbitrary locations. Hence the action code in the transitions of the state “No Snap” mirror those of the transitions in “Snap Points”, except that different parameters are used.

3.2.2 Formalism-specific behaviour

The purpose of formalism-specific reactive behaviour models is to modify only those portions of the generic user-interface behaviour that require special handling. There are two good reasons for not simply copying the generic UI behaviour in its entirety and then modifying it for a specific formalism. One reason is that it would be needlessly difficult to figure out, by looking at the behaviour model, what the formalism-specific requirements were. A second reason lies in the fact that the generic UI behaviour may evolve over time. Assuming a likely scenario where the evolutions are quite minor and that the formalism-specific behaviour is in fact very specific, then the formalism can benefit from evolutions to the generic UI behaviour without modification itself.

Implementing formalism-specific behaviour requires determining where (i.e. within which scope) events actually occur. This is easy, since an application’s main event loop receives all user input events, including keyboard input, with mouse coordinates attached. Using this, the basic idea is that if those coordinates lie within the scope of a formalism, then that event should be sent to that formalism. However, two questions quickly arise: what is the scope of a formalism and what if an entity within a formalism would benefit from defining its own UI behaviour?

The answer to these questions lies in using the entities of a formalism itself to define the scope of UI behaviour. For example, a class is an entity of a UML class diagram formalism. Continuing with this example, a formalism-wide scope could be defined by adding an extra entity that contained all the classes in the formalism. Recall that the formalism-specific behaviour approach is a combination of hierarchically nested graphical user-interfaces, the nested event propagation found in graphical user interface libraries, and the scoping behaviour of variables common to most programming languages. To see how this can be, first replace the widgets, such as buttons, of a nested GUI with formalism entities. Then replace the event propagation through the widget classes of a GUI library with event propagation through the nested formalism entities. Finally, replace the scope of a variable typically defined by a pair of matching braces in programming languages with the visual bounding box of a formalism entity. Hence, instead of a simple button widget receiving a mouse-click, any input event can be captured by a formalism entity’s scope and then propagated across multiple entities, unless of course that entity’s scope was local.

Hence, formalism-specific behaviour requires the addition of an “artificial” entity whose sole purpose it to define the scope of the formalism. This artificial entity hierarchically and visually contains

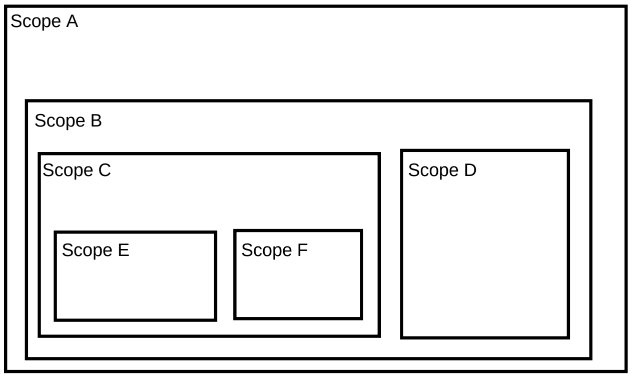


Figure 3.2: Hierarchical scopes

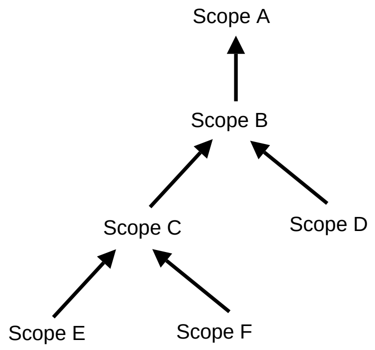


Figure 3.3: Nested event propagation

all the other entities of a formalism. The hierarchical aspect refers to the fact that the artificial entity is the parent of all the other children entities. If the formalism has hierarchical entities itself, then these entities are automatically considered as participating in the scoped behaviour hierarchy. Alternately, more artificial entities can always be used if further refinements to the visual modeling behaviour were deemed useful.

Using an internal data-structure representing hierarchy, it is now a simple matter to send nested events to only those entities that the event enters the scope of. To illustrate this, refer to figure 3.2 and consider an event, “fubar” occurring inside the box “Scope F”. Since the event “fubar” occurred within the bounding box of “Scope F”, the entity represented by “Scope F” receives the event “fubar”. The event is then propagated along the nested hierarchy according to figure 3.3. Therefore the behaviour statecharts associated with the entities “Scope C”, “Scope B”, and “Scope A” each receive the event in succession. Note that “Scope A” is actually the application itself and events sent to it go to the generic UI behaviour statechart. Moreover, “Scope B” is a container entity for an entire formalism, say formalism X, one of potentially many concurrently active formalisms. The remaining scopes are all entities of this formalism X that happen to be hierarchical in nature. Scoped entities are neither required to have an associated behaviour statechart, nor to deal with every event they receive if they do have one.

A scoped entity receiving an event can of course handle the event by taking some action. It is also necessary, however, that it be able to stop the event from propagating to higher levels of the nested hierarchy depicted in 3.3. This idea of preventing events from propagating up the nested hierarchy is similar to that seen in many graphical user interface libraries such as Tcl/Tk. For example, imagine the generic UI statechart allowed one to move all selected entities by 1 pixel to the right with each right-arrow key-press. If the scoped entity then tries to map the right-arrow key-press event to display a dialog box it would hardly make sense to both display the dialog box and to move everything to the right. Therefore the behaviour statechart of an entity can set a flag on the event itself to indicate to the main event loop whether or not to keep propagating an event. Note that the event is an object and that this particular mechanism is safe in a multi-threaded event loop.

Another consideration is that a complex event sequence may be initiated in a given scope level but subsequent input events required to complete the sequence occur outside of the scope level. This effectively “freezes” whatever the user was doing until they return the mouse cursor back to the appropriate scope level. This may be the desired behaviour in some cases. In others though, such as if we wish to input an event sequence to create arrows, it is not the desired behaviour. This is clear since arrows may start in one scope, but in the process of drawing control points before reaching the target of the arrow, it is easy to leave that original scope. Therefore it becomes necessary to introduce the concept of "locks". A behaviour statechart associated with a given entity can simply lock the event loop, effectively transferring all events to itself. The locking entity is also responsible for releasing the lock. Note that this locking behaviour is analogous to that of modal windows/dialogs sometimes encountered in user-interfaces. The most common use of a modal dialog is to force a user to respond to a question before they can resume operating an application. This guarantees the application will not enter an ambiguous state.

Finally, it is possible for an event to occur outside of a scoped entity that nonetheless affects it. For example, one could select some entities on the canvas. Later one hits the delete key. The coordinates of the delete event can occur anywhere, yet the event very dramatically affects the selected entities. This is dealt with in the next subsection.

3.2.3 Pre/post UI observers

Pre/post user-interface observer statecharts are necessary to catch "external" events with internal effects. In other words, some events can occur at any coordinate on the canvas and yet have an impact far away from that location. For example, one could put the mouse cursor outside the application window, and press the delete key. The effect is to delete all the previously selected entities, but since the event coordinates are outside of any entity, no scoped behaviour is possible for the delete event.

To overcome this difficulty, a pre UI statechart an observes event before the generic UI behaviour statechart acts on it. Similarly, the post UI statechart observes the event after the generic UI behaviour statechart. This observation is sufficient to enable propagating an event, such as delete, directly to the behaviour statecharts of all affected entities. Note that if an event is handled by an entity's behaviour statechart or a statechart lock is in effect, neither the generic UI behaviour statechart nor the pre/post statechart receive the event. Also pre/post statecharts are "observers", since they cannot "handle" events as can the formalism-specific behaviour statecharts. If they could "handle" events, then conflicts would quickly arise in a multi-formalism environment. In subsection 3.4.3, observer statecharts are further illustrated by means of an example.

3.2.4 Behavioural conformity

The proposed system for modeling the reactive behaviour of formalism-specific visual modeling environments, raises the issue of behavioural conformity. Structural conformity is easy to define; if a class B inherits from a class A, then B must have all of A's features. Behavioural conformity is much more difficult. Thus although B inherits from A, nothing prevents B form behaving completely differently from A. As David Harel and Eran Gery explain in [HG97], even statechart behaviour models cannot guarantee that the behaviour of B is not radically different from A's. Nor does B starting with A's statechart solve the problem.

This issue, with respect to the proposed system, means that if B is the outer scope, A is an inner scope, then B should either refine or add to C's behaviour. For example, suppose B decides to change C's behaviour for the right-arrow key from moving entities right by one pixel to displaying a dialog box. This would be confusing and unexpected to the user. If instead the right-arrow key is modified to moving things by 20 pixels at a time or if a completely new key input is used, there is no problem. Unfortunately, we are not aware of any methods for ensuring behavioural conformity at this time, so it is up to formalism developers to maintain it.

3.3 Case-study: DCharts formalism

To demonstrate the usefulness of explicitly modeling UI behaviour, including layout, a visual modeling environment for the DCharts formalism was re-created. DCharts, a formalism created by Thomas Feng [Fen04], is a combination of DEVS (Discrete EVent Systems specification) and UML statecharts. Both DEVS and UML statecharts can be mapped to DCharts, so in terms of expressive power DCharts is at least as powerful as these two. The DCharts formalism is described in greater detail in the following subsection.

A new visual DCharts formalism is warranted due to a number of deficiencies in Thomas Feng's AToM^3 implementation. Note that none of these "deficiencies" prevented the use of the existing DCharts formalism to model, simulate, and generate code for all the statecharts used in this thesis. Nonetheless, the most serious of these deficiencies is the fact that critical information, such as triggers and actions, are not shown in the model outside of the editing dialogs. For a formalism whose importance lies in being quickly understood and serving as documentation, this is an important point. On the other hand, too much information can be worse than too little if it cannot be conveniently fit on a monitor or printed page. Thus, the new formalism must be able to hide and show information as the user demands (i.e. support zooming). However, allowing users to dynamically modify the displayed information will alter icon sizes, thus increasing the need for automatic layout.

A second serious deficiency in the original implementation is the lack of automatic layout. Because statecharts have a hierarchical structure general purpose layout methods cannot be directly applied to them. In other words, unless a layout method is constructed for the purpose of dealing with compound graphs, graphs with hierarchical containment, the layout method will be of no use. Without automatic layout support there is a significant risk that modelers will avoid adding to an existing statechart model simply because the task of doing the layout manually is so time-consuming. An even worse situation occurs when a domain-specific[1] model is transformed, via a visually specified graph grammar, into the DCharts formalism. In this case, the generated DCharts model carries no graphical information, such as coordinates and size of states, thus the result is essentially a random layout.

A third problem in the old DCharts formalism is that it allows users to construct statecharts that are illegal at the abstract syntax level (i.e. that do not adhere to the meta-model). This is particularly devastating for novice modelers. For example, one can have a default history state floating around, unconnected to anything.

A fourth problem lies in the fact that a number of elements of the non-visual DCharts formalism were never actually implemented in the visual formalism. This includes the final state, sub-model importation, transition priorities, a Statemate statechart compatibility mode switch, and the ability to set code for macros, an initializer, interacter, and a finalizer.

A final, if trivial motivation for a new formalism lies with multiple arrows starting and ending on the same state. This is particularly evident in the generic UI behaviour model, where a large number of arrows were super-imposed one on top of the other to save space. Not only is this very time-consuming for the modeler, it takes a significant amount of time to compute and draw splines, thus increasing load times (spline computation and drawing might be less of an issue in an efficient implementation; alas AToM^3, which uses Tkinter for graphics, is not so efficient). A better approach is to simply draw just one spline with the capability of representing any number of arrows.

