Visual Layout of Graph-Like Models

Tarek Sharbak

mhdtarek.sharbak@student.uantwerpen.be

University of Antwerp, Belgium

Abstract

Visual modeling makes designing and implementing complex systems an efficient process and drastically decreases errors and complexity. Moreover, Domain-Specific modelling constrains the modeler to only use what the language offers, which makes the model limited to the problem set in hand and closer to the mind-set of the specialized modeler. However, complex systems models tend to have many visual elements and thus can be difficult to comprehend if not properly laid out.

Keywords: Statechart, Graph, Domain-Specific Language

---

1. Introduction

Visual formalisms are used to create models of problems and simulate the real world. The main difference between visual formalisms and textual 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 from which the user can extract information. Users should be able to understand these models at a glance. Therefore, a poor layout would make a model very difficult to read, this brings the need for a good layout support in 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 multi-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.

In section 2 and 3, I summarize the thesis of Denis Dubé [1] and discuss the layout of graph-like models and the aesthetics that need to be taken

into consideration to make models easier to understand. In section 4, I will lay out the details of the implementation that I did based on "Rapid Development of Scoped User Interface" [2] [3], in which I used statecharts to model the behavior of the different visual elements of the simple RPG implantation that is written in Python. And a conclusion will be made in section 5 of this paper.

2. Graph Basics and Layout

Domain-Specific models of a certain formalism are constrained according to the design and purpose of that formalism, such that the user will only be able to create models that confirm to the Domain-Specific language being used. Models can be viewed as graphs, and the constraints of the models are reflected as different types of graphs; some are directed graphs where the edges between the vertices have a certain direction and an edge between vertex A and vertex B is distinguished from the edge 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…

2.1. Visual Aesthetics

Visual aesthetics are the measurable qualities of a drawing. A good graph drawing technique is the one which optimizes the use of these aesthetics to reach an optimal drawing. Different Domain-Specific formalisms require a specific balance of these aesthetics, which requires the use of multiple drawing algorithms in multi-formalisms visual modeling tools like AToM^3 and now AToMPM. Visual aesthetics are [1]:

- Graph Area
- Vertex Placement
- Edge Crossings
- Edge Bends
- Direction of Flow
- Edge Length
- Mental Map
- Vertex Connections

2.2. Graph Drawing Techniques

As mentioned before, different drawing algorithms are to be used to reach an optimal model drawing for each specific type of models, I review here some of these algorithms [1]:

- *Layered:* Widely used and offers 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).
- *Force-directed:* This technique is based on virtual physics models. The simulation of graphs as these physical objects will 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. Orthogonal drawing techniques produce good layouts because they optimize a wide range of visual aesthetics.
- *Linear Constraints:* Linear constraints provide a declarative approach to layout, and it requires a mathematical linear solver to work. Many implementation of linear constraints have been done for graph layout and even the use of a non-linear solvers 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 techniques for graph drawing like 3D layout, Circular, Competitive learning, Multi-dimensional, Graph grammars, Edge routing, and Graph browsing.

3. Formalism-Specific UI and Layout Behavior

Tools like AToM3, which supports multi-formalisms, require more robust and dynamic layout behavior algorithms. AToM3, which philosophy is to model everything, uses a generic user-interface behavioral model in statecharts to model the behavioral layout of the basic visual

modeling environment. This generic model can be later refined and extended by layout behavior of a specific formalism [1].

Domain-Specific modeling allows the modeler to analyze the system within the specific mental model of the problem, and it constraints the modeler and allows only valid models to be created. In AToM3, 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. However, the operational semantics and the reactive behavior are dynamic, thus 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.

3.1. Generic UI Behavior

The entire generic-layout behavior of AToM3 is shown in Figure 1 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 [1].

3.2. Formalism-specific Behavior

Formalism-Specific reactive behavior allows for easy modification to the parts of the generic-layout behavior that require special implementation. To reach that goal, 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 of that formalism. However, some events require the user to move the mouse outside the scope of the formalism while keeping this scope 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 statechart.

