Visual Layout of Graph-Like Models

Tarek Sharbak

MhdTarek.Sharbak@uantwerpen.be

Abstract

The modeling of complex software systems has been growing significantly in the last years, and it is proving to be more effective and efficient than traditional programming in the analysis and design of big complex systems. However, with models growing both in variety and size, we face the problem of making a good layout and increase readability of the models in a wide range of formalisms, specifically when working with domain-specific models. In this project, we are going to treat models as graphs and use Statecharts to model the interactive behaviors of models. We are going to look at the various methods and techniques used in graph drawing with respect to readability and users interactions.

Keywords: *Graphs; Graph Drawing; Visual Layout; Domain Specific Modeling*.

Table of Contents

Visual Layout of Graph-Like Models

Visual formalisms are used to create models of problems and simulate the real world. The main difference between visual formalisms and non-visual ones is obviously the fact that visual formalisms use graphical icons and arrows to represent the entities of the model. These icons and arrows have a visual layout in from which the user can extract information. Users should be able to understand these models at a glance and a poor layout would make a model very difficult to read, this bring the need for good layout support in the visual modeling tools.

Most visual modeling tools use hard-coded and inflexible layout behavior. However, the main goal here is to come up with a framework to model the reactive behavior of visual modeling environments that support multiple formalisms. The models are considered to be graphs and to come up with a good graph drawing algorithms; one has to have a thorough knowledge of graph theory and the existing literature of graph drawing techniques.

Graph Basics

Problems of the real world can be analyzed as models of a certain formalism. Any given formalism constraints the models that can be derived from it, such that the user will only be able to create models that correspond to certain formalism and prevent them from creating invalid models. These models are nothing but constrained graphs, and the constraints reflect in the behavior of the graph; some graphs are directed graphs where the edges between the vertices have a certain direction and an edge between vertex A and vertex B is different than the one between B and A. In addition to that, a model can be constrained by how many vertices is allowed to be created in the model, etc…

Visual Aesthetics

Visual aesthetics are the measurable qualities of a drawing. Different domain-specific formalisms require different metrics of the visual aesthetics, and the graph drawing techniques try to reach an optimal drawing for different formalisms.

Graph Area

Humans have a better comprehension of graphs that are displayed on one page rather than more than one page, which will require the frequent change of pages to understand the full graph. Similarly, the respect for aspect ratio in a graph drawing is very important because humans have a preference to views that correspond to the golden ration (1.618) like 4:3 and 10:6 which is even closer.

Vertex Placement

When vertices are not point-sized, they should not overlap in a drawing and they should be placed uniformly, which will minimize the graph area. Symmetry is also another vertex placement criteria but it was not proved that it increases readability.

Edge Crossings

One of the most important problems in graph drawing is edge crossings. It adds difficulties when extracting information from the graph and following the direction flow. Additionally, the angle made by edge crossing is very important for readability, where a 90-degrees angle makes easy to distinguish edges, whereas a small angle, makes following the edges very hard.

Edge Bends

A straight edge is much easier to follow than a bent one, but to avoid edge crossings; edge bending can be a good solution. Therefore, the graph drawing technique should try to keep the edge bends to a minimum.

Direction of flow

When the directed edges move in one flow direction, the user can more easily follow the direction flow of the entire graph, which shows the graph structure and hierarchy if applicable.

Edge Length

A long edge is far more difficult to follow than a short edge. Therefore, the edge length should always be as short as possible.

Mental Map

Users build a mental map when working with visual objects. It helps them remember where each object is and retrace the connections faster and easier. Graph drawing should always try to keep that mental map unchanged as much as possible. Otherwise, users will have to spend more time building different maps every time a model changes.

Vertex Connections

As mentioned before, the angle between two neighboring visual objects can largely enhance how user distinguish different objects. A larger angle between multiple edges that are connected to the same vertex makes differentiating between these edges and their arrows easier to the viewer.

Graph Drawing Techniques

The visual aesthetics sometimes contradict between one another, for example, eliminating edge crossings might require more edge bends, and maintaining the mental map of the model might require long edges and non-uniform direction of flow. Therefore, different graph drawing techniques try to optimize different criteria as much as possible. In this section, we identify the following drawing techniques.