Thus, the primary goals of the new DCharts formalism are to display as much information as possible, handle time-consuming layout for the user, and prevent (or warn) the user from creating illegal syntactic constructs. The best way of accomplishing all this is with modeling.

[1]A domain-specific model is a model in a formalism that closely resembles the original problem. Modeling traffic in a "Traffic" formalism is a pertinent example, whereas modeling traffic in a general DEVS formalism is not.

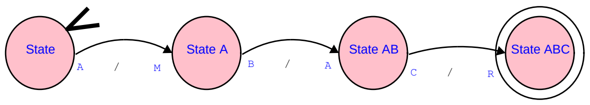


Figure 3.4: Example model in the FSA formalism (layered layout used)

3.3.1 DCharts formalism overview

The DCharts formalism is a combination of DEVS (Discrete EVent Systems specification) and UML statecharts. The author of the formalism is Thomas Feng and the best source of information is his Master's thesis [Fen04]. DCharts share the same abstract syntax as UML statecharts, thus if the concrete syntax is chosen to match, the two formalisms are visually identical. It is in fact trivial to map an entire UML statechart onto a corresponding DCharts model, although the reverse may not be possible. In any case, the following description will be limited to UML statecharts since this is necessary and sufficient to understand the new visual DCharts formalism.

Defining UML statecharts requires first describing the finite state automata (FSA) formalism it extends. A finite state automaton is very simple, as figure 3.4 shows. It consists of states and transitions between states. One state is the initial state and this is the first "active" state. Each transition has a trigger, which is a symbol. If a symbol is provided to the FSA and the active state has a transition that has that symbol as a trigger, the target state of the transition then becomes the active state. If no such transition exists, then the FSA halts, since it does not recognize the language, the set of input symbols, that were fed to it. The FSA also includes at least one "final" state. If a transition sets the active state to be one of the final states and the input is at an end, then the FSA has accepted the input sequence of symbols.

Although the FSA formalism is powerful enough to define regular expressions, it is woefully inadequate to express conditional transitions, actions, hierarchy, and concurrency. Fortunately, David Harel extended FSA to deal with these in [Har87]. Conditional transitions are simply transitions that will fire on an event only if both the trigger is matched and the conditional statement evaluates to true. Actions are events or code that is executed whenever a transition fires, a state is entered, or a state is exited. Hierarchy is achieved by adding composite states that can contain other states, including more composite states. The inside of a composite state is a FSA in its own right, with its own default state (the equivalent of an initial FSA state). Since transitions can exit and return to a given hierarchical level, it becomes necessary to add history states as well. These history states restore the active state within the composite state when a transition returns to the composite state. Hence history allows overriding the default state. Finally, concurrency is added using orthogonal partitions of a composite state. Each orthogonal partition is a simultaneously executing FSA, each with an active state.

3.3.2 DCharts meta-model

The new DCharts formalism, shown in figure 3.5, was modeled in a class diagram formalism within AToM^3. This formalism is similar to UML class diagrams, in that it has classes with attributes, associations with multiplicities, and inheritance. The main difference lies in the fact that the AToM^3 version allows you to immediately generate a formalism-specific editor, with a generic visual modeling environment, from the class diagram.

The perfectly rectangular boxes in the class diagram become the nodes/vertices in the generated formalism. Each of them gets a name attribute that appears on or near the visual icon in the generated formalism. The nodes and the meaning of their attributes are as follows:

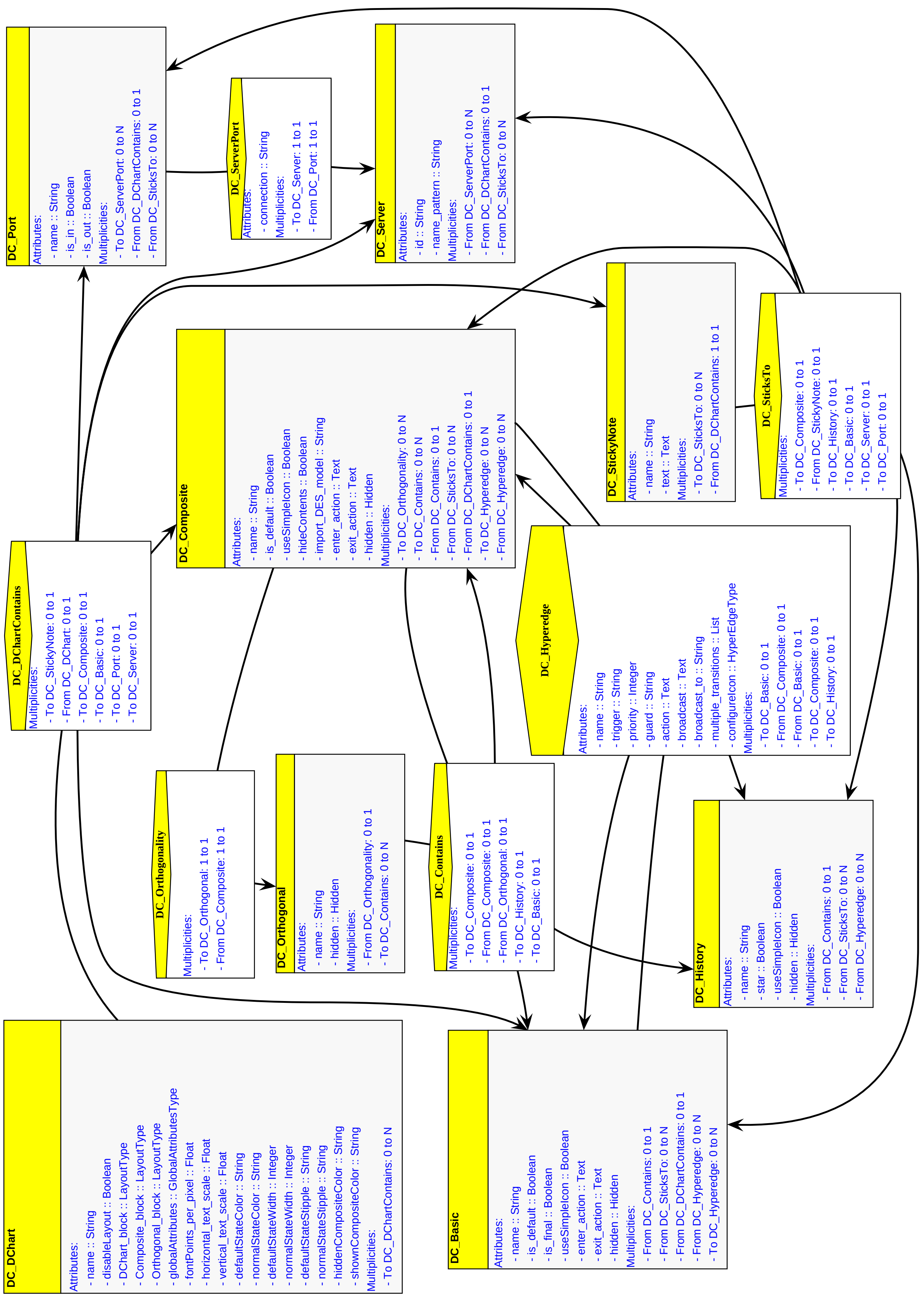

DC_Port
Attributes:
- name :: String
- is_in :: Boolean
- is_out :: Boolean
Multiplicities:
- To DC_ServerPort: 0 to N
- From DC_DChartContains: 0 to 1
- From DC_SticksTo: 0 to N
DC_ServerPort
Attributes:
- connection :: String
Multiplicities:
- To DC_Server: 1 to 1
- From DC_Port: 1 to 1
DC_Server
Attributes:
- id :: String
- name_pattern :: String
Multiplicities:
- From DC_ServerPort: 0 to N
- From DC_DChartContains: 0 to 1
- From DC_SticksTo: 0 to N
DC_StickyNote
Attributes:
- name :: String
- text :: Text
Multiplicities:
- To DC_SticksTo: 0 to N
- From DC_DChartContains: 1 to 1
DC_SticksTo
Multiplicities:
- To DC_Composite: 0 to 1
- From DC_StickyNote: 0 to 1
- To DC_History: 0 to 1
- To DC_Basic: 0 to 1
- To DC_Server: 0 to 1
- To DC_Port: 0 to 1
DC_Composite
Attributes:
- name :: String
- is_default :: Boolean
- useSimpleIcon :: Boolean
- hideContents :: Boolean
- import_DES_model :: String
- enter_action :: Text
- exit_action :: Text
- hidden :: Hidden
Multiplicities:
- To DC_Orthogonality: 0 to N
- To DC_Contains: 0 to N
- From DC_Contains: 0 to 1
- From DC_SticksTo: 0 to N
- From DC_DChartContains: 0 to 1
- To DC_Hyperedge: 0 to N
- From DC_Hyperedge: 0 to N
DC_DChartContains
Multiplicities:
- To DC_StickyNote: 0 to 1
- From DC_DChart: 0 to 1
- To DC_Composite: 0 to 1
- To DC_Basic: 0 to 1
- To DC_Port: 0 to 1
- To DC_Server: 0 to 1
DC_Hyperedge
Attributes:
- name :: String
- trigger :: String
- priority :: Integer
- guard :: String
- action :: Text
- broadcast :: Text
- broadcast_to :: String
- multiple_transitions :: List
- configureIcon :: HyperEdgeType
Multiplicities:
- To DC_Basic: 0 to 1
- From DC_Composite: 0 to 1
- From DC_Basic: 0 to 1
- To DC_Composite: 0 to 1
- To DC_History: 0 to 1
DC_Orthogonality
Multiplicities:
- To DC_Orthogonal: 1 to 1
- From DC_Composite: 1 to 1
DC_Orthogonal
Attributes:
- name :: String
- hidden :: Hidden
Multiplicities:
- From DC_Orthogonality: 0 to 1
- To DC_Contains: 0 to N
DC_Contains
Multiplicities:
- To DC_Composite: 0 to 1
- From DC_Composite: 0 to 1
- From DC_Orthogonal: 0 to 1
- To DC_History: 0 to 1
- To DC_Basic: 0 to 1
DC_History
Attributes:
- name :: String
- star :: Boolean
- useSimpleIcon :: Boolean
- hidden :: Hidden
Multiplicities:
- From DC_Contains: 0 to 1
- From DC_SticksTo: 0 to N
- From DC_Hyperedge: 0 to N
DC_DChart
Attributes:
- name :: String
- disableLayout :: Boolean
- DChart_block :: LayoutType
- Composite_block :: LayoutType
- Orthogonal_block :: LayoutType
- globalAttributes :: GlobalAttributesType
- fontPoints_per_pixel :: Float
- horizontal_text_scale :: Float
- vertical_text_scale :: Float
- defaultStateColor :: String
- normalStateColor :: String
- defaultStateWidth :: Integer
- normalStateWidth :: Integer
- defaultStateStipple :: String
- normalStateStipple :: String
- hiddenCompositeColor :: String
- shownCompositeColor :: String
Multiplicities:
- To DC_DChartContains: 0 to N
DC_Basic
Attributes:
- name :: String
- is_default :: Boolean
- is_final :: Boolean
- useSimpleIcon :: Boolean
- enter_action :: Text
- exit_action :: Text
- hidden :: Hidden
Multiplicities:
- From DC_Contains: 0 to 1
- From DC_SticksTo: 0 to N
- From DC_DChartContains: 0 to 1
- From DC_Hyperedge: 0 to N
- To DC_Hyperedge: 0 to N