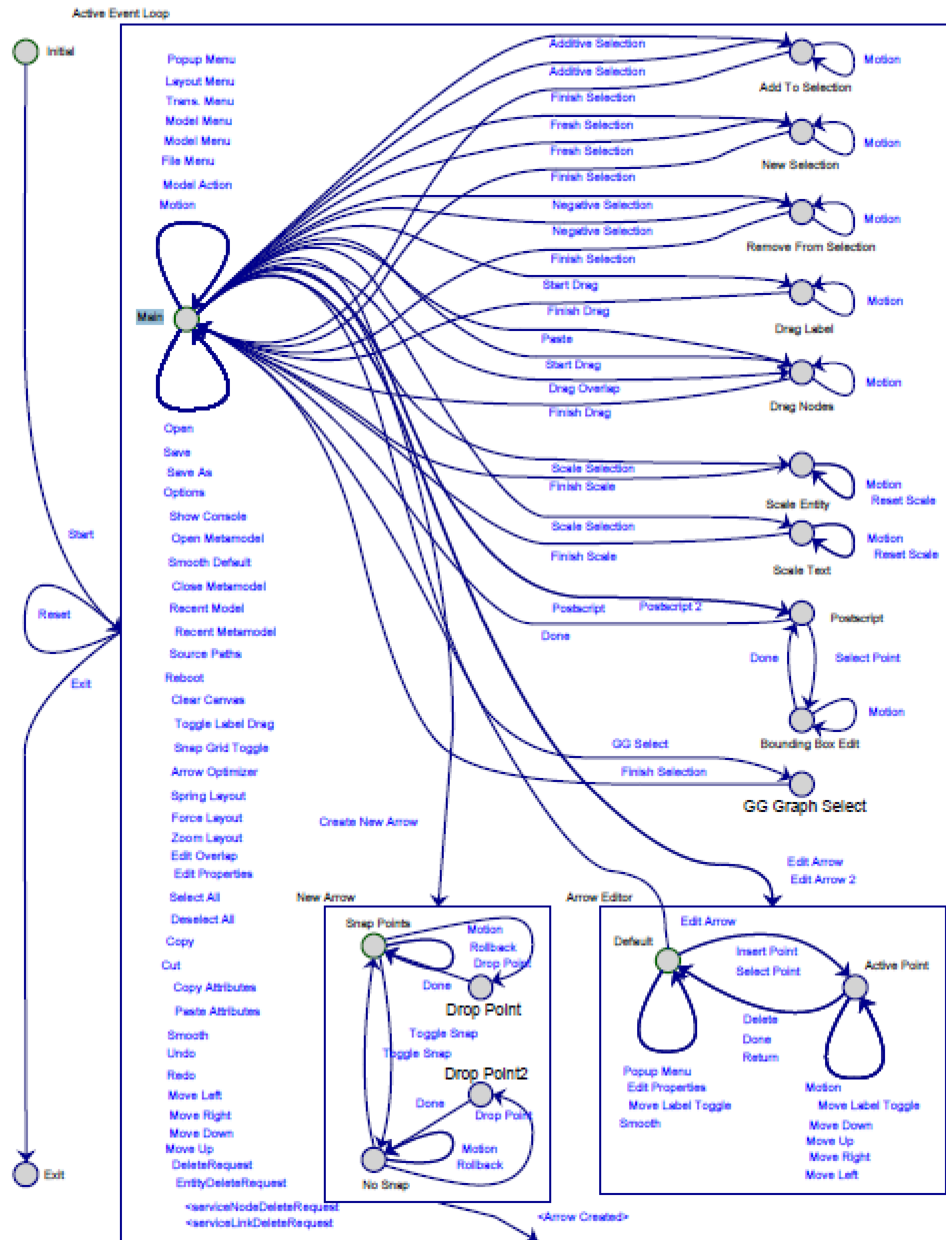

Figure 1: Generic user-interface behavior statechart

4. Implementation

For this project I based my work on the work done by Detlev Van Looy [3]. I followed the path that he followed by creating a scoped UI statechart that reflects the behavior of the RPG game and then I modified the drawing code accordingly.

