Statechart modelling of visual modeling environments' behaviour

Xia Shengjie

2005.Feb.29

Abstract

This report first introduces an important classification framework of visual language. Each class of the framework defines a family of visual language according to the characteristics of their graphical objects and composition rule. Some visual language has been implemented based on the given class of framework by using the Visual Language Compiler-Compiler(VLCC) system, a graphical system that can automatically generate the visual programming environment. The second part presents the concept and the notation of the statechart, a powerful and expressive state-based visual language used for modelling the behaviour of a part of system. The last part demonstrates how to construct the user interface of DChart Editor with statechart using the user interface-control-model(UCM) architecture.

Part I
A Classification Framework of Visual languages

1 Visual Language Syntax

A visual language may be seen as a set of visual sentences made up by graphical objects (the vocabulary). Syntax of visual language is descibed through the vocabulary, the relations used to make the sentences, and a set of rule used to define visual sentence. The graphical objects have three types of attributes, i.e. graphical attributes, syntactic attributes and semantic attributes.

Graphical attributes describe the appearance of the object, such as position, size, shape, color, etc. Syntactic attributes are used to relate graphical objects to make the sentences. Only when the syntactic attributes of a set of graphcal objects have been instantiated, the sentences are formed. Semantic attributes are used to associate semantics to a graphical object.

2 Class of Visual Languages

According to the type of syntactic attributes of the graphical objects and the type of the relation used to form the visual sentences, the visual language classes can be divided into two catagories: Connection- and geometric-based.

2.1 Connection-based Classes

The graphical objects connected to each other compose the sentences. Their syntactic attributes can be single attaching points or sets of attaching points. We refer to them as attaching regions. The former is the special case of the latter.

Connections can be achieved in major two ways, i.e.overlapping and linking.

2.1.1 Class Plex

One way to model graph-structured visual language is by using the class *plex*. The structure of some visual languages like flowcharts, chemical structures, Boolean and electric circuits can be described by it very well.

This class has the limitation on the number of the connections for each terminal graphical object. The graphical objects of this class is refereed to as NAPEs (N attaching Point Entities)[2].

Syntax attributes: a pre-defined set of attaching points.

Relations: plex interconnection denoted by ***JOINT*** and defined as follows.

Definition 1. Given the NAPEs a and b, then a JOINT(h,k) b holds if and only if the attaching point of a is connected to the attaching point k of b. [1]

2.1.2 Class Graph

The class graph is suitable for modeling the general graph-structured visual languages including data flow graphs, state transition diagrams, Petrinets, entity-relationship diagrams, SADT diagrams, Class and Object diagrams.

Syntax attributes: a pre-defined set of regions known as attaching region.

Relations: Graph interconnections, denoted by ***JOINT*** and defined as follows.

Definition 2. Given the Graphical objects a and b, then a JOINT(h,k) b holds if and only if the attaching region of a is connected to the attaching region k of b. [1]

2.2 Geometric-based Classes

In geometric-based class of visual language a visual sentence can be composed by spatially arranging a set of graphical objects in the Cartesian plane according to some spatial composition rule.

2.2.1 Class String

This class is used to model all the string languages. It is linear case of a geometric class.

Syntactic attribute of a character: An integer denoting its position within the string.

Relations: The string concatenation, denoted by CONC which is defined as below.

Definition 3. Given the characters a and b with positions i and j, then a CONC b holds if and only if j = i + 1. [1]

2.2.2 Class Iconic

The graphical objects of this class are icons. An icon can be seen as an image within a box of fixed size.

Syntactic attribute: A pair of integers denoting the coordinates of the icon in a grid of cells of the same size as an icon bounding box.

Relations: iconic spatial composition REL, defined as follows.

Definition 4. An iconic spatial compostion REL includes a set pair of integers(m,n). Given a relation REL and tow icons a and b with syntactic attributes(x, y) and (h, k), then a REL b holds if and only if if $\ni$ (m,n) $\in$ REL such that h=x+m and k=y+n. [1]

2.2.3 Class Box

The typical visual language modeled by this class is Nassi-Schneidermann diagrams. This diagram is used to describe the computer program. The class box is also suitable to model high-level interface language for geographical information system and the visual calendar.

