UI Development Using Statecharts

Detlev Van Looy

University of Antwerp

Abstract

In this paper, I summarize "Rapid Development of Scoped User Interfaces"[3]. I then explain how I created a simplified example modelling a Scoped UI for creating statecharts using hierarchically linked statecharts in detail and report my findings.

Keywords: scoped UI, hierachically linked statecharts, summary, example

1. Introduction

When developing a complex User Interface, there may be many different components which may all have very different behaviour and relations. A user interface should also be easy to maintain and adapt, to keep up with constantly changing system requirements. The problem is that code-centric implementations of a UI are no longer adequate. An interface developer needs to try to minimize "accidental complexity" [2]. The author of the original paper [3] claims that "an elegant solution to these problems may be found in Multi-Paradigm Modelling [4]. By modelling every aspect of the system-to-be-built, at the most appropriate level of abstraction, using the most appropriate formalisms, it becomes possible to completely capture the structure, behaviour and visual appearance of a UI, to rapidly generate prototype implementations, to easily adapt the UI as project requirements change, and, finally, to synthesize a UI and maintain it." I will further test this claim by checking for myself that the use of Hierarchically-linked Statecharts to model Scoped User Interfaces does indeed keep things simple.

There have been many works about the modeling of User Interfaces. Works using HlS in particular are obviously the original paper, and also

Email address: Detlev.VanLooy@student.ua.ac.be (Detlev Van Looy)

"Modelling the Reactive Behaviour of SVG-based Scoped User Interfaces with Hierarchically-linked Statecharts" [1].

In section 2 I give the summary of "Rapid Development of Scoped User Interfaces"[3]. The following section explains how I implemented the UI. Then section 4 shows what the UI looks like and how its used. The next section compares my implementation with that of others. Finally I give conclusions and possible future work.

2. Summary

The following is a summary of "Rapid Development of Scoped User Interfaces"[3].

2.1. Introduction

Complex behavioural relationships between UI components can be difficult to express, encode and maintain. A developer must also be able to quickly adapt to changing system requirements. A developer must minimize accidental complexity (No Silver Bullet). The project is about facilitating rapid, domain-specific modelling of the UI, and believes this may best be achieved by explicitly modelling the behaviour of each individual UI component. The approach used is different from related research in two ways:

1. It attempts to solve the problems of UI development by casting it as a pure language engineering problem.
2. It is primarily concerned with modelling the reactive behaviour of the class of user interfaces that are made up of hierarchically-nested entities.

2.2. Scoped User Interfaces

A Scoped User Interface is one in which reactive visual components such as buttons and windows, but also domain-specific entities, are hierarchically nested. A Scoped UI, then, is one which has a notion of hierarchical scope, and can bind an event to the most tightly-binding component in a hierarchy, based upon event coordinates. We focus on domain/formalism-specific modelling environments which have the potential to greatly improve productivity as they:

1. Match the user's mental model of the problem domain;

2. Maximally constrain the user (to the problem at hand, through the checking
3. Domain constraints) making the language easier to learn and avoiding modelling errors by construction";
4. Separate the domain-expert's work from analysis and transformation expert's work;
5. Are able to exploit features inherent to a specific domain or formalism. This will for example enable specific analysis techniques or the synthesis of efficient code exploiting features of the specific domain.

Two challenges when developing Scoped UIs:

1. Describing the interaction between user on the one hand and the various entities in the UI on the other hand. Solve this by considering the entities as actors.
2. Avoid creating an entirely new specification of UI behaviour for each formalism. Solve this by having a single generic specification at the root level. Scope-specific modifications can then be made to this generic specification.