Layered

Layered graph drawing technique is widely used because it offers a relatively easy implementation and covers a wide range of visual aesthetics. It however constraints the graph to be a digraph (directed graph), to have an overall direction of flow, and to be acyclic (does not contain any cycle). Graphs with cycles can be pre-processed to remove cycles before drawing. Layered drawing consists of the following steps, layer assignment, crossing minimization, horizontal placement, and extensions.

Force-directed

This technique is based on virtual physics models. Since the behavior of the physical objects is known to result in a good state, the simulation of graphs as these physical objects will also yield a good layout. This technique simulates vertices as molecules and virtual forces created where edges exist.

Orthogonal

Orthogonal drawings are typically drawn as a grid where vertices and edges are assigned integer numbers as coordinates, and they are connected with horizontal and vertical lines. We distinguish two main techniques, the pure orthogonal (vertices has degrees ≤ 4) and the quasi-orthogonal layout (no restrictions on the degrees).

Orthogonal drawing techniques produce good layouts because they optimize a wide range of visual aesthetics. They are divided into three phases, topology, shape, and metrics.

Linear Constraints

The main use of linear constraints is in the layout of windows in user interfaces, which makes the standard implementation of such a technique not sufficient for graph layout. Linear constraints provide a declarative approach to layout, and it requires a linear solver to work.

Many implementation of linear constraints have been done for graph layout and even the use of a non-linear solved has been introduced to solve non-linear problems.

Expensive Methods

Other methods for graph drawing are very computational expensive. However, some of these techniques are ideal for certain formalisms. These methods are, Simulated annealing, Genetic algorithms, and Rule-based techniques.

Other Techniques

Less common but used techniques for graph drawing are 3D layout, Circular, Competitive learning, Multi-dimensional, Graph grammars, Edge routing, and Graph browsing.

Graph Drawing Implementations

A good visual modeling tool should provide methods to make the drawing and reading of graphs easy, because models are as useful as how readable they are. AToM^3, a tool that supports multi formalisms, supports all of the automatic layout techniques that are mentioned in this section. The drawing algorithms that are implemented in AToM^3 all work through an abstraction layer that is designed in the tool to allow easy plugging in of other drawing algorithms.

AToM3

A Tool for Multi-formalism Meta-Modeling. AToM^3 is used for modeling, meta-modeling, and transforming models with graph grammars. It can also be extended for the simulation and code generation from models, which makes the tool highly extensible.

Graph exports and imports

Models in AToM^3 can be exported to GML, GXL, or DOT format, and existing graph tools can be used to draw the graphs. However, exported models lose significant amount of information when transformed.

Simply exporting graph layout to another freely available tool is insufficient and mostly limiting. Also exporting the model into another format, use another tool to process the graph and re-import the graph to AToM^3 is a troublesome process for the user.