The graphical objects of this class have different shapes, and can be syntactically described and manipulated by their bounding box with variable size.

Syntactic attribute: Pairs of integers denoting the coordinates of the upper-left and lower-right points of the graph object's bounding box.

Relations: box spatial composition REL, which generalize the iconic composition. In order to define the relations, we use two functions $UL_{REL}(m, n, m1, n1)$ and $LR_{REL}(m, n, m1, n1)$ represent respectively the coordinates of the upper-left(m, n) and lower-right(m1, n1) points of the bounding box of a box a.

Definition 5. Given a box a with syntactic attributes (x, y) and (x1, y1) and a box b with syntactic attributes (h,k) and (h1,k1), then a REL b holds iff (h,k) $UL_{REL}(x, y, x1, y1)$and(h1,k1)$LR_{REL}(x, y, x1, y1)$. [1]

The definition of box spatial composition REL result in there types of the relations: INCLUDE, INTERSECT and the spatial concatenations RIGHT and DOWN .

3 An Inclusion Hierarchy of Visual Language Classes

The figure 3.1 shows the inclusion hierarchy of the visual language classes which describes the generalization-specialization relation between them.

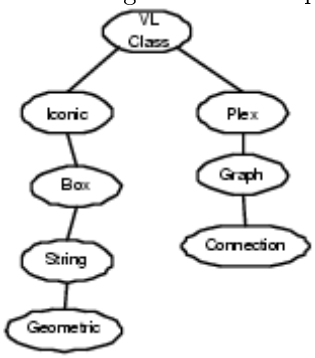

Figure 3.1 VL classes inclusion hierachy

4 Hybrid Classes of Visual Languages

The complex visual languages can not be modelled only by the basic visual classes. Integrating features from one or more basic visual classes can extend their description capabliities. Hybrid modeling is very important concept used to design many practical visual language. Box-and-graph visual languages like Statecharts, UML diagrams are an example of hybrid modeling. For example, in figure 4.1, the graphical object 1 of statecharts sentence has the systactic attributes not only from the class graph but also from the class box. The former is used to connect the states, and the latter is used to express the spatial INCLUDE relation which denotes the containment between states and superstates in statechart.

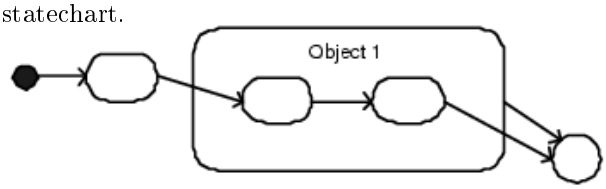

figure 4.1 A statechart sentence

Annotate relation is introduced when designing the hierachical languages, i.e, visual language whose graphical objects can be annotated by visual sentences from prossibly different visual languages. This relation meke the nesting of the language possible.

5 The VLCC System

The Visual language Compiler-Compiler(VLCC) system is a tool used to automatically generate the visual environment implementing visual language. The VLCC has two main components that are the *Grammar Editor* (GE) and the *Visual Programming Environment Generator* (VPEG) (see figure 5.1).

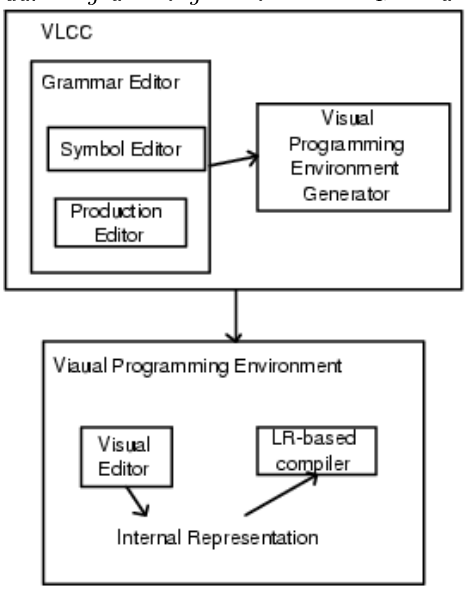

figure 5.1 The VLCC architecture

The designer uses the symbol editor of the GE specifies the visual aspect (the graphics) and logical feature (the syntactic attributes) of the visual language. In additon, GE allows the designer to define the syntax and the semantics of the visual language through a text editor or graphically through a visual production editor. With the specification, VPEG generates an integrated visual programming environment that consist of the graphical editor for composing the visual sentences and a compiler for interpreting and translating them. Each new environment generated by VLCC has three levels:

1. the graphical interface
2. the visual language class implementation
3. the language compiler and tokens

Each level is inherited or uses classes from the higher level. The first level implements the interface of the graphical editor. The second level implements a given visual language class mentioned above, that is, it's stucture of the visual sentence including the syntactic attributes and the feasible relations. The third level is specific to each visual language and implements the compiler and tokens for that language. The first two levels are pre-defined and implemented in libraries, and the third level is created by a visual language designer. The

designer defines token's graphcal features and the relation between its syntactic attributes and its image through *Grammar Editor*.

Part II
Constructing the User Interface with Statecharts

6 The typical approach to user interface development: bottom-up

The user interface application is event drive. A user generates an event to an object and a sequence of actions is executed in response to that event. The application waits until next event generated. Not only the event but also the other factors determine actions such as the values of the other interface objects, the events that have been previously supplied by the user and so on. It is clear that the behaviour of interface objects must be co-ordinated with the behaviours of other objects.

The event-action paradigm results in the bottom-up approach used to construct the user interface. The developer must understand all the behaviours of the application in detail, and then put them together. So there is no abstract view of the application.

The event handler is responsible for one part of behaviours of the application. In an application there usually exist more than one event handlers. They co-ordinate with each other through the use of global varialbes. Thus, it's impossible for each event handler to be understood in isolation from the rest of code. For the same reason, it is very difficult to modify one part of the code without affecting the other part of the application.

The bottom-up approach to user interface development relies on the skill of the developer to identify all the possible ways in which a user can supply events to the application. It's difficult to get the appliction to work correctly.

Above all, Constucting user interface application based on the event handlers via global variables is not a good way.

7 The user interface-control-model architecture

The user interface-control-model(UCM) architecture provides the user interface with solid foundation on which the code can be written, tested and maintained.

The UCM will centralize the control of the user interface in a small number of control objects. Each control object is resposible for co-ordinating the behaviour of a group of related user interface objects. see figure 7.1.

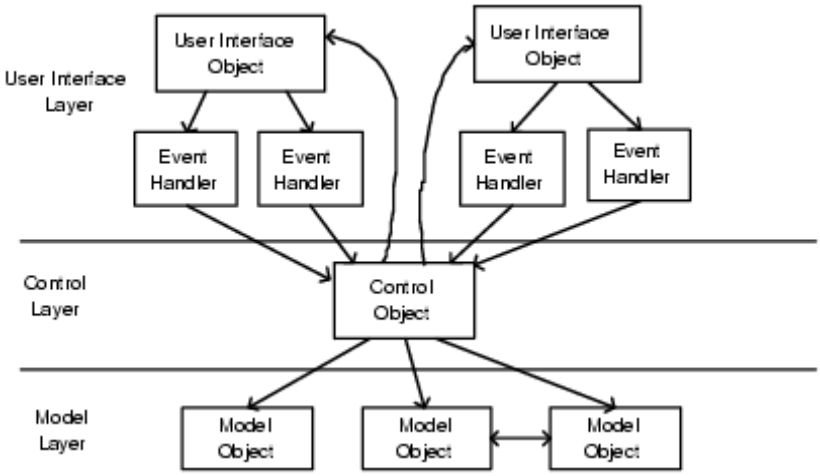

Figure 7.1 The UCM architecture for user interface software.

The event handler associated with the user interface objects are simply used to forward events supplied by the user to the appropriate control object, that is, the event handler simply calls the corresponding method belong to the control object. The control object maintains the state of the user interface as a whole, and is responsible for sending the message to the user interface objects when a user event occurs.

The events that a user suppliers will cause the application to move from one state to another state and teh state defines the set of possible events that a user can supply. This is the event-state-action paradigm. It is the nature of all direct manipulation user interfaces.

The UCM architecture is good in principle, but creating control objects that are easy to understand requires a powerful design notation.