2.3. Hierarchically-linked Statecharts

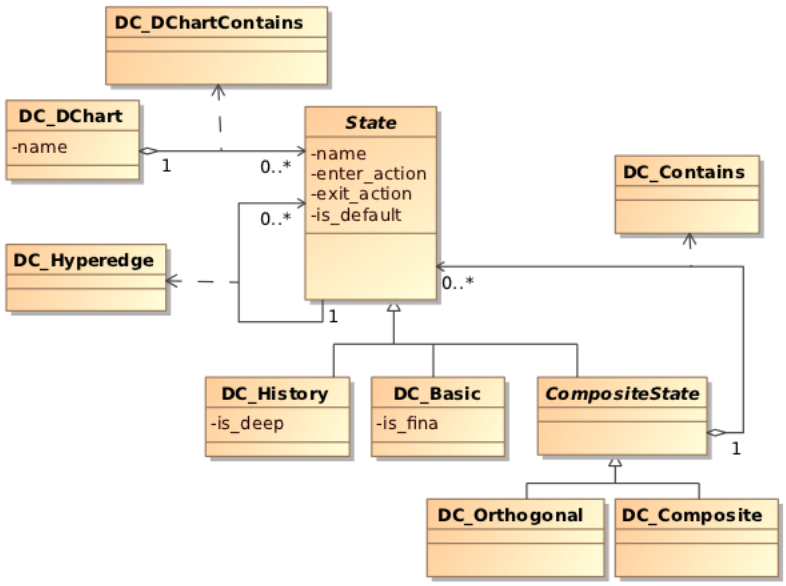
Hierarchically-linked Statecharts (HlS) is a formalism for visually describing the structure and behaviour of Scoped UIs. HlS make it easier to develop applications with complex UI behaviour faster and more reliably. Specifically, HlS entails the following workflow:

1. One uses an appropriate formalism, such as UML Class Diagrams, to specify the Abstract Syntax of the visual language. This entails specifying all elements in the domain one wishes to model, and qualifying their relationships with other elements. This Class Diagram, together with constraints over its elements is commonly known as a meta-model.
2. Subsequently, one models the Concrete Visual Syntax by associating a visual entity (such as an iconic shape) of the application being developed.
3. One finally specifies UI behaviour using Statecharts, such that each Statechart is associated with a class and specifies the reactive behaviour of each instance of that class.

2.4. Example

For the implementation, the AToM tool is used.

Figure 1: DCharts Meta-model in the Class Diagram formalism

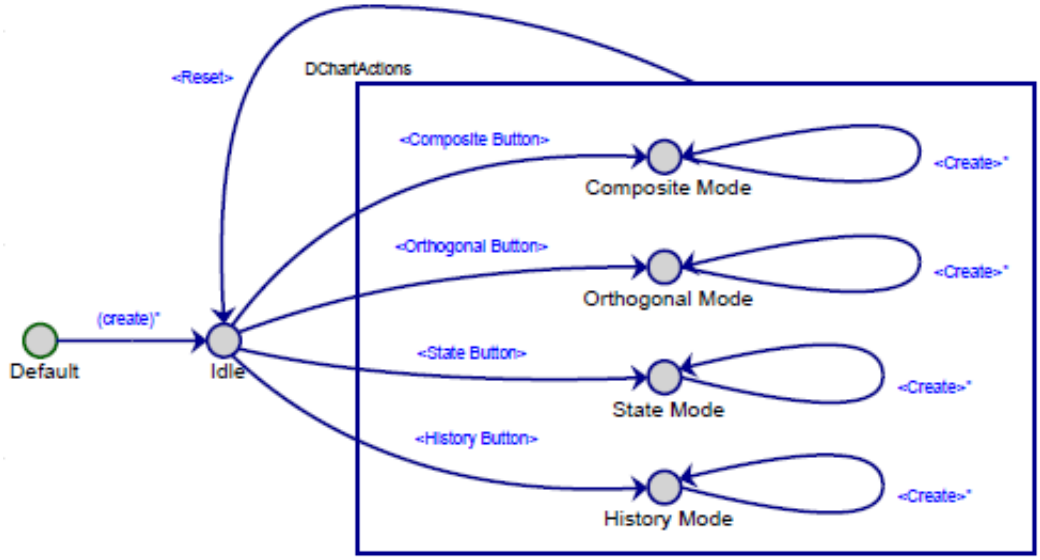


2.4.1. Specifying DCharts Abstract and Concrete Syntax

The abstract syntax of the DCharts visual language, is shown in Fig.1. The concrete syntax is known from using DCharts for the wristwatch assignment in MoSIS.

- DC DChart is a representation of the entire model. All other entities will be contained by this entity.
- DC Basic corresponds to a simple state that does not hierarchically contain others.
- DC Composite is nearly identical to DC Basic. A major structural difference is that it can contain other states.
- DC History is the history (pseudo-)state.
- DC Orthogonal is an orthogonal block that allows for concurrently active states.