Figure 3.5: DCharts Meta-model in the Class Diagram formalism (layered and manual layout)

- DC_DChart is a representation of the entire model. All other entities will be contained by this entity, since it is responsible for providing basic UI handling. It has a boolean that can enable or disable all layout. Disabling layout might be useful for micro-managing the layout by hand for perfect presentation material. The following three attributes are complex. They allow the user to set the layout statechart for DC_DChart, DC_Composite, and DC_Orthogonal respectively. The setting part is implemented as a button that pops up a new instance of AToM^3 that allows creating or editing the layout statechart. When done, the user presses the "OK" button and the statechart is compiled into executable code. When the user finally exits the entire edit dialog, used to modify the layout attribute, the newly compiled statechart code actually replaces the layout statecharts of all the associated entities already on the canvas and of entities created thereafter. In other words, it is possible for a user to modify layout behaviour at run-time by editing a model. Of course within that model, the user could write a multi-thousand line layout algorithm within an action code field. More typically, the user would switch between available algorithms, the sub-algorithms used by a given algorithm, and the parameters fed to these algorithms. The next attribute is the globalAttributes, which allows global DCharts model attributes to be set. These attributes within an attribute are: macros, Statemate algorithm compatibility, initializer, finalizer, and interactor. The remaining attributes of the DC_DChart are simple controls on the visual appearance of the entities in the DCharts formalism. The first three control fitting icons to a block of text while the remaining ones control colors, line thicknesses, and stippling.
- DC_Basic corresponds to a simple state that does not hierarchically contain others. It has the usual statechart attributes: a boolean default or final state indicator, and action code fields triggered by entering and leaving the state. It also has a boolean useSimpleIcon switch to allow the user to zoom the amount of information displayed. In other words, the user can switch from showing the name and action code fields inside a box that fits these or to simply showing a small round icon and just the name. Finally, the hidden attribute is simply a text field automatically constructed from the name and action code and is what is displayed in the non-simple icon.
- DC_Composite is nearly identical to DC_Basic. A major structural difference is that it can contain other states. DC_Composite has two additional attributes that DC_Basic does not have: a boolean hideContents and a string import_DES_model. Both these attributes achieve the same effect of zooming, but in different ways. Enabling hideContents simply forces all the entities contained by the composite state to be invisible. The icon of the composite state is altered to show this. Also, transitions to and from the hidden entities simply appear from the center of the composite state. On the other hand, giving import_DES_model a valid filename means that entering this state results in a new statechart taking control. This allows for true hierarchical decomposition. Finally, the useSimpleIcon attribute only toggles the display of action code in a DC_Composite, rather than also changing the icon to a different shape.
- DC_History is the history state, a device to remember the active state in a hierarchical context. It has a star attribute to indicate whether that history is shallow or deep. A deep history "remembers" the active state of all hierarchical composite states inside the composite state with the deep history state. The other attributes are simply there for the sake of code re-use (i.e. code used for DC_Basic and DC_Composite to swap icons).
- DC_Orthogonal is an orthogonal block that allows for concurrently active states. It's really just a partitioning of a DC_Composite so doesn't need any attributes.
- DC_StickyNote is simply a device to annotate the model with even more information. It consists of a text field and has the visual appearance of a UML note. A visual arrow can be drawn from it to any other entity.

- DC_Port is used to represent a DCharts networking port. It has two boolean attributes, in_port and out_port. An in_port can receive events from the network, whereas the out_port sends them to the network. These events have the syntax "<port-name>.<event-string>".
- DC_Server is used to represent a DCharts server. It has the attributes id and name_string that are used to find the server. Usually you can just set both of these to the name of the DES file generated from a DCharts model. Refer to Thomas Feng's Master's thesis for details on these [Fen04].

The entities whose icons have a hexagonal shape at the top are generated as relationships/edges. They come in two types, which are set via edit dialogs. The first type is the invisible hierarchical relationship. The following entities are of this type: DC_ChartContains, DC_Contains, and DC_Orthogonality. AToM^3 was extended to internally keep track of such hierarchical relationships, so finding parents and children is easy. The second type of relationship is the visible arrows, which possess attributes just like the nodes did. The visual relationships and the meaning of their attributes are as follows:

- DC_Hyperedge, is really a simple directed transition between states. Only in the meta-model is it a hyper-edge. It consists of common statechart attributes such as a trigger, guard (condition), and action code. It also has DCharts specific attributes: priority, broadcast code and broadcast_to field. In an effort to save modeler time, the multiple_transitions attribute allows the user to make one DC_Hyperedge instance behave like any number of them. In other words, it replicates the trigger, guard, action code, broadcast, and broadcast_to fields as many times as the user needs. The configureIcon attribute has a visual representation as the text label associated with the transition. In the edit dialog, it also allows the user to choose exactly which fields should be part of the text label. For example the user can choose to display just the trigger, or any combination of all the fields previously described. Moreover, if multiple_transitions is used, each transition's information is concatenated in the label.
- DC_ServerPort represents a connection between a server and a port. It has the sole attribute connection, that is used to specify the port of the server to which the client is connected.

3.4 Formalism-specific UI modeling

Although the class diagram in figure 3.5 is sufficient to generate a working formalism, more modeling is needed to achieve our goals outlined at the start of section 3.3. This requires altering the buttons model of the generated formalism, observing events with the use of pre and post statecharts, acting on events with DCharts formalism and entity-specific statecharts, and handling layout for each hierarchical DCharts entity (again with a statechart).

In the following subsections, the labels on the states and transitions of the UI behaviour statecharts use a custom notation to make them more expressive. A star, x*, indicates that action code is present. A plus, x+, indicates that a different statechart handles the action. Parenthesis, <x>, indicate that the trigger event is generated by another statechart, such as the pre/post UI observers or another UI behaviour statechart. Regular brackets, (x), indicate the event was generated by the initialization routine for the entity when it is first instantiated. Square brackets[2] [x] indicate that the event was generated by the statechart itself, usually within the action code of a state.

[2]An established notation for statecharts defines square brackets to indication a condition, however this conflict cannot be resolved in time for the initial submission of this thesis.

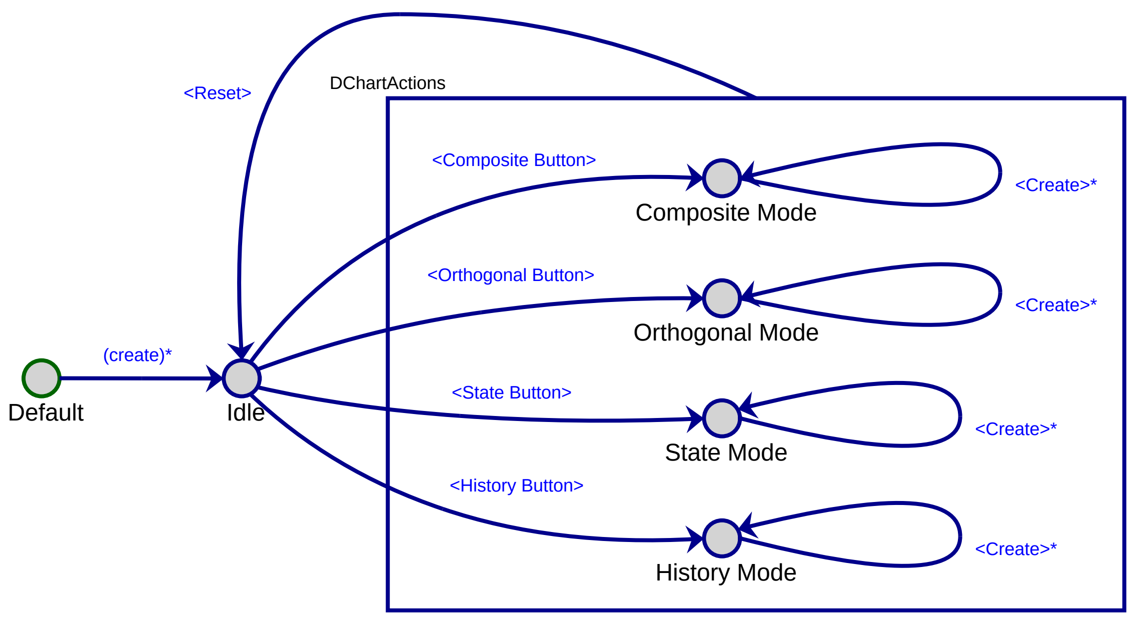


Figure 3.6: Button behaviour statechart

3.4.1 Buttons model

The buttons model is a trivial model, in the aptly named Buttons formalism. The buttons in this model correspond directly with the buttons that appear in the AToM^3 application's formalism toolbar. Buttons models are automatically generated from a meta-model, such as a class diagram, to create all the entities specified in the meta-model. For the new DCharts formalism, two routine additions to the automatically generated buttons model are required. These consist of adding two buttons that trigger statechart simulation and code generation.

A more exotic change to the buttons model is also needed. The DC_DChart entity creation button is modified to instantiate 5 different statecharts. A statechart for controlling buttons behaviour, a pre and a post statechart, a DC_DChart specific behaviour statechart, and finally a DC_DChart specific layout statechart. These will be discussed further in the following subsections. The number of different statecharts may seem excessive, but DC_DChart is not an ordinary entity. Its purpose is to provide a formalism-specific override to the generic UI behaviour.

The other entity creating buttons, such as for creating a DC_Basic state, are also modified. Instead of the buttons creating the entity in question on the canvas, they directly send an event to the DC_DChart's button statechart. Thus, the DC_DChart is made fully responsible for the creation of all other entities. Due to this approach, it is impossible to create a new entity outside of the visual container the DC_DChart forms.

3.4.2 Button Behaviour model

The button behaviour model is quite simple and is shown in figure 3.6. When the button to create entity X is pushed, the events "<Reset>" and "<X Button>" are sent to this statechart. If not already there, the statechart moves to an Idle state upon receipt of the first event. The second event then moves it to a state whereby entity X can get instantiated. It then waits for an event requesting the creation of that entity. The "<Create>" event is generated by the DC_DChart specific behaviour statechart when it intercepts and handles the "Model Action" event. See section 3.4.4 for more details.