8 User interface software design with statechart

The statechart notation (Harel, 1987) avoids duplicates states and duplicate event arrows, and overcomes the problem of large number of states and event arrows by using state transition diagram as the design notation. The state in a statechart have a hierachical structure so that it is possible to consider the systme at different level.

8.1 The statechart notation

8.1.1 Simple transitions

A statechart is simply a network of states and events. Figure 8.1 displays a simple transition from one state to another. It means if event 1 occurs in state A and condition a is true at the time, then make the transition to state B.

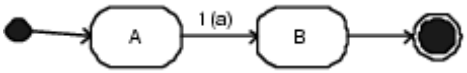

Figure 8.1 A simple transition

Usually there is a number of actions associated with each transible. It's impossible to fit these action into the diagram so that for each statechart. An event-action table like table 11.3 is used to provide the precise detail of every state transition in the diagram.

8.1.2 States with depth: refinement and cluster

In a statechart, the states are usually arranged in a hierachy, that is, states have depth. In figure 8.2, State A and B provide a high-level abstract representation of the system, and the state A contains the the lower level of detail.

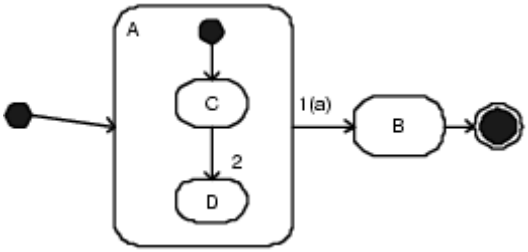

Figure 8.2 state A with depth

Depth allows the a statechart diagram to be viewed at different levels of abstraction. The high-level states can be identified first and the lower-level details can be delayed until a later time. This is one purpose of depth.

The other purpose served by depth is clustering. States can also be used to cluster groups of state to reduce the number of event arrows in a statechart. The statechart in figure 8.2 is the result of clustering state A and B in figure 8.3.

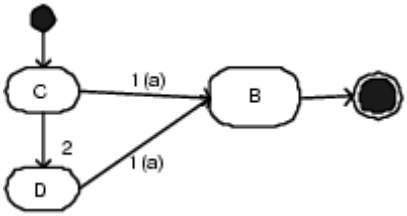

Figure 8.3 statechart without depth

8.1.3 Default start states

A start state is identified by a short arrow terminated with a solid circle. For example, in figure 8.2, the default start state at the highest level is state A. Within state A, the default starting state is C. Thus, the initial state will be state C within state A.

8.1.4 End State

Harel does not define an end state in his notation. But the Unified Modelling Language includes statecharts and the symbol used for the end state in that notation is a solid circle surrounded by an empty circle(see figure 8.3)

8.1.5 The history mechanism

The history mechanismm provides a way of entering a group of states based on the systems history in that group. That is, the state entered is the most recently visited state in that group. In figure 8.4 when event 1 occurs and state A is entered the history mechanism is used to determine the next state, that is, either enter the most recently visited state in the group (B1, B2, B3) or enter state B1 if this is the first visit to the group.

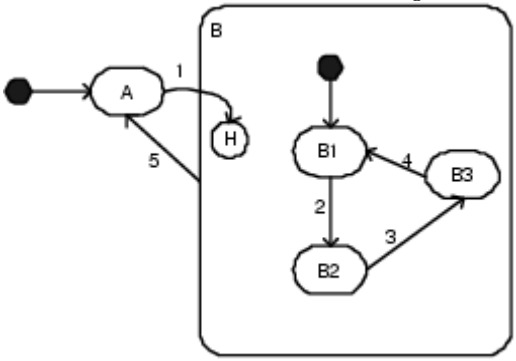

Figure 8.4 Shallow history state

The history H of a system is only applied to the level in the hierarchy in which it appears. To apply the history mechanism at a lower level, it is necessary to use a history symbol in the lower levels. H represents the shallow history state.

An asterisk can be attached to the history symbol H to indicate that the history of the system should be applied all the way down to the lowest level in the state hierachy. In figure 8.5 suppose state C2 was the most recently entered state in state C. When event 1 occurs, state C will be entered and then state C2.