Figure 2: Button Behaviour Statechart



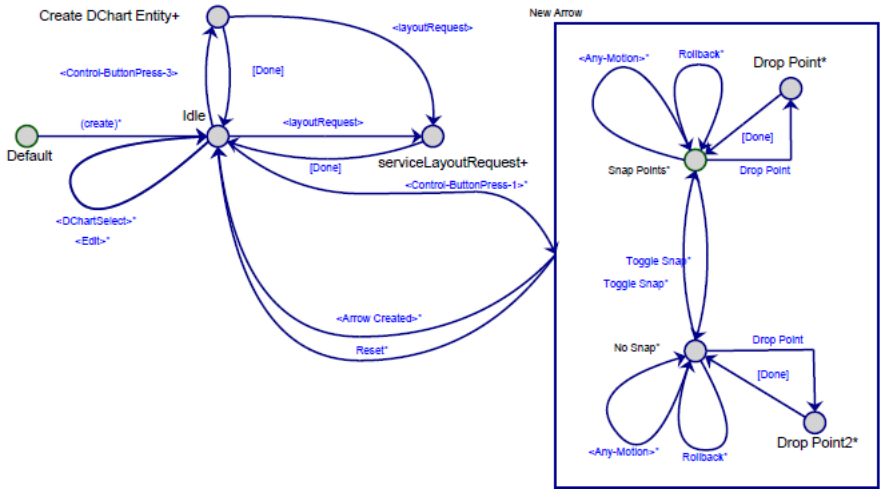
2.4.2. Specifying Formalism-specific Behaviour Using DCharts

The above describes the abstract and concrete syntax of the visual language, however we still need to model the behaviour of a language-specific visual modelling environment. In the following, the labels on the states and transitions of the UI behaviour Statecharts use a custom notation:

- 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.
- Regular brackets: (x) indicate the event was generated by the initialization routine for the entity when it is first instantiated.
- Square brackets: [x] indicate that the event was generated by the Statechart itself, usually within the action code of a state.

Button Behaviour model. Shown in Fig.2. When the button to create entity X is pushed, the events <Reset>" and <X Button>" are sent to this Statechart. <Reset> takes us to state Idle. <X Button> moves it to

Figure 3: DC DChart behaviour Statechart



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.

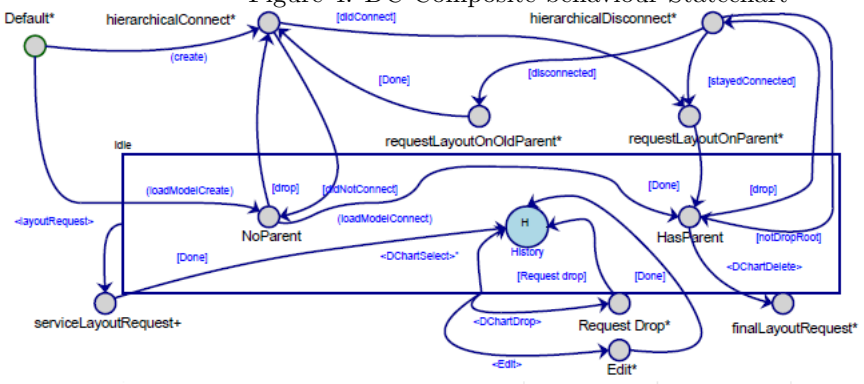
DChart entity-specific behaviour models. All visual entities of the DCharts formalism require their own behaviour models. The most important are the root entity that contains all other entities, and the composite state.

DC DChart behaviour Statechart. Shown in Fig.3. Following events trigger interesting behaviour:

1. <Control-Button-Press-3> indicates a new DCharts entity should be added to the canvas. The actual creation of the entity is handled by the Button Behaviour model above.
2. <Control-Button-Press-1 triggers a modal lock, forcing all events to be routed only to this Statechart. The lock is released when an arrow is created <Arrow Created>* or when the process is aborted Reset*. For the convenience of the user, only transitions may be drawn, a drag-and-drop behaviour model exists for creating and destroying containment relationships (shown later).