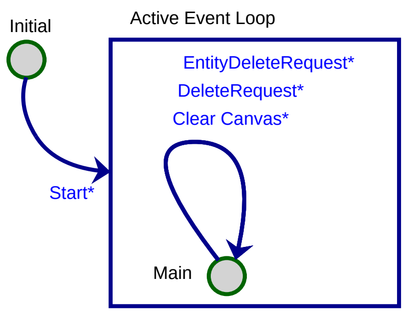


Figure 3.7: Pre UI observer statechart

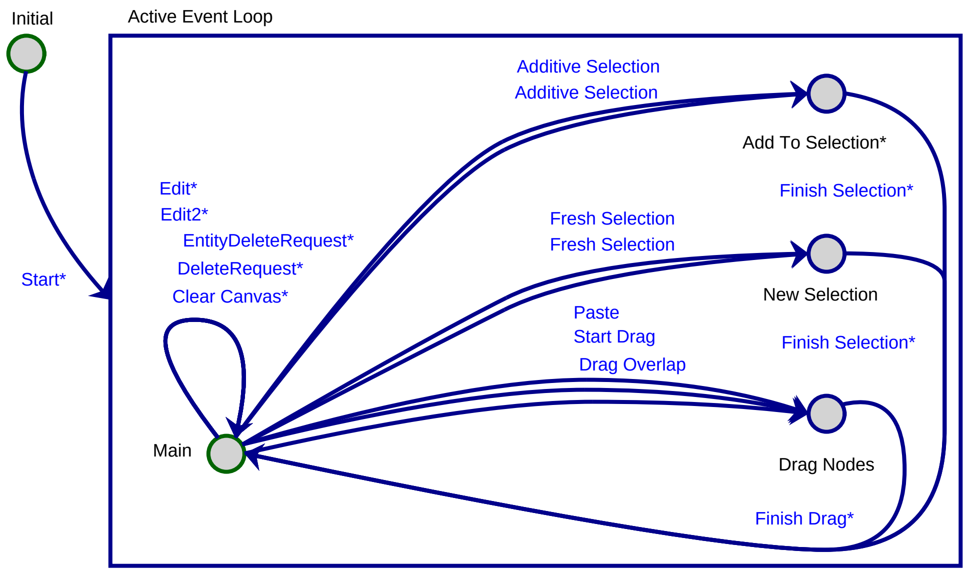


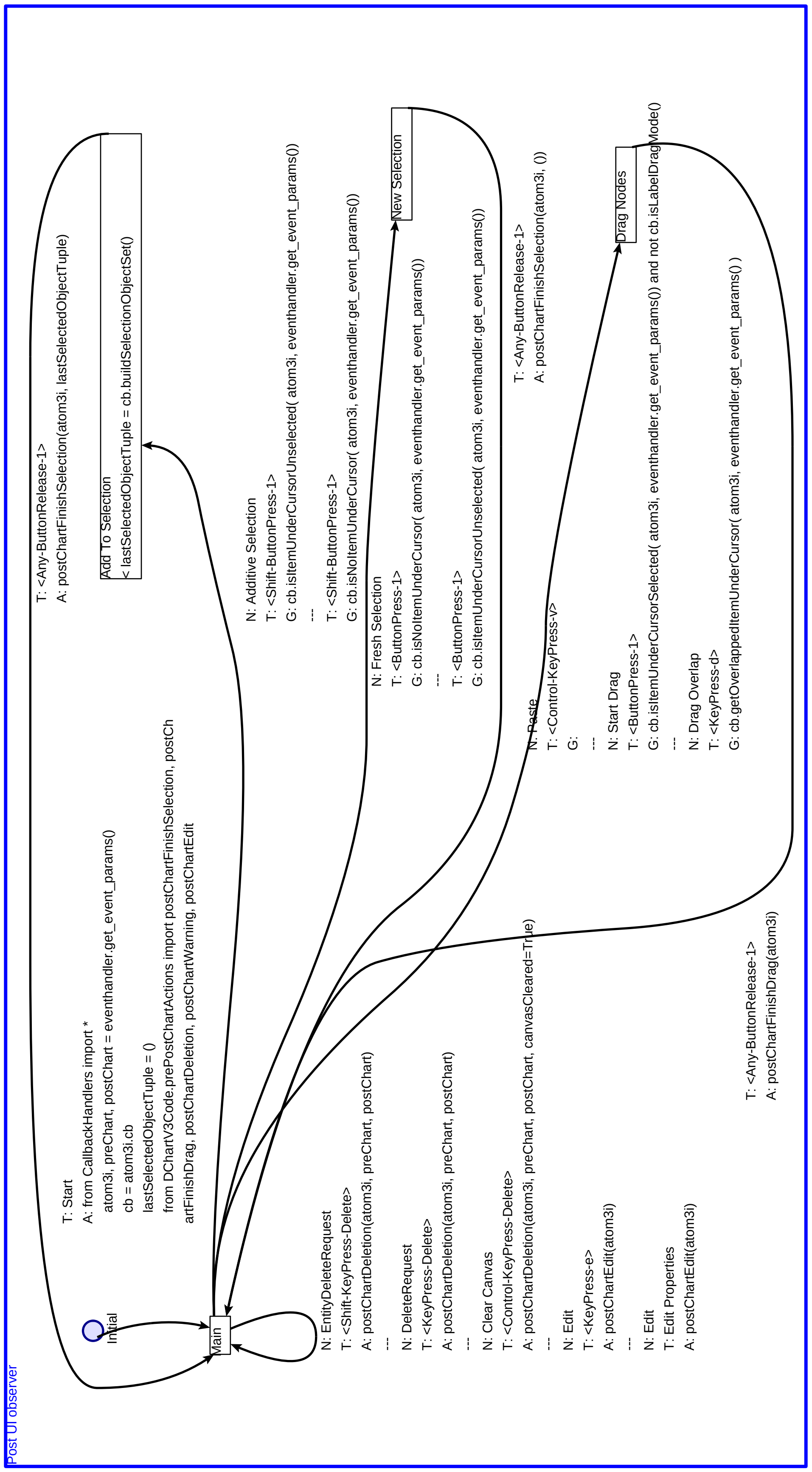
Figure 3.8: Post UI observer statechart

3.4.3 Pre/post observer statechart models

Recall that a pre UI statechart observes events before the generic UI behaviour statechart acts on them. The pre UI statechart is shown in figure 3.7. Likewise, the post UI statechart observes events just after the generic UI behaviour statechart acts on them and is shown in figures 3.8 and 3.9. For the new DCharts formalism, these observers prove useful mainly for the following four events: deletion, selection, the drop after dragging selected entities, and edit.

The deletion event is useful for two reasons. The first of these is rather trivial. It removes the pre/post statecharts from the main event loop if the DC_DChart instance has been deleted. The second is a layout consideration. If a state is deleted, then its parent, a composite state or the DC_DChart, may require less area. Thus it makes sense to send the behaviour statechart of the parent a layout request event. The parent's behaviour chart will in turn send a layout request event to the parent's layout statechart, where the layout will finally be handled. At the very least, this layout will result in the parent container shrinking itself to occupy less space. Ultimately, the parent may also completely redraw itself and its contents in a new configuration that takes advantage of the state's removal. Layout statecharts are discussed in greater detail in section 3.4.5.

The selection event makes it possible to detect what entities are selected. If an entity with hierar-


Post UI observer
Initial
Main
T: Start
A: from CallbackHandlers import *
atom3i, preChart, postChart = eventhandler.get_event_params()
cb = atom3i.cb
lastSelectedObjectTuple = ()
from DChartV3Code.prePostChartActions import postChartFinishSelection, postCh
artFinishDrag, postChartDeletion, postChartWarning, postChartEdit
N: EntityDeleteRequest
T: <Shift-KeyPress-Delete>
A: postChartDeletion(atom3i, preChart, postChart)

N: DeleteRequest
T: <KeyPress-Delete>
A: postChartDeletion(atom3i, preChart, postChart)

N: Clear Canvas
T: <Control-KeyPress-Delete>
A: postChartDeletion(atom3i, preChart, postChart, canvasCleared=True)

N: Edit
T: <KeyPress-e>
A: postChartEdit(atom3i)

N: Edit
T: Edit Properties
A: postChartEdit(atom3i)
T: <Any-ButtonRelease-1>
A: postChartFinishSelection(atom3i, lastSelectedObjectTuple)
Add To Selection
< lastSelectedObjectTuple = cb.buildSelectionObjectSet()
N: Additive Selection
T: <Shift-ButtonPress-1>
G: cb.isItemUnderCursorUnselected(atom3i, eventhandler.get_event_params())

T: <Shift-ButtonPress-1>
G: cb.isNoItemUnderCursor(atom3i, eventhandler.get_event_params())
New Selection
N: Fresh Selection
T: <ButtonPress-1>
G: cb.isNoItemUnderCursor(atom3i, eventhandler.get_event_params())

T: <ButtonPress-1>
G: cb.isItemUnderCursorUnselected(atom3i, eventhandler.get_event_params())
T: <Any-ButtonRelease-1>
A: postChartFinishSelection(atom3i, ())
N: Paste
T: <Control-KeyPress-v>
G:

N: Start Drag
T: <ButtonPress-1>
G: cb.isItemUnderCursorSelected(atom3i, eventhandler.get_event_params()) and not cb.isLabelDragMode()

N: Drag Overlap
T: <KeyPress-d>
G: cb.getOverlappedItemUnderCursor(atom3i, eventhandler.get_event_params())
Drag Nodes
T: <Any-ButtonRelease-1>
A: postChartFinishDrag(atom3i)

Figure 3.9: Post UI observer statechart bootstrapped in the new DCharts formalism

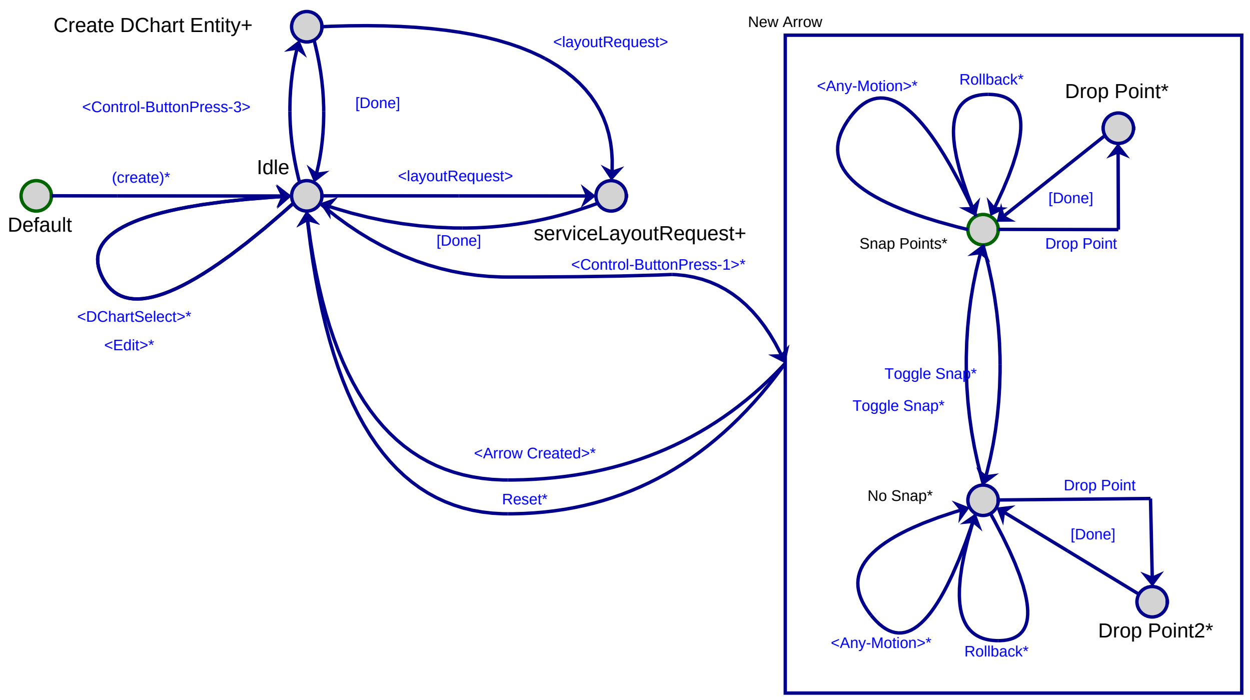


Figure 3.10: DC_DChart behaviour statechart

chical children, via containment relationships, is selected, then the children should also be selected. This makes it impossible for a user to delete or drag a container entity without doing the same to the contained children entities.

The drop event at the end of a drag and drop operation allows for dragging multiple entities outside the DC_DChart scoped UI environment. This is very useful for temporary editing by a modeler. For example, one could drag a composite state out of the DC_DChart, which triggers a containment disconnect (as the next section will show), and then drag the composite state back into another composite state inside the DC_DChart, triggering a new containment relation inside a composite state.