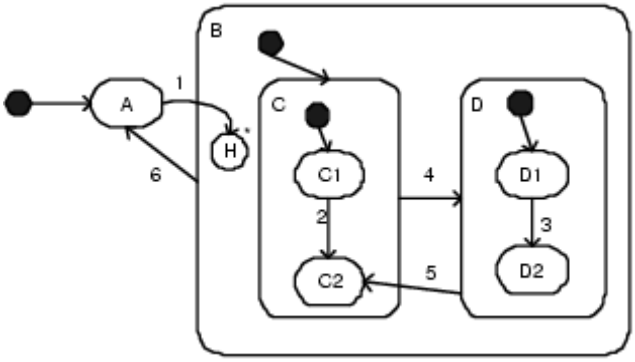

Figure 8.5 Deep history state

8.1.6 Concurrency

Figure 8.6 shows that state C has two components A and B. Entering state C cause (A1,B1) to be entered. Event 1 will cause the transition state A1 to A2 and state B1 to B2 to occur simultaneously and so the new state will become (A2, B2). The transition 2 has the condition 'in B2', which indicate that the transition will only occur if component B is in state B2. This means that states A and B are not completely independent of each other; State A must know something about the inner state of B.

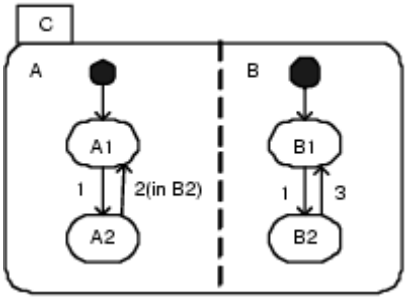

Figure 8.6 Concurrency in statechart

Figure 8.7 to 8.13 illustrates some several different ways of entering and exiting concurrent states.

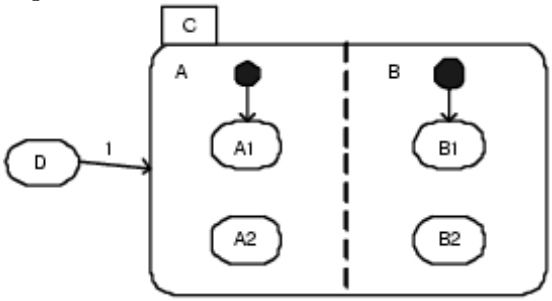

Figure 8.7 Event 1 causes entrance to the default states (A1, B1)

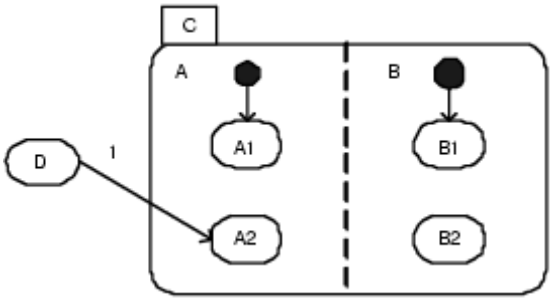

Figure 8.8 Event 1 cause entrance to state A2 and default state B1

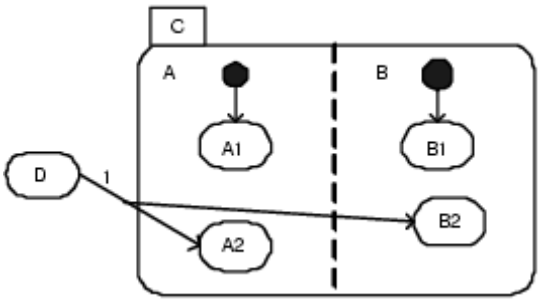

Figure 8.9 Event 1 causes entrance the state (A2, B2)

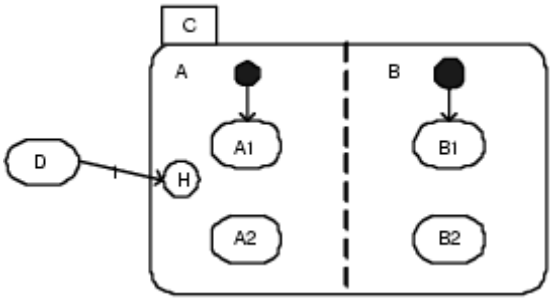

Figure 8.10 Event 1 cause entrance to the most recently visited state in A (or state A1 if it is in state A for the first time) and the default state B1