DC Composite behaviour Statechart. Shown in Fig.4. The behaviour of DC Composite is the most complex, fortunately it is also re-usable by many

Figure 4: DC Composite behaviour Statechart



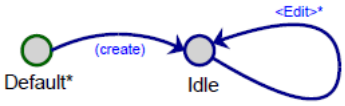
other entities. There are two possibilities for initialization:

1. An interactive session is in effect, the (create) trigger signals the creation of a new DC DChart. Immediately, the user is presented with a dialog asking to which of the entities in the region of the newly created DC Composite, they would like to contain the new composite state. If it is succesfully connected to 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 succesfully connected, a [didNotConnect] event is generated and the active state is set to NoParent.
2. The model is being loaded, a (loadModelCreate) event is first sent, setting the active state to NoParent. Then, a second (loadModelCreate)" 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:

1. The <DChartSelect>* event makes it so all hierarchival children are selected.
2. The <Edit> event indicates the user has opened an edit dialog for the DC Composites attributes. This may trigger requests for layout.
3. The <DChartDrop> event indicates this composite state, among potentially many other entities has just been dragged and then dropped. This generates [Done] to restore the state to NoParent or HasParent.

Figure 5: DC HyperEdge behaviour Statechart



Followed by [Drop], this makes it so hierarchical connection/disconnection is attempted. The latter occurs only when the entity has been dropped outside of its parent container and the user explicitly agreed to disconnect it. This triggers a <layoutRequest> followed by an attempt to hierchically connect the disconnected composite state in its new location.

4. The <DChartDelete> event indicates that this composite state has to be deleted.

DC HyperEdge behaviour Statechart. Shown in Fig.5. Trivially simple, a directed edge with one source and one target. Initialized with a (create) event, after that is awaits <Edit>* to apply changes made in its edit dialog. These affect the label associated with the transition.

2.4.3. Conclusion

It is now shown that it is possible to model complex, scoped, formalism-specific UI behaviour using HlS.

3. Implementation

In this section I will explain how I implemented the scoped UI for creating a statechart using HlS. I will do this one behaviour at a time, so first I will explain creation of states, then creation of edges, etc. Before I start explaining how I implemented this, I will first say what is lacking from my implementation:

1. I am missing the option to edit states/edges.
2. I am missing proper drawing of edges. A transition in my implementation is a straight line and always drawn from the middle of the righthandside of a State to the middle of the lefthandside of a State.

The main reason for this is because I felt that I had already done enough for this experiment to be able to draw conclusions. Another reason is that the process of adding additional behaviour is very repetitive and always follows

Figure 6: Buttons menu

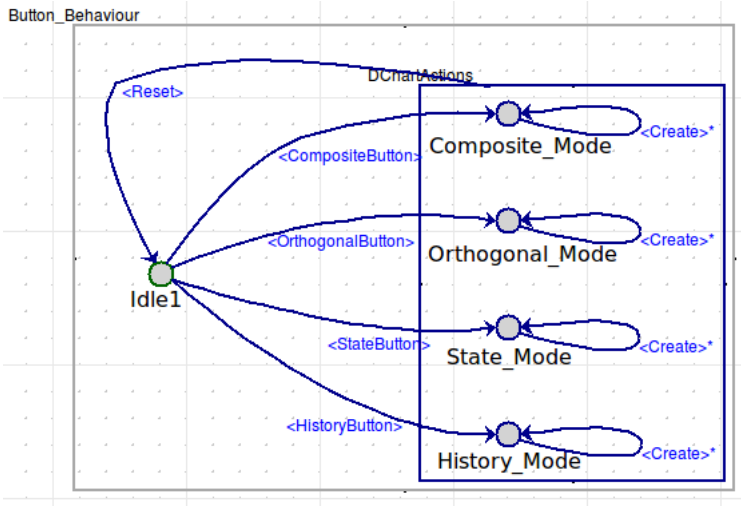
the same basic plan. This is explained below. Another thing to note is that for simplicity I put every UI component in the same statechart, using orthogonal states. Because I didn't have many components this was easier than making a new statechart for every component.

Buttons menu.
Before I can explain creation of entities I must first explain the behaviour of my button menu. When running my implementation, the first thing you will see is the buttons shown in Fig.6.

Clicking one of these will trigger an <XButton>and a <Reset>event in the Button Behaviour statechart (Fig.7) depending on which button was clicked. My Button Behaviour Statechart is exactly the same as in the original paper.