Finally, the edit event simply triggers an edit dialog. It could have been handled with entity-specific behaviour statecharts. However, if edit were only handled by the DC_DChart behaviour statechart, then an entity outside the containment of DC_DChart could not be edited.

3.4.4 Formalism entity-specific behaviour models

All visual entities of the DCharts formalism require their own behaviour models. The most important of these are those associated with the artificial entity that contains all others of the DCharts formalism and that of the composite state. Referring to the class diagram in figure 3.5, these correspond to DC_DChart and DC_Composite respectively. At the other extreme, the behaviour statechart for the transition edge, DC_Hyperedge, is trivial. All the remaining entities, excluding the non-visual containment relationships, use behaviour statecharts that are subsets of that of the composite state.

DC_DChart behaviour statechart

The behaviour of the DC_DChart entity begins with initialization when the entity is first created. This initialization includes a “(create)*” trigger that sets the active state to “Idle”. From then on, the following five events trigger interesting behaviour:

1. The “<DChartSelect>*” event is generated by the post UI observer statechart. The event indicates that DC_DChart has been selected by the user. It is then necessary to ensure that all the hierarchical children of DC_DChart are also selected so that delete and drag operations work as expected.
2. The “<Edit>*” event is also generated by the post UI observer statechart. It indicates that the user has opened an edit dialog on the DC_DChart attributes. The action code for the transition with this event trigger applies the changes made to the DC_DChart attributes, which range from the color of a default state to which layout statechart to use for a given class of entities.
3. The “<Control-Button-Press-3>” event is directly captured from the main event loop and explicitly handled, thus halting its propagation. This event indicates that a new DCharts formalism entity should be added to the canvas. Note that the same event is generated if one uses the AToM^3 menu system or a keyboard/mouse shortcut. The actual creation of an entity is of course handled by the button behaviour statechart previously seen in subsection 3.4.2.
4. The “<Control-Button-Press-1>*” event, is also directly captured from the main event loop. Moreover, this event triggers a lock, forcing all events in main event loop to only this statechart. The lock is only released when either an arrow is finally created or the process is aborted, via the “<Arrow Created>*” and “Reset*” events respectively. It is necessary to refine the behaviour found in the generic UI behaviour statechart for two reasons. The first is merely for the convenience of the user. Instead of allowing the user to draw arrows to indicate containment relationships, only transitions may be drawn. This saves time, and a perfectly good drag-and-drop method exists for creating and destroying containment relationships as shall be shown later in this subsection. The second reason is simply to know when transitions are actually created so that their UI behaviour statecharts may be initialized.
5. The “<layoutRequest>” event is generated exclusively by the UI behaviour statecharts of the children entities of DC_DChart. This event occurs when a new entity is created since the new entity will be contained by the DC_DChart and thus upsets the old layout. The event can also occur when DC_DChart is idle, such as when an entity is manually dragged by the user. The layout request is forwarded to the layout statechart of the DC_DChart, described in subsection 3.4.5.

DC_Composite behaviour statechart

The behaviour of the DC_Composite, the composite state, is the most complex of all. Fortunately, it is also re-usable by many other entities as shall be shown further on in this subsection. The initialization phase is rather involved, with two main possibilities. The first is that an interactive session with the user is in effect, in which case the “(create)” trigger signals the creation of a new DC_DChart. Immediately, the user is presented with a dialog asking them to which of the entities in the region of the newly created DC_Composite, they would like to contain the new composite state. If the composite state is successfully connected to either a DC_DChart or another DC_Composite, then the “[didConnect]” trigger is generated, followed by a “<layoutRequest>” event to the container, and finally a “[Done]” event to set the state to “HasParent”. If the composite state is not successfully connected, then a “[didNotConnect]” event is generated and the active state is set to “NoParent”.

Finally, the second of the two possibilities is that the model was being loaded rather than interactively edited. In this case, a “(loadModelCreate)” event is first sent when the DC_Composite is first instantiated, setting the active state to “NoParent”. Then a second “(loadModelCreate)”

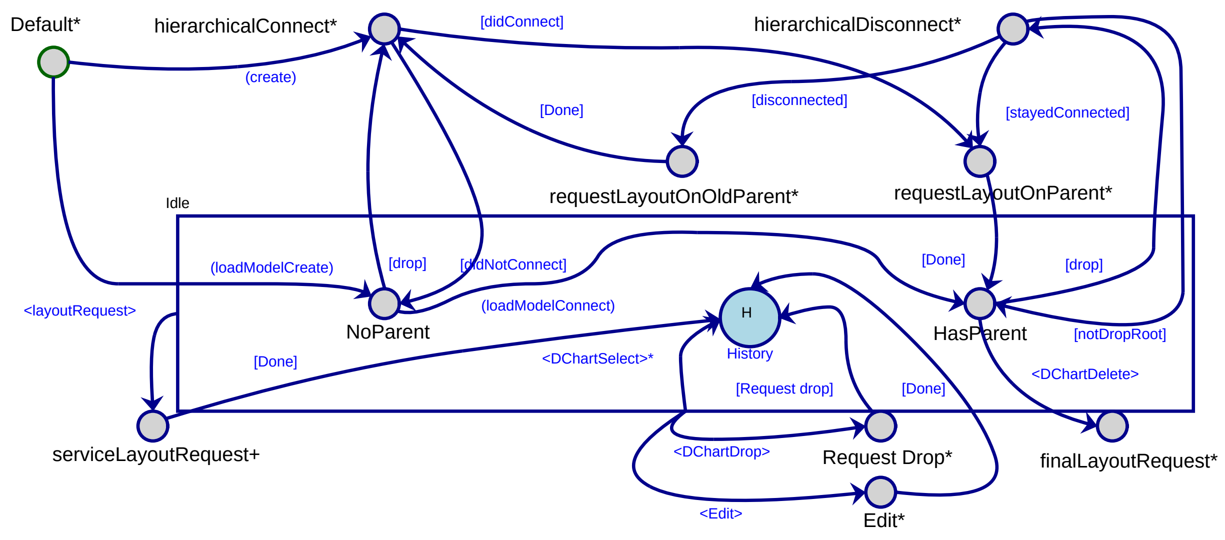


Figure 3.11: DC_Composite behaviour statechart

event is sent if a containing relationship is instantiated with this DC_Composite as its parent, thus setting the active state to “HasParent”. The following is a list of all the events that occur after the initialization phase. Unless stated otherwise, the events are generated by the post UI observer statechart.

1. The "<DChartSelect>*" event is dealt with in the same manner as the DC_DChart UI behaviour statechart. All hierarchical children are selected.
2. The “<Edit>” event indicates that the user has opened an edit dialog on the DC_Composite attributes. In particular, the user may have changed the amount of information visually displayed by the DC_Composite. At the furthest extreme, the user may have requested that all children entities contained by the composite state be rendered invisible, thus reducing the size of the composite state drastically. Thus the transition with this trigger event will execute action code to apply the changes and often generate a “<layoutRequest>” event as well.
3. The "<DChart Drop>" event indicates that this composite state, among potentially many other entities, has just been dragged and then dropped. The transition with this trigger promptly generates two events: “[Done]”, which restores the active state to either “NoParent” or “HasParent”, followed by “[drop]”, which causes hierarchical connection or hierarchical disconnection, respectively, to be attempted. A disconnection occurs only if the entity has been dropped outside of its parent container and the user has explicitly agreed to disconnect it. This triggers a “<layoutRequest>” followed by an attempt to hierarchically connect the disconnected composite state in its new location.
4. The “<DChartDelete>” event indicates that this composite state is to be deleted. Before being erased, it warns its hierarchical container parent with a “<layoutRequest>”. In this fashion the parent can find a new layout that takes advantage of the extra space afforded by the deleted entity.
5. The "<layoutRequest>" event is generated exclusively by the UI behaviour statecharts of the children entities of DC_Composite, just as it was in DC_DChart. This even occurs whenever the children of this entity are modified by the user, such as by add/removing them from the DC_Composite or by simply moving them. The layout request is forwarded to the layout

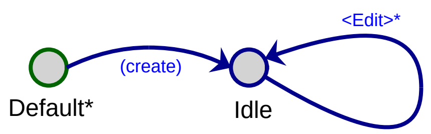


Figure 3.12: DC_Hyperedge behaviour statechart

statechart of the DC_Composite, described in subsection 3.4.5.

DC_Hyperedge behaviour statechart

The behaviour of the DC_Hyperedge or transition, is trivially simple, as figure 3.12 shows. As noted earlier, the transition is a hyper-edge only in the meta-model, in the generated DCharts formalism itself it is a simple directed edge with one source and one target. The transition is first initialized with a "(create)" event. Afterwards, it simply awaits "<Edit>*" events from the post UI behaviour chart in order to apply changes made in its edit dialog. These changes affect the information content of the label associated with the transition.

Other behaviour statecharts

The UI behaviour statecharts of the remaining DCharts formalism entities are subsets of the one previously shown for DC_Composite. For the DC_Orthogonal entity which denotes an orthogonal partition of a composite state, and is also a hierarchical container entity, the only structural difference is the removal of the "<Edit>" event. Orthogonal partitions have no attributes that require application after being edited by the user in an edit dialog. The only other difference is that DC_Orthogonal forwards layout requests to its own layout statechart as will be shown in the next section.

The remaining entities are not hierarchical container entities, but rather primitive children. They include DC_Basic, DC_History, DC_Port, DC_Server, and DC_Stickynote. The main structural difference between their UI behaviour statecharts and that of the one for DC_Composite is that they do not accept the "<layoutRequest>" event. Naturally, no entity exists that would generate and send it to them. Also, some of them do not have attributes whose modification by the user must be applied, so they do not accept the "<Edit>" event either. In those that do accept the "<Edit>" event, they apply different attributes than would the DC_Composite UI behaviour statechart. In all other aspects, the UI behaviour statecharts are identical to that of the DC_Composite.

3.4.5 Layout Behaviour

The final link to the built-in automatic layout methods described in chapter 2 are the layout behaviour statecharts. Each hierarchical container type can potentially have a different layout behaviour statechart. Recall that the hierarchical containers in DCharts are: DC_DChart, DC_Composite, and DC_Orthogonal. Moreover, the layout statechart models associated with each of these can be modified at run-time by the user, compiled, and substituted for all the existing layout statecharts without restarting the AToM^3 tool. This enables rapid prototyping of new layout behaviours by both the formalism developer and the final user.