Abstraction layer design

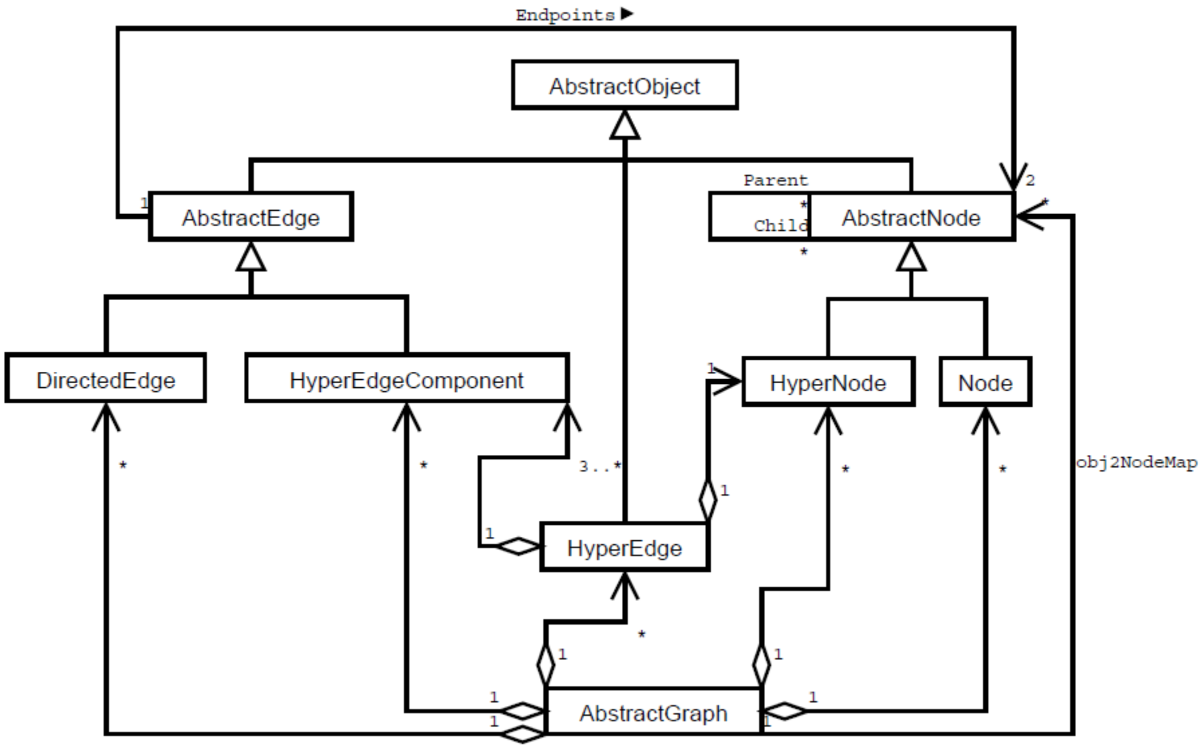

Figure 1: Abstraction layer class diagram

An abstraction layer is built in AToM^3 to provide an interface for automatic layout algorithms. This abstract layer has two main benefits, isolate the layout techniques from the internal data structure of AToM^3, and make these algorithms as reusable libraries. A class diagram of the layer is shown in Figure 1.

The following auto drawing techniques are implemented in AToM^3,

- Layered.
- Spring-embedder (Force-directed).
- Force-transfer.
- Tree-like and circle.
- Linear Constraints.

Formalism-Specific UI and Layout Behavior Using Statecharts

Tools like AToM^3 support multiple formalisms require more robust and dynamic layout behavior algorithms. AToM^3, which philosophy is model everything, uses a generic user-interface behavioral model in statecharts to model the behavior layout of the basic visual modeling environment. This generic model can be later refined by layout behavior of a specific formalism.

Domain specific modeling is good because it allows the modeler to analyze the system within the specific mental model of the problem, and it also constraints the modeler and allows only valid models to be created. In AToM^3, all four aspects of any given formalism are modeled explicitly, the abstract syntax and the concrete syntax of the formalism are static in nature, so they are modeled using Class Diagram or Entity Relationship formalism. The operational semantics and the reactive behavior on the other hand are dynamic, and they are modeled mostly using graph transformations. The reactive behavior defines how a certain model reacts to a sequence of input events, like mouse or keyboard clicks.

Generic UI Behavior

The entire generic-layout behavior of AToM^3 is shown in figure 2 created using statecharts. This approach has the advantage of being easily modifiable and completely isolated from other layout behavior models, such an advantage that would be difficult to do if the user interfaces were hard-coded.

Formalism-specific Behavior

Formalism-specific reactive behavior allows for easy modification to the parts of the generic-layout behavior that require special implementation for the given formalism. To do so, we need to identify a formalism scope within the environment so that the application's main loop would direct the event to the specific behavior layout when the mouse cursor is inside that scope. We can achieve that by having a virtual entity in the formalism that contains the visual objects of that formalism and thus defines the scope for that formalism. However, some events require that the user moves the mouse to outside the scope of the formalism but with having the specific behavior to stay active, we can do that by introducing "locks", which can simply lock the event loop and effectively direct all input to the specific layout behavior.

Pre/post UI observers

Some event such as delete of a certain entity would happen when the user clicks the delete button after selecting the desired entity, hence that the mouse coordinates is not necessary inside the scope of the deleted entity, or entities that belong to different scopes are selected. To solve this problem, pre and post UI observers are introduced, where the pre-observer captures the input before the main application loop and the post-observer captures them afterwards. Hence that, the pre/post observers are only observers and their sole purpose is to direct the input to the correct layout behavior, and they do not handle events. Otherwise, conflicts would occur in multi formalisms environments.