Creation of states. A state is created by ctrl+rightclicking on the canvas. This sends a <ControlButtonPress3>event to my DChart Behaviour

Figure 7: Button Behaviour Statchart

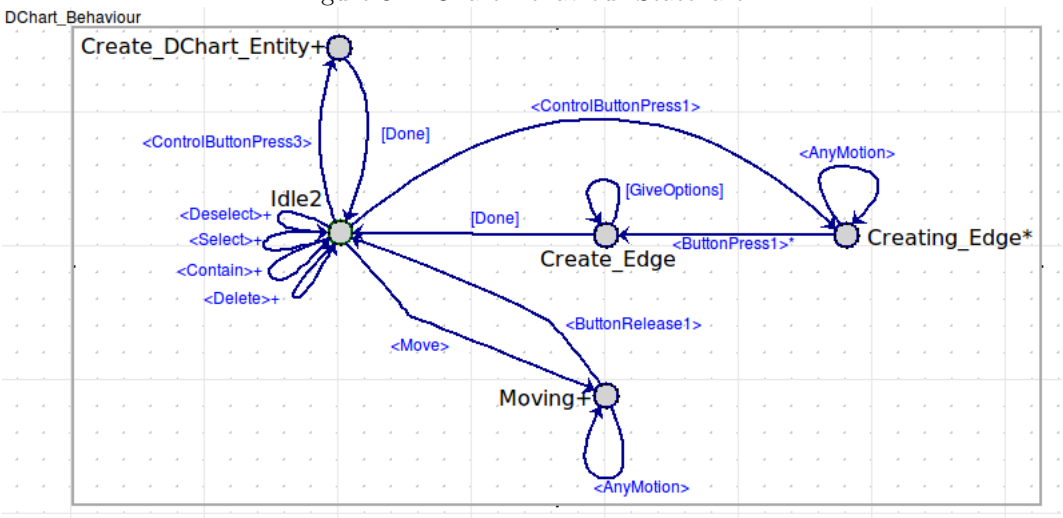


Statechart shown in Fig.8. This then sends a <Create>* event to my Button Behaviour Statechart, which will call the correct drawing method, depending on which mode it is in. In my implementation it was not necessary to make the Button Behaviour send another event to for example the basic state behaviour statechart, because at that point it is already known what has to be drawn, so in this example it would just call drawBasicState(). I've made it so states look almost exactly as in AToM3 to make it feel familiar.

Creation of edges. Creation of an edge is initiated by ctrl+leftclicking on a state, this sends a <ControlButtonPress >event to my DChart Behaviour Statechart putting us in state "Creating Edge*" and locking all input to this statechart. Every time we enter this state, a method drawShowLine() is called, this draws an edge from the clicked state to the cursor, indicating where the edge will go. Everytime the mouse moves we send an <AnyMotion >event, making us re-enter "Creating Edge*" and updating the showline. Clicking the left mouse key will then send a <ButtonPress1*>event. There are many options now depending on which state the transition originates from and what the user clicked. We call a method createEdgeHelper() to figure out what needs to happen, there are 4 options:

1. The user did not click on a state or a state to which an edge is impossible (for example a state can not contain itself) this will result in either

Figure 8: DChart Behaviour Statchart

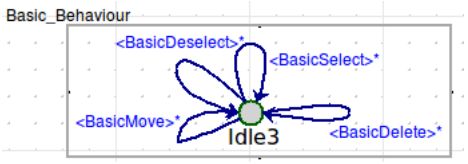


nothing happening or an error being shown, saying why the edge is impossible. A [Done] event is sent and we are done with the creation of the edge.

2. The user clicked on a state and we know for sure it has to be a transition (for example from one basic state to another basic state) this will result in drawTransition() being called. Afterwards a [Done] event is sent and we are done with the creation of the edge.
3. The user clicked on a state and we know for sure it has to be a contain (for example from an orthogonal state to a basic state which is not yet contained by another state) this will result in drawContainment() being called. Afterwards a [Done] event is sent and we are done with the creation of the edge. (Actually this will also send a <Contain>+ event, because we need to contain something, but this is explained later.)
4. The user clicked on a state and we can not know for sure what it has to be (for example from a composite state to a basic state which is not yet contained by another state) this will result in a [GiveOptions] event being sent, which asks the user whether he wants the edge to be a transition or a contain. The rest is then handled the same way as 2 or 3.

Selection of entities. The general plan for adding behaviour from now on is always the same:
following some action by the user, we send an event in our DChart Behaviour Statechart which then calls a method to decide what to do after which an event is sent to the appropriate Statechart. This statechart will then execute the necessary behaviour. Figures 9, 10, 11, 12, 13, show the behaviour for all entities.