Figure 3.13 shows a typical layout statechart. In this case the built-in layout algorithm it draws upon is Force-transfer, described in detail in section 2.6. Structurally, it is very simple. It is initialized with a "(create)*" event and is thereafter ready to do layout. The "<applyLayout>" event, which triggers the layout sequence, is generated by the behaviour statechart associated with the entity requiring layout. This occurs when the entity's behaviour statechart enters the state

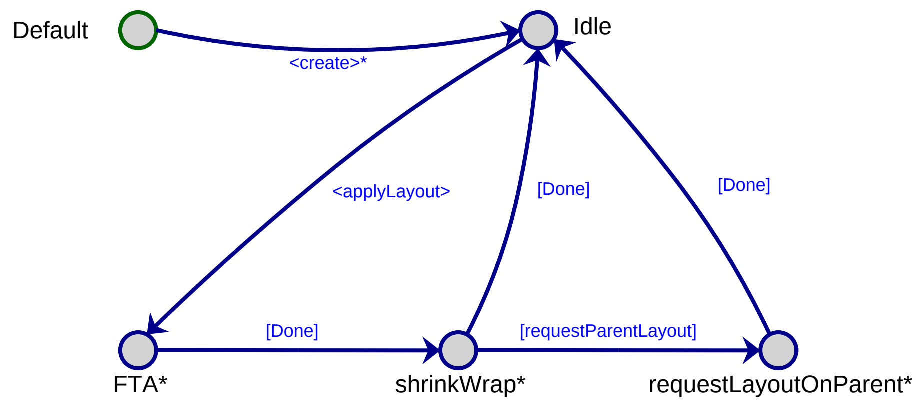


Figure 3.13: Force-Transfer Layout Statechart

"serviceLayoutRequest+". Thereafter, the following sequence of actions occur:

1. Apply the built-in general purpose layout algorithm. Inside the action code, a choice is made of which types of entities and links that should be sent to the layout algorithm. This choice is generally limited to only the direct children entities of the hierarchical container parent and the visual arrows between them. Moreover all the parameters passed to the layout algorithm are chosen. After the layout is applied, a "[Done]" event is generated.
2. Apply a trivial shrink-wrapping algorithm. This simply fits the hierarchical parent to be visually just large enough to contain all its children entities. As is explained in the third action, either a "[Done]" or a "[requestParentLayout]" event are generated when done.
3. Send a layout request to the behaviour statechart of the parent of this hierarchical container. This action is only taken conditionally, which is depicted in the layout statechart using two alternate transitions. Clearly one condition is that the hierarchical container possesses a hierarchical parent itself. The other condition, is that the hierarchical container has either moved or changed size. Obviously, if neither position nor size have changed, the layout of the higher-levels of the hierarchy are completely unaffected. Finally, a "[Done]" event is generated and the layout statechart returns to the "Idle", ready state.

The propagation of layout requests described in action 3, flow upwards only, in the current implementation. In other words, they propagate from the lowest to the highest level of the hierarchy. This is because the lowest level of the hierarchy will determine a layout that uses a certain area. Requesting this lower level hierarchical container to use more space is of no value, since area is at a premium. Requesting it to use less space is equally unfeasible, since it will simply result in overlap. By obscuring information, overlap defeats the purpose of automatic layout.

However, suppose instead that a layout statechart provided support for multiple layout algorithms with different amounts of area efficiency. In this situation, it would indeed make sense for the higher level hierarchical containers to require the lower level hierarchical containers to use more or less area by choosing different layout algorithms. The difficulty, of course, would be that the lower level hierarchical container is not guaranteed to be able to meet certain area requirements, barring overlap, no matter what layout it chooses. Therefore, some compromise would be the best this system could achieve.

A final consideration is that the modeled behaviour of the DCharts formalism is designed for interactive sessions with the user. An example of such a session is shown in figure 3.14. In the figure, the entire reactive behaviour of a standard four button wristwatch is modeled. However, a statechart model might be generated automatically using graph transformations. This can be dealt with by simply adding a button to the formalism that does a reversed, breadth first search of the model, and directly sends layout request events to each hierarchical container.

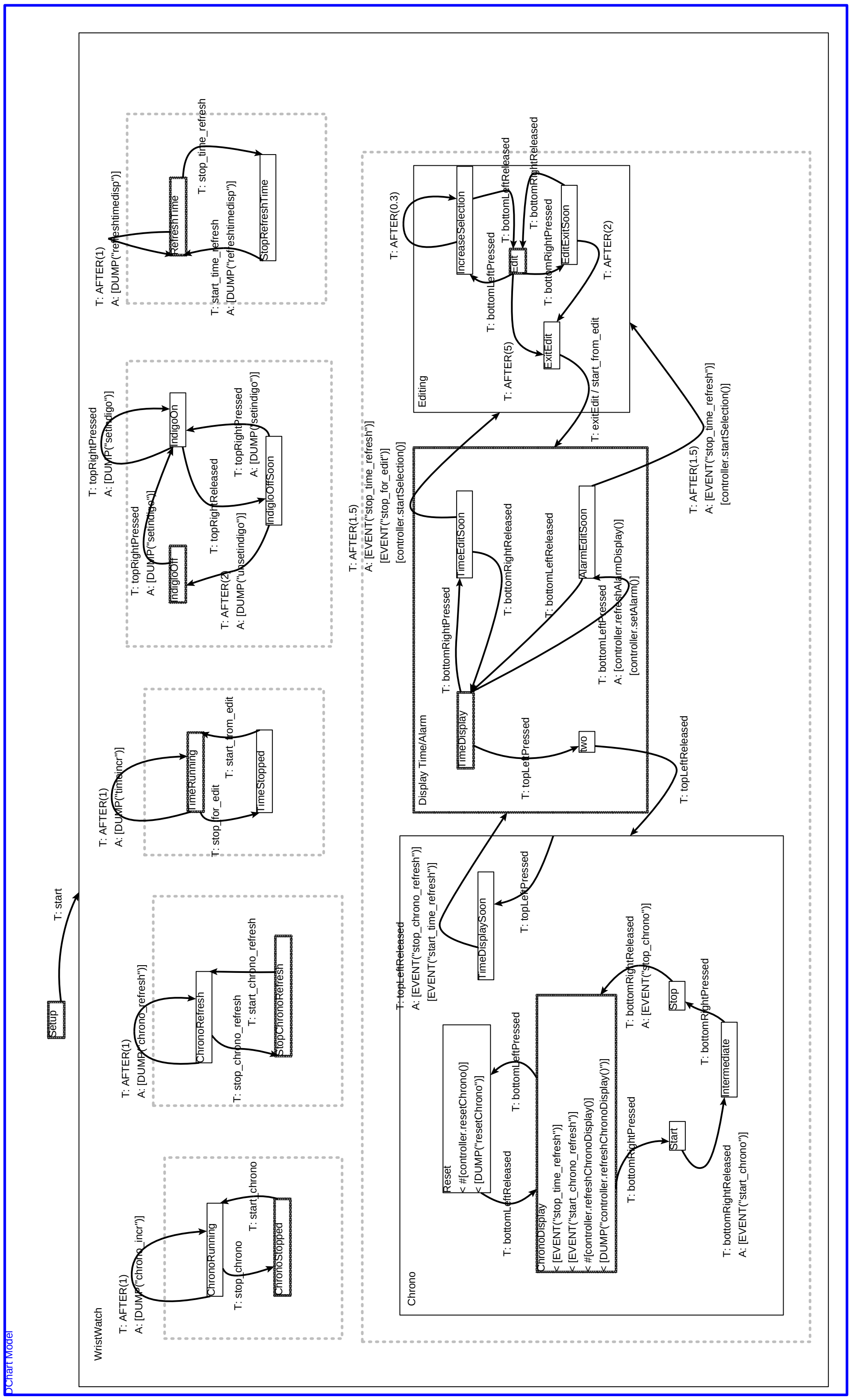


Figure 3.14: Wristwatch behaviour model in the new DCharts formalism

Conclusion

The goal of this thesis was to find a better solution for the specification of the reactive behaviour of multi-formalism visual modeling environments. Such a solution was needed due to the growing complexity of visual models and the need to decompose complex systems into models of multiple formalisms. Moreover, it is necessary that new formalisms be easy to create and integrate into the visual modeling environment, since complex systems are best modeled in formalisms that are as close as possible to the problem domain. Thus the modeled framework presented in chapter 3 for dealing with reactive behaviour, including layout behaviour, of visual modeling environments in a fashion that naturally supports multiple formalisms simultaneously is a significant improvement over the hard-coded approaches found in known existing visual modeling tools.

In visual modeling, the arrangement of the vertices and edges of a model are highly significant. Since manual layout is so time consuming, automatic layout is an important component of the visual modeling environment, the behaviour of which is included in the modeled framework presented. To realize the automatic layout, a number of graph drawing techniques were implemented and were described in detail in chapter 2. These techniques included a fairly sophisticated layered layout engine as well as a multiple physical simulation based spring-embedder layout engine. Moreover, efforts were made to tap into existing graph drawing tools (such as yED) via export/import mechanisms, though this proved generally unsatisfactory. Finally, a linear constraint based layout was implemented, making use of the QOCA linear constraint solving toolkit. All the different graph drawing techniques realized prove necessary to handle the specific needs of the many different formalisms encountered in a multi-formalism visual modeling tool such as AToM^3.

Many graph drawing techniques exist in the literature. Prior to the implementations described in chapter 2, a thorough review of the existing techniques was compiled. This review, described in the first chapter, included the visual aesthetics optimized by each technique as well as the computational complexity of the algorithms whenever possible.

Future work

In many ways, this thesis is just a starting point. Although the framework for the reactive behaviour of the visual modeling environment is robust, the code synthesized from the behaviour models is not nearly as efficient as possible (*i.e.*, some commercial tools can synthesize highly optimized code). Moreover, further work is required to fully explore the layout behaviour of the visual models and solve some important layout algorithm challenges. Finally, to make the worth of this approach even clearer, an existing (commercial) visual modeling environment should be recreated and improved upon.

Layout behaviour

The layout behaviour component of the framework for reactive behaviour of the visual modeling environment requires further work. In particular, in the current implementation, layout requests propagate strictly upward from lower hierarchical levels to higher hierarchical levels. There are a number of situations in which this is insufficient. Consider a situation whereby a higher hierarchical level must be constrained to use a certain amount of area. It is thus necessary that this higher level be able to negotiate with lower levels to use less area. This can be accomplished in several ways. The lower level hierarchical components may employ different layout algorithms, different layout algorithm parameters, or use zooming to scale down the amount of information shown.

Layout challenges

There remain some important challenges in developing layout algorithms for visual modeling. At a high level, there remains the question of whether the layout should be fully automatic (*i.e.*, the user creates/modifies a model but does not participate in the visual arrangement of the components of the model at all) or manual with automatic support. The former approach promises ultimate efficiency for the modeler. The caveat is that the graph drawing technique used must work at interactive speeds and provide sufficient quality, as measured by a broad range of visual aesthetics among which mental map is particularly important in this context. In the current implementation, models are drawn manually (with the exception of automatic vertex overlap removal via the force transfer algorithm) and additional layout is provided only at the modeler's explicit request.

The decomposition of a model into hierarchical components, despite its great usefulness, presents additional difficulties for graph drawing algorithms. Recall that most graph drawing algorithms are non-polynomial in complexity, thus a hierarchical decomposition provides huge speed increases, because many small problems are much easier to solve than a single large problem in this context. The difficulties occur when elements of a hierarchical component have edges with an end that lies outside this component (an external edge). This means that the layout of the entire graph is not optimized by the localized layout of the hierarchical components since, in the current implementation, such external edges are ignored. Thus it is necessary to extend layout algorithms to handle these external edges. For the layered drawing technique, this may be done by either implementing a hierarchical aware variant or by simply adding row and layer constraints to the vertices (*e.g.*, a vertex with an external edge to a vertex above the hierarchical component should be constrained to the top layer). For a spring-embedder drawing technique, one could imagine treating the external vertices as anchor points and proceeding as usual, although it may prove necessary to project the external vertices to the borders of the hierarchical component so as not to artificially inflate the size of this component. Finally, it was previously assumed that layout at high hierarchical levels did not affect lower levels. This is clearly no longer the case with external edges, thus layout behaviour becomes increasingly complex, and requires negotiation between levels as previously discussed.

At a low level, the efficiency of all the layout algorithms could be considerably improved. In particular, the layered, spring-embedder, and force-transfer drawing techniques would benefit most. The simplest improvement is just to re-implement them in a non-interpreted language. The spring-embedder and force-transfer can also be improved using partitioning techniques to eliminate the need for pair-wise calculations between every vertex.