4.1. Role Playing Game

For this project I had to recreate a simple RPG in Python, consisting of the abstract syntax of the entities of the RPG as classes (Scene, Standard Tile, Obstacle, Hero, and Goal). After that, I created a simple GUI that will be the modelling environment of the RPG game, Figure 2 shows the buttons used by the GUI to draw the elements of the game on the canvas.

Figure 2: Button controls of the GUI

The user can click on any of the buttons and then click a combination of keys (Ctrl + Right Mouse Button) to draw the selected element on the canvas. This behavior is specified using the scoped statecharts and will be shown later. Some entities have associations between each other (Scene contains Tiles and Tiles contains Heroes or Goals), these associations can be created by clicking (Ctrl + Left Mouse Button) and then clicking on the source element and dragging the edge to the destination and then pressing (Left Mouse Button) to create the association.

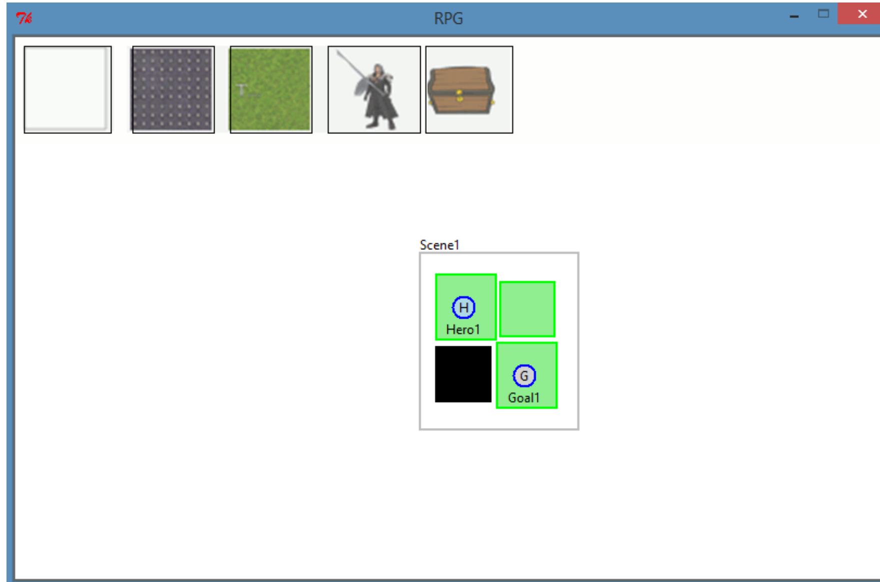

Figure 3: Showing the GUI of the visual development interface of the RPG

4.2. Scoped Behavior Statechart

Implementing the UI behavior using statecharts allowed for a very flexible implementation and extremely high reusability of the models. The behavior statecharts were implemented using AToM[3]. The main component of the statechart shown in Figure 5 handles the main loop of the program and its actions are activated when the user clicks on any of the controls in Figure 2, according to the selected button, the statechart will move to one of the modes (Standard, Scene, Goal, Hero, and Obstacle) where the user can then create on the canvas by pressing the combination (Ctrl + Right Mouse Button). When the user chooses another button, a "reset" event is sent to the statechart followed by the selected button action.