Figure 9: Basic Behaviour Statchart



So for selection of entities, when the user leftclicks an entity, a <Select>+ event is sent to the DChart Behaviour Statechart. This calls a helper function typeHelper() to see which kind of entity was selected, based on that a select

Figure 10: Orthogonal Behaviour Statchart

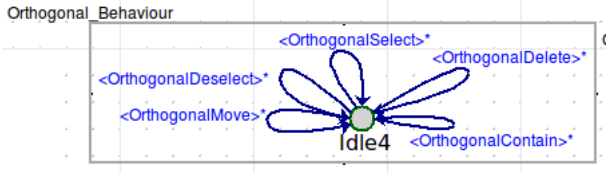
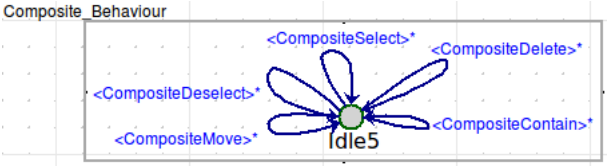


Figure 11: Composite Behaviour Statchart



event is sent to the corresponding statechart. This will draw the selection of that entity (turns the colour of it to red). Note that each type may have different behaviour, e.g. for orthogonal and composite states we also have to select everything which is contained within those states. This is done via recursion from top to bottom.

Deselection of entities. For deselection of entities, for example when the user leftclicks on a blank spot on the canvas, a <Deselect>+ event is sent to the DChart Behaviour Statechart. This calls a helper function typeHelper() to see which kind of entity has to be deselected (if any), based on that a deselect event is sent to the corresponding statechart. This will draw the deselection of that entity (turns it back to its original colour). Note that each type may have different behaviour, e.g. an orthogonal state has to turn grey and a composite state has to turn blue (or green if it is default). Again a recursive method handles the deselection of everything which is contained.

Deletion of entities. For deletion of entities, when the user presses the Delete key, a <Delete>+ event is sent to the DChart Behaviour Statechart. This calls a helper function typeHelper() to see which kind of entity has to be deleted (if any), based on that a delete event is sent to the corresponding statechart. This will draw the deletion of that entity (removes it from the canvas). When an entity is deleted all outgoing and incoming edges (including the invisible containment edges) are deleted as well. Deletion of an orthogonal or composite state does NOT delete the entities it contains. Note that each type may have different behaviour, e.g. an orthogonal state has to delete all outgoing containment edges while a basic state does not have

Figure 12: History Behaviour Statchart

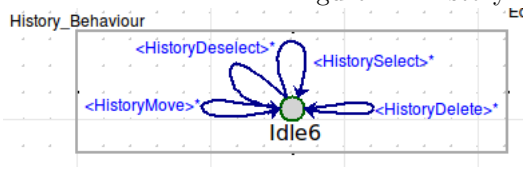
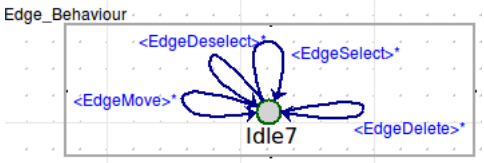


Figure 13: Edge Behaviour Statchart



those.

Containment of entities. Containment of an entity is when an orthogonal/composite state wraps itself around the entities it contains. This can happen in two cases:

1. A containment edge has been drawn
2. An item which is contained has moved

Note we do not send a containment event when an entity is deleted.

For containment of entities, a <Contain>+ event is sent to the DChart Behaviour Statechart. This calls a helper function typeHelper() to see which kind of entity has to do the containment (note that this can only be either an orthogonal or a composite state), based on that a contain event is sent to the corresponding statechart. This will draw the containment by the entity. Note that this is a recursive function which goes from bottom to top, because the state which is doing containment might be contained by another state, etc..