Recreate an existing tool

Rebuilding an existing modeling environment such as Simulink (`www.mathworks.com/products/simulink/`) would aid in proving the worth of the framework described in this thesis. In particular, it would show that a completely modeled approach is possible, efficient, and easily maintained. The easy maintenance stems from two aspects of modeling reactive behaviour with statecharts. The first is that small changes to the behaviour can be realized with similarly small changes to the statechart model, irrespective of any large changes that may occur in the generated code. The second is that unlike code, statecharts are easily understood, thus serving as documentation in and of themselves. Once an existing modeling environment has been rebuilt, then completely new formalism-specific environments should be modeled and synthesized (and formalisms allowed to co-exist).

As a final note, the figures used in this thesis are the result of manual layout unless otherwise stated. Manual layout was generally used whenever automatic layout was not available at the time of model creation and when maximum compactness of the model was desired. All implementations used in this thesis and additional pictures are available at `msdl.cs.mcgill.ca/people/denis/`.

4
Glossary

Vertex An element of the set V, the set of vertices of which a graph G is composed. Also called node or point.

Edge An element of the set E, the set of edges connecting the vertices in a graph G. Edges can be drawn as a single line or as a poly-line sequence. If a line is curved then it is drawn as a spline. Orthogonal edges are a sequence of connected horizontal and vertical line segments.

Graph A pair G = (V, E) of disjoint sets. The set E is an mapping of edges to a pair of vertices. The vertex pairs of an edge are unordered. If an edge maps both ends to the same vertex, it is a loop. Multiple edges can possess the same endpoint vertex or vertices. Also called a multi-graph.

Digraph Digraphs are the directed version of graphs. In digraphs, the order of the endpoint vertices in the description of an edge is important. Thus the edges e_1=(v_1, v_2) and e_2=(v_2, v_1) are different for digraphs but identical for graphs.

Subgraph A graph G' is a subgraph of G if V' $\subseteq$V and E' $\subseteq$E. In other words, the graph G contains everything in G'.

Bipartite-graph A graph such that the vertices can be divided into exactly two groups and that no edge has both endpoint vertices in the same group.

Forest A graph containing no cycles.

Tree A graph containing no cycles and such that all components are connected.

Incident A vertex is incident to an edge if it is an endpoint of the edge.

Adjacent A vertex is adjacent (neighbor) of another vertex if an edge connects them.

Degree The number of edges connected to a vertex.

Indegree The number of incoming edges to a vertex.

Outdegree The number of outgoing edges from a vertex.

Source A vertex of a digraph with no incoming edges.

Sink A vertex of a digraph with no outgoing edges.

Acyclic-graph A graph with no cycles.

st-digraph An acyclic digraph with exactly one source and one sink (also called bipolar digraph).

Path A sequence of vertices such that from each of its vertices there is an edge to the successor vertex.

Connected-graph A graph G = (V, E) where for all vertices $v_1$and v_2, there exists a path from $v_1$to v_2(also called 1-connected).

Biconnected-graph A graph where any two vertices are joined by two vertex-disjoint paths (also called 2-connected). Equivalently, a graph that has no cut-vertex. The removal of a cut-vertex divides a connected subgraph into two or more smaller connected subgraphs.

Sparse-graph A graph G = (V, E) with |E| = O(|V|).

Small-graph A natural definition for this is a sparse graph G = (V, E), where |V| is small enough that NP-hard layout problems can be solved exactly. This is typically true for graphs with less than 100 vertices.

Medium-graph Similarly to a small-graph, a medium-graph has |V| small enough that layout problems with quadratic time complexity can be solved in reasonable time. There is no agreed upon range for a medium-graph in the literature, but as a general guideline, a graph with less than 1000 vertices can be considered of medium size.

Large-graph Any graph larger than a medium-graph. Layout problems on such graphs must have linear or nearly linear time complexity to terminate within reasonable time. At present, the largest graphs that can be drawn in reasonable time have at most 10^7vertices [YKC02].

Compound-digraph A directed graph G = (V, E_A, E_I) . The edges E_A are the usual directed adjacency edges typically drawn as arrows. The edges E_I are directed inclusion edges and are typically drawn as a geometric inclusion. The parent vertex is usually drawn as a rectangle with the children vertices drawn inside this boundary. The inclusion edges are required to form a tree or hierarchical structure.

Hypergraph A graph G = (V, E) where V and E are disjoint sets and the elements of E are non-empty subsets of V (of any cardinality). In other words, hypergraphs have hyperedges that can connect from one to many vertices.

Mixed-graph A graph containing both directed edges and undirected edges.

Graph-theoretical-distance The graph theoretical distance of two vertices is a measure of the shortest path between them.

Bibliography

[ALPW04] Siew Cheong Au, Christopher Leckie, Ajeet Parhar, and Gerard Wong. Efficient visualization of large routing topologies. *Int. J. Netw. Manag.*, 14(2):105–118, 2004.

[And98] Elias Andersson. Automatic layout of diagrams in rational rose. Master's thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, March 1998.

[AS01] Maneesh Agrawala and Chris Stolte. Rendering effective route maps: improving usability through generalization. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 241–249, New York, NY, USA, 2001. ACM Press.

[Bar98] Roswitha Bardohl. Genged - a generic graphical editor for visual languages based on algebraic graph grammars. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, pages 48–55, Washington, DC, USA, 1998. IEEE Computer Society.

[BBD00] Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Trans. Comput.*, 49(8):826–840, 2000.

[BBS01] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[BCPDB04] John M. Boyer, Pier Francesco Cortese, Maurizio Patrignani, and Giuseppe Di Battista. Stop minding your p's and q's: Implementing a fast and simple dfs-based planarity testing and embedding algorithm. In Giuseppe Liotta, editor, *Graph Drawing, Perugia, Italy, September 21-24, 2003*, pages 25–36. Springer, 2004.

[BDPP99] Giuseppe Di Battista, Walter Didimo, Maurizio Patrignani, and Maurizio Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, pages 297–310, London, UK, 1999. Springer-Verlag.

[BGL95] Giuseppe Di Battista, Ashim Garg, and Giuseppe Liotta. An experimental comparison of three graph drawing algorithms (extended abstract). In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 306–315, New York, NY, USA, 1995. ACM Press.

[BGL$^+$97] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, Emanuele Tassinari, and Francesco Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7(5-6):303–325, 1997.

[BHR96] Franz-Josef Brandenburg, Michael Himsholt, and Christoph Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 76–87, London, UK, 1996. Springer-Verlag.

[BJL01] Christoph Buchheim, Michael Junger, and Sebastian Leipert. A fast layout algorithm for k-level graphs. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, volume 1984, pages 229–240, London, UK, 2001. Springer-Verlag.

[BJM02] Wilhelm Barth, Michael Junger, and Petra Mutzel. Simple and efficient bilayer cross counting. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 130–141, London, UK, 2002. Springer-Verlag.

[BK98] Therese Biedl and Goos Kant. A better heuristic for orthogonal graph drawings. *Comput. Geom. Theory Appl.*, 9(3):159–180, 1998.

[BK02] Ulrik Brandes and Boris Kopf. Fast and simple horizontal coordinate assignment. In Petra Mutzel, Michael Junger, and Sebastian Leipert, editors, *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44, London, UK, 2002. Springer-Verlag.

[BM01] Oliver Bastert and Christian Matuszewski. *Drawing graphs: methods and models*, volume 2025. Springer-Verlag, London, UK, 2001.

[BMSX97] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 87–96, New York, NY, USA, 1997. ACM Press.

[BNT86] C Batini, E Nardelli, and R Tamassia. A layout algorithm for data flow diagrams. *IEEE Trans. Softw. Eng.*, 12(4):538–546, 1986.

[Boy05] John Boyer. A new method for efficiently generating planar graph visibility representations. In P. Eades and P. Healy, editors, *Proceedings of the 13th International Conference on Graph Drawing 2005.* Lecture Notes Comput. Sci., Springer-Verlag, Proceedings of the 13th International Conference on Graph Drawing 2005 2005. To appear.

[BP90] Karl-Friedrich Bohringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 43–51, New York, NY, USA, 1990. ACM Press.

[BV03] Spencer Borland and Hans L. Vangheluwe. Transforming Statecharts to DEVS. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S154 – S159. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[Cim95] Robert Cimikowski. An analysis of some heuristics for the maximum planar subgraph problem. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 322–331, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[CMP99] Sitt Sen Chok, Kim Marriott, and Tom Paton. Constraint-based diagram beautification. In *VL '99: Proceedings of the IEEE Symposium on Visual Languages*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[DH96] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[Die05] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, New York, 3rd edition, 2005.

[Dji95] Hristo Djidjev. A linear algorithm for the maximal planar subgraph problem. In *WADS '95: Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 369–380, London, UK, 1995. Springer-Verlag.

[DL98] Walter Didimo and Giuseppe Liotta. Computing orthogonal drawings in a variable embedding setting. In *ISAAC '98: Proceedings of the 9th International Symposium on Algorithms and Computation*, pages 79–88, London, UK, 1998. Springer-Verlag.

[dLGV04] Juan de Lara, Esther Guerra, and Hans L. Vangheluwe. Meta-Modelling, Graph Transformation and Model Checking for the Analysis of Hybrid Systems. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, Lecture Notes in Computer Science 3062, pages 292 – 298. Springer-Verlag, 2004. Charlottesville, Virginia, USA.

[dLV02a] Juan de Lara and Hans L. Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science 2306, pages 174 – 188. Springer-Verlag, April 2002. Grenoble, France.

[dLV02b] Juan de Lara and Hans L. Vangheluwe. Computer aided multi-paradigm modelling to process petri-nets and statecharts. In *International Conference on Graph Transformations (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, October 2002. Barcelona, Spain.

[dLV02c] Juan de Lara and Hans L. Vangheluwe. Using AToM^3 as a Meta-CASE tool. In *4^{th} International Conference on Enterprise Information Systems (ICEIS)*, pages 642 – 649, April 2002. Ciudad Real, Spain.

[dLV02d] Juan de Lara and Hans L. Vangheluwe. Using meta-modelling and graph grammars to process GPSS models. In Hermann Meuth, editor, 16^{th} *European Simulation Multiconference (ESM)*, pages 100–107. Springer-Verlag, June 2002. Darmstadt, Germany.

[dLV04] Juan de Lara and Hans L. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3 - 4):309–330, June - August 2004. Special Issue on Domain-Specific Modeling with Visual Languages.

[dLV05] Juan de Lara and Hans L. Vangheluwe. *Management of Object-Oriented Development Process*, chapter Model-Based Development: Meta-Modelling, Transformation and Verification, pages 289–312. The Idea Group Inc., October 2005.

[dLVA04] Juan de Lara, Hans L. Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSyM)*, 3(3):194–209, August 2004.

[dLVAM03] Juan de Lara, Hans L. Vangheluwe, and Manuel Alfonseca Moreno. Computer Aided Multi-Paradigm Modelling of Hybrid Systems with AToM^3. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference*, pages 83 – 88. Springer-Verlag, July 2003. Montréal, Canada.

[Dwy01] Tim Dwyer. Three dimensional UML using force directed layout. In *CRPITS '01: Australian symposium on Information visualisation*, pages 77–85, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[Eig03] Markus Eiglsperger. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach.* Ph.D. thesis, Fakultät für Informations- und Kognitionswissenschaften (Wilhelm-Schickard Institut für Informatik), Tübingen University, Sep 2003.

[EKS03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff, New York, NY, USA, 2003. ACM Press.