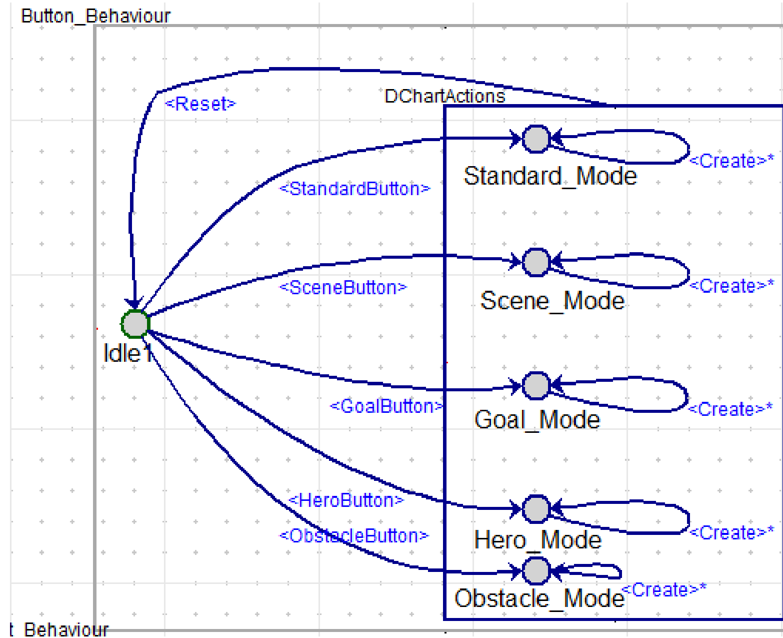

Figure 5: The Buttons behavior statechart

The second statechart in Figure 5 shows the behavior of the GUI when the user does one of the following actions (Selecting an entity, delete, move, create, deselect, or create an edge between two entities). Hence that the create state can only be reached if the previous statechart was not in the idle state, which means that the user has selected an entity to create from the button toolbox.

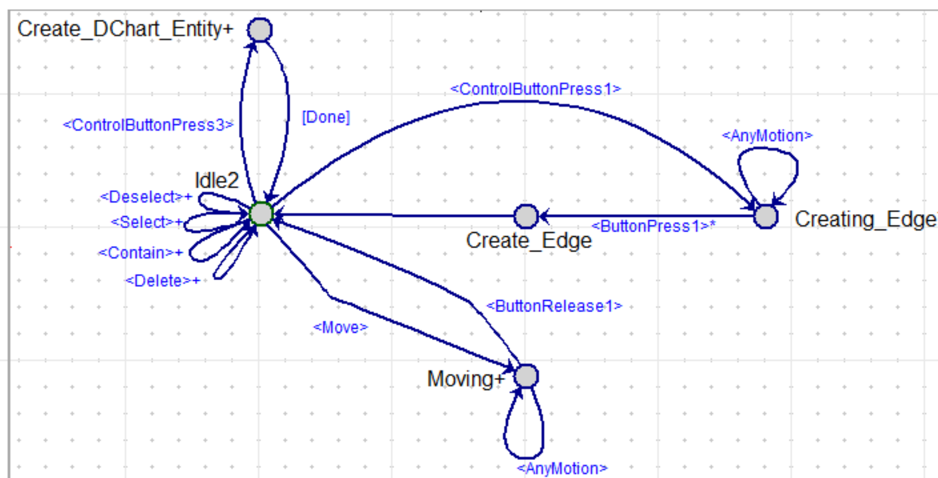

Figure 4: Statechart that handles (create, select, deselct, move, delete, create edge) actions

When the controller sends a select, deselect, contain, or delete trigger, a different specified statechart for each entity type, depending on the selected entity, receives that action and continues the work from there by sending the desired actions back to the drawing controller. This way of implementation showed me that I had to duplicate the statecharts as many as the controls I have. In my case I only had to do 5 duplicates (with minor changes) for each entity. However, when the project gets bigger, this implementation is inefficient and the need to use a more effective implementation arises. Another challenge was the need to hard-code the abstract and concrete syntax, which yielded in so many bugs and errors and thus were difficult to maintain.

5. Conclusion and Future Work

Complexity of defining a user interface behavior of models can be dramatically decreased by modeling the UI behavior using statecharts. Such technique insures high reusability and much less errors and bugs in the design. However, the approach that I took in this project was not the optimal solution for the problem at hand. A layered based implementation with a generic UI behavior and extending that with formalisms specific behaviors for each entity in the models as described in Dubé's thesis [1] would have been the better choice. Therefore, as a future work for this project, I suggest modelling the abstract syntax and concrete syntax of the RPG instead of hard-coding them, which can help with the verification of the UI elements over their corresponding definitions, and then dividing the behavior statecharts into a more well-defined layered behavior statecharts, where all entities have a shared behavior, which will eliminate the redundancy in my implementation, and a specific behavior statecharts for the extended behaviors of these entities.

References

[1] D. Dubé, "Graph Layout for Domain-Specifc Modeling," 2006.

[2] D. Dubé, J. Beard and H. Vangheluwe, "Rapid Development of Scoped User Interfaces".

[3] D. Van Looy, "UI Development Using Statecharts".