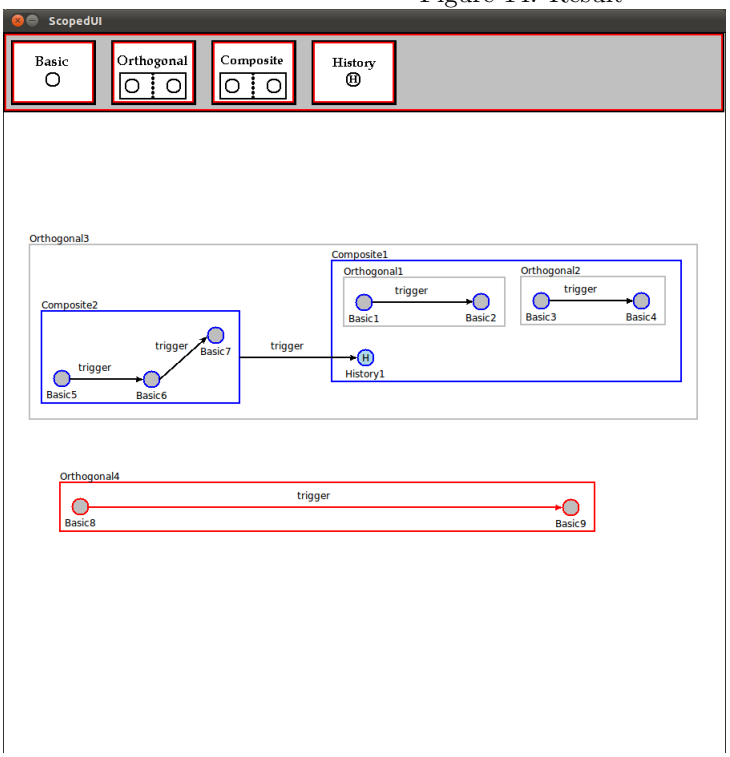
Movement of entities. For movement of entities, a movement is initiated when the user leftclicks (and holds it down) an already selected entity, a <Move>+ event is sent to the DChart Behaviour Statechart. This takes us to state "Moving+" and locks all input. Every time we enter "Moving+" it calls a helper function typeHelper() to see which kind of entity has to be moved (note that this can not be an edge, because we do not allow movement of edges by themselves), based on that a move event is sent to the corresponding statechart and to the Edge Behaviour Statechart. This will

draw the movement of the state and every edge which needs to be moved. Note that each type may have different behaviour, e.g. an orthogonal state has to move everything it contains. Again a recursive method handles the movement of everything which is contained. When we are in state "Moving+", everytime the mouse moves we send an <AnyMotion >event, making us re-enter "Moving+" and thus updating the drawing. When the user releases the left mouse button, a <ButtonRelease1 >event is sent followed by a <Contain >+ event, finalizing the movement of the entity.

4. The Result

An example of what the result looks like is shown in Fig.14. We see that the bottom orthogonal state is currently selected because it is outlined in red, together with the states it contains. I already mentioned what buttons

Figure 14: Result



to press for which behaviour in the section above, but I will give a short list of all buttons here:

- Left click one of the buttons at the top to start creating entities.
- Ctrl+right click on the canvas to create the desired entity.
- Left click on an entity to select it.
- Left click on the blank canvas to deselect the selected entity.
- Left click (and hold down) a selected entity to move it.
- Release left click to finish movement of an entity.
- Ctrl+left click on an entity to start the creation of an edge.
- Left click on an entity while creating an edge to draw an edge to this entity.
- Left click on the blank canvas while creating an edge to cancel the creation.

5. Comparison

In this section I will compare my implementation with the one from the original paper. The point of this paper was never to improve the original implementation, it was rather to create a simple implementation which mimicked the original. I think my result is satisfactory, the experience of using the UI is just as smooth as the original. The only lacking functions are the editing of states/edges and proper drawing of edges, but the drawing of edges is not something within the scope of this paper.

6. Conclusions and future work

The main conclusion of this paper is that the original authors were right about the minimization of accidental complexity. So much so that adding new behaviour to the UI became a rather quick task, easily divided into concrete steps. Finding bugs or slightly tweaking the UI was transparent. Another well known fact about software is that it is almost always easier to imagine what is going to happen when there is a visual representation available.

Future work to this project in particular would probably see to the addition of functionality to edit states/edges and proper drawing of edges. For

future work regarding the general idea behind this paper I believe the original authors make a nice claim, namely "We believe HlS can be used as the assembly language for UI behaviour modelling."

References

[1] Beard, J., Vangheluwe, H., 2009. Modelling the reactive behaviour of svg-based scoped user interfaces with hierarchically-linked statecharts.

[2] Brooks, F., 1987. No silver bullet: Essence and accidents of software engineering.

[3] Denis Dube, J. B., Vangheluwe, H., 2009. Rapid development of scoped user interfaces.

[4] Mosterman, P., Vangheluwe, H., 2004. Computer automated multiparadigm modeling: An introduction.