[EL89] P. Eades and X. Lin. How to draw a directed graph. *In Proc. IEEE Workshop on Visual Languages*, pages 13–17, 1989.

[ESK04] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In János Pach, editor, *Graph Drawing, New York, 2004*, pages 155–166. Springer, 2004.

[EW94a] Peter Eades and Sue Whitesides. Drawing graphs in two layers. *Theor. Comput. Sci.*, 131(2):361–374, 1994.

[EW94b] Peter Eades and Nicholas C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[Fen03] Thomas Huining Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S147 – S166. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[Fen04] Thomas Huining Feng. Dcharts, a formalism for modeling and simulation based design of reactive software systems. Master's thesis, School of Computer Science, McGill University, Montréal, Canada, February 2004.

[FK96] Ulrich Fobmeier and Michael Kaufmann. Drawing high degree graphs with low bend numbers. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 254–266, London, UK, 1996. Springer-Verlag.

[FK97] Ulrich Fobmeier and Michael Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 134–145, London, UK, 1997. Springer-Verlag.

[FLM94] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 388–403, Berlin, Germany, 10–12 1994. Springer-Verlag.

[For02] Michael Forster. Applying crossing reduction strategies to layered compound graphs. In Michael T. Kobourov, Stephen G.; Goodrich, editor, *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 276–284, London, UK, 2002. Springer-Verlag.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[GGK00] P. Gajer, M. Goodrich, and S. Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In Joe Marks, editor, *Graph Drawing*, volume 1984 / 2001, pages 211–222. Springer, 2000.

[GJ83] M. R. Gary and D. S. Johnson. Crossing number is np-complete. *SIAM Journal Algeraic and Discrete Methods*, 4:312–316, 1983.

[GKN04] Emden Gansner, Yehuda Koren, and Stephen North. Topological fisheye views for visualizing large graphs. In *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'04)*, pages 175–182, Washington, DC, USA, 2004. IEEE Computer Society.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[GM04] Carsten Gutwenger and Petra Mutzel. Graph embedding with minimum depth and maximum external face (extended abstract). In Giuseppe Liotta, editor, *Graph Drawing, Perugia, 2003*, pages pp. 259–272. Springer, 2004.

[GMW01] Carsten Gutwenger, Petra Mutzel, and Rene Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 246–255, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[GN98] Emden R. Gansner and Stephen C. North. Improved force-directed layouts. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, pages 364–373, London, UK, 1998. Springer-Verlag.

[GT97] Ashim Garg and Roberto Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 201–216, London, UK, 1997. Springer-Verlag.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[HL03] Xiaodi Huang and Wei Lai. Force-transfer: a new approach to removing overlapping nodes in graph layout. In *CRIPTS '03: Proceedings of the twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 349–358, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[HMM02] Nathan Hurst, Kim Marriott, and Peter Moulder. Dynamic approximation of complex graphical constraints by linear constraints. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 191–200, New York, NY, USA, 2002. ACM Press.

[HN02] Patrick Healy and Nikola S. Nikolov. How to layer a directed acyclic graph. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, pages 16–30, London, UK, 2002. Springer-Verlag.

[Hos01] Hiroshi Hosobe. A modular geometric constraint solver for user interface applications. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 91–100, New York, NY, USA, 2001. ACM Press.

[HS94] David Harel and Meir Sardas. Randomized graph drawing with heavy-duty preprocessing. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 19–33, New York, NY, USA, 1994. ACM Press.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[JdLM04] Hans Vangheluwe Juan de Lara and Pieter J. Mosterman. Modelling and analysis of traffic networks based on graph transformation. *Formal Methods for Automation and Safety in Railway and Automotive Systems*, page 11, December 2004. Braunschweig, Germany.

[JLMO97] Michael Junger, Eva K. Lee, Petra Mutzel, and Thomas Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 13–24, London, UK, 1997. Springer-Verlag.

[JM96] Michael Junger and Petra Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

[JM97] Michael Junger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms Appl.*, 1(1):1–25, 1997.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KKM01] G. W. Klau, K. Klein, and P. Mutzel. An experimental comparison of orthogonal compaction algorithms. In J. Marks, editor, *Proc. 8th Internat. Symp. on Graph Drawing (GD 2000)*, volume 1984, pages 37–51, Colonial Williamsburg, VA, USA, September 2001. Springer–Verlag.

[KMS94] Corey Kosak, Joe Marks, and Stuart Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Trans. Systems, Man and Cybernetics*, 24(3):440–454, 1994.

[LART86] Eli Messinger Carl Meyer Charles Spirakis Lawrence A. Rowe, Michael Davis and Allen Tuan. A browser for directed graphs. Technical Report UCB/CSD-86-292, EECS Department, University of California, Berkeley, 1986.

[LE98] W. Lai and P. Eades. Routing drawings in diagram displays. In *APCHI '98: Proceedings of the Third Asian Pacific Computer and Human Interaction*, pages 291 – 296, Washington, DC, USA, 1998. IEEE Computer Society.

[LJVdLM04] Simon Lacoste-Julien, Hans L. Vangheluwe, Juan de Lara, and Pieter J. Mosterman. Meta-modelling hybrid formalisms. In Pieter J. Mosterman and Jin-Shyan Lee, editors, *IEEE International Symposium on Computer-Aided Control System Design*, pages 65 – 70. IEEE Computer Society Press, September 2004. Taipei, Taiwan.

[Mas92] Toshiyuki Masui. Graphic object layout with interactive genetic algorithms. In *Proc. of the 1992 IEEE Workshop on Visual Languages*, pages 74–80, Seattle, WA, 1992.

[Mas94] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 103–108, New York, NY, USA, 1994. ACM Press.

[Mey98] Bernd Meyer. Competitive learning of network diagram layout. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, pages 56–63, Washington, DC, USA, 1998. IEEE Computer Society.

[MH98] Guy Melancon and Ivan Herman. Circular drawings of rooted trees. Technical report, CWI (Centre for Mathematics and Computer Science) Amsterdam, The Netherlands, Amsterdam, The Netherlands, 1998.

[MK99] Mark Minas and Oliver Köth. Generating diagram editors with DiaGen. In *AGTIVE*, pages 433–440, 1999.

[ML03] Rafael Marti and Manuel Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Appl. Math.*, 127(3):665–678, 2003.

[MMSB01] Kim Marriott, Peter Moulder, Peter J. Stuckey, and Alan Borning. Solving disjunctive constraints for interactive graphical applications. In *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239, pages 361–376, London, UK, 2001. Lecture Notes In Computer Science, Springer-Verlag.

[MR02] Paul Mutton and Peter Rodgers. Spring Embedder Preprocessing for WWW Visualization. In *Proceedings Information Visualization 2002*. IVS, IEEE, July 2002.

[Mut97] Petra Mutzel. An alternative method to crossing minimization on hierarchical graphs. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 318–333, London, UK, 1997. Springer-Verlag.

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[Pat04] Pitch Patarasuk. Crossing reduction for layered hierarchical graph drawing. Master's thesis, Florida State University, 04-29 2004.

[PB03] Ernesto Posse and Jean-Sébastien Bolduc. Generation of DEVS modelling and simulation environments. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S139 – S146. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[PCJ97] H. C. Purchase, R. F. Cohen, and M. I. James. An experimental study of the basis for graph drawing algorithms. *Journal of Experimental Algorithmics*, 2(4), 1997.

[PdLV02] Ernesto Posse, Juan de Lara, and Hans L. Vangheluwe. Processing causal block diagrams with graph-grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23 – 34, April 2002. Grenoble, France.

[PM99] Ken Perlin and Jon Meyer. Nested user interface components. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 11–18, New York, NY, USA, 1999. ACM Press.

[PMCC01] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *CRPITS '01: Australian symposium on Information visualisation*, pages 129–137, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[PT97] Achilleas Papakostas and Ioannis G. Tollis. A pairing technique for area-efficient orthogonal drawings. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 355–370, London, UK, 1997. Springer-Verlag.

[PT98] Achilleas Papakostas and Ioannis G. Tollis. Algorithms for area-efficient orthogonal drawings. *Comput. Geom. Theory Appl.*, 9(1-2):83–110, 1998.

[QE01] Aaron Quigley and Peter Eades. Fade: Graph drawing, clustering, and visual abstraction. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, pages 197–210, London, UK, 2001. Springer-Verlag.

[RR97] Mauricio G. C. Resende and Celso C. Ribeiro. A grasp for graph planarization. *Networks*, 29(3):173–189, 1997.

[San94] Georg Sander. Graph layout through the vcg tool. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 194–205, London, UK, 1994. Springer-Verlag.

[San95] G. Sander. Vcg – visualization of compiler graphs. Technical report a 01/95, FB 14 Informatik, Universität des Saarlandes, 1995.

[San96a] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, University of Saarlandes, CS Dept., Saarbrücken, Germany, 1996.

[San96b] Georg Sander. A fast heuristic for hierarchical manhattan layout. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 447–458, London, UK, 1996. Springer-Verlag.

[San96c] Georg Sander. Graph layout for applications in compiler construction. Technical Report A01/96, FB14 Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany, 1996.

[SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, 1991.

[ST81] Tagawa S. Sugiyama, K. and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[Ste01] Alexander Stedile. JMFGraph - a modular framework for drawing graphs in java. Diploma thesis, Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology A-8010 Graz, Austria, November 18 2001.

[SYTI92] Kazuo Sugihara, Kazunari Yamamoto, Kojui Takeda, and Mitsuyuki Inaba. Layout-by-example: A fuzzy visual language for specifying stereotypes of diagram layout. In *VL*, pages 88–94. IEEE Computer Society, 1992. Proceedings of the 1992 IEEE Workshop on Visual Languages, September 15-18, 1992, Seattle, Washington, USA.

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[TBB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1):61–79, 1988.

[TNB04] Alexandre Tarassov, Nikola S. Nikolov, and Jürgen Branke. A heuristic for minimum-width graph layering with consideration of dummy nodes. In Celso C. Ribeiro and Simone L. Martins, editors, *WEA*, volume 3059 of *Lecture Notes in Computer Science*, pages 570–583. Springer, 2004.

[VdL02] Hans L. Vangheluwe and Juan de Lara. Meta-models are models too. In Yücesan, E., Chen, C.-H., Snowdon, J.L., and Charnes, J.M., editors, *Winter Simulation Conference*, pages 597 – 605. IEEE Computer Society Press, December 2002. San Diego, CA.

[VdL04] Hans L. Vangheluwe and Juan de Lara. Computer automated multi-paradigm modelling for analysis and design of traffic networks. In *Winter Simulation Conference*, pages 249–258, 2004.

[VM94] Gerhard Viehstaedt and Mark Minas. Interaction in really graphical user interfaces. In Takayuki Dan Kimura Allen L. Ambler, editor, *Visual Languages*, pages 270–277. IEEE Computer Society Press, 1994.

[WPCM02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[YKC02] D. Yehuda Koren; Carmel, L.; Harel. Ace: a fast multiscale eigenvectors computation for drawing huge graphs. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 137–144, 2002.

[YYL03] Lu Yang, Wei Yan, and Xiaoming Li. A network topology auto-layout algorithm based on "concentric-arrange" model. CN and DS Laboratory, Peking University, China, 2003.

[ZZO01] Ke-Bing Zhang, Kang Zhang, and Mehmet A. Orgun. Using graph grammar to implement global layout for a visual programming language generation system. In *CRPITS'11: Proceedings of the Pan-Sydney area workshop on visual informtaion processing conference on Visual information processing 2001*, pages 115–121, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.