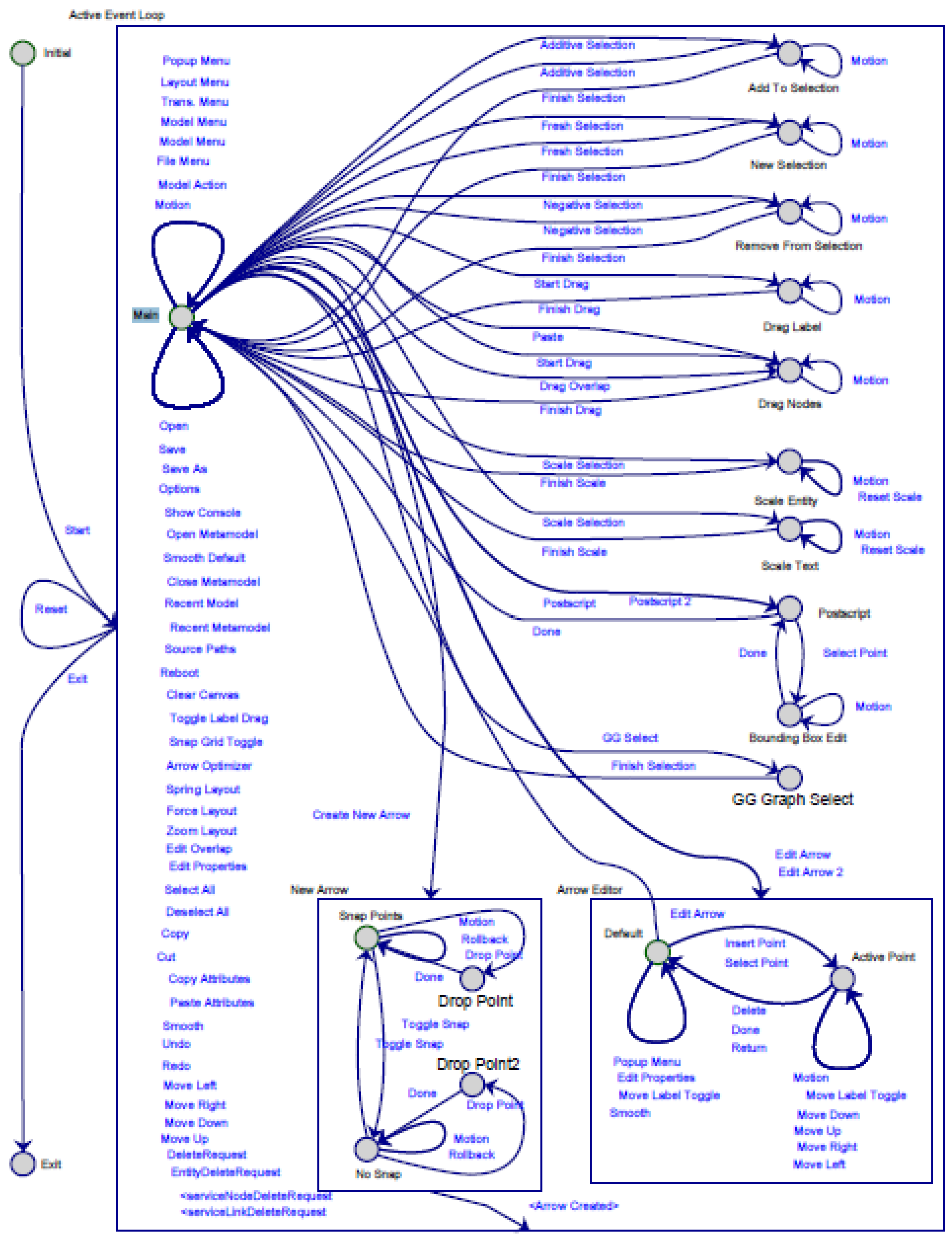

Figure 2: Generic user-interface behaviour statechart

Reference

Dubé, Denis. "Graph Layout for Domain-Specific Modeling." (2006): 107.