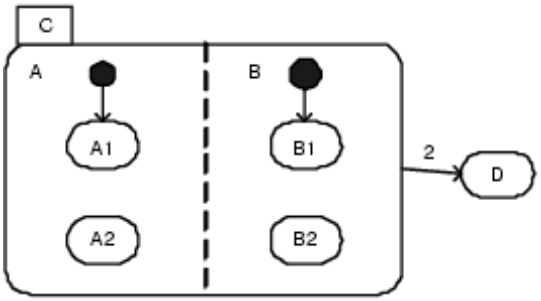

Figure 8.11 when event 2 occurs, state D will be entered. Event 2 is an unconditional exit from the concurrent state C and is therefore the most general form of exit.

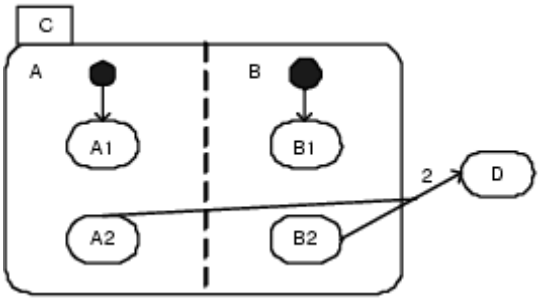

Figure 8.12 State D will be entered when the concurrent state is in (A2, B2) and event 2 occurs.

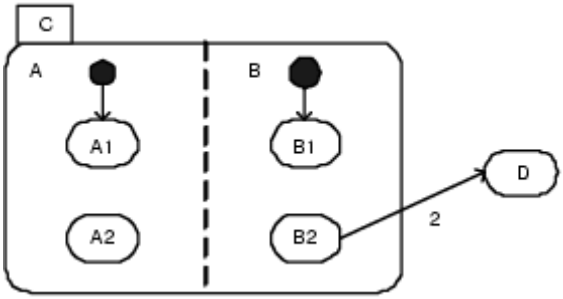

Figure 8.13 state D will be entered when B component of the concurrent state C is in state B2 and the A component is in any state.

8.1.7 Delays and time-outs

A delay mechanism can be applied on any state within a statechart. A delay will prevent a user being able to perform events in that state for a specific period of time after entering that state. For instance, in Figure 8.16, on entry to state A clicking the 'OK' button will have no effect until 3 seconds have elapsed. After the statechart has been in state A for 3 seconds, the event 'clickOK' will cause a transition to state B.

Figure 8.16 State with deferrable event

A time-out mechanism can also be imposed on any state. In figure 8.17, if the application remains in state A for 3 seconds and no event have occured, then the application will time-out and cause a transition to state C. If a user clicks button at any point in the 3 seconds then the timer will be reset.

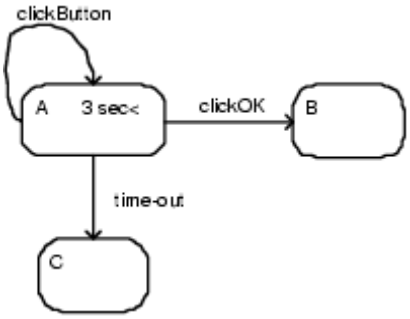

Figure 8.17 State with time-out event

8.1.8 Transient states

Transient states are different from normal states because all the transitions leaving such a state are simply conditions without an associated event. On entry of the state, the conditions on the event arrow are evaluated and the next state is determined and entered immediately.

8.1.9 Event priorities

The event leaving the same state with different condition would cause non-deterministic behaviour in the statechart because it is possible that more than one condition could be evaluate to true. To avoid this situation, relative priorities can be assigned to the events leaving a state. Event priorities are not part of Harel's notation.

8.1.10 Parameterized states

Parameterized state is the state that accepts parameters and can be used to replace several similar states within the statechart. Harel did not create the notation for parameterized state, but this is very useful concept when controlling modal alert messages in a user interface.

8.2 Heuristics of User interface design

The design heuristics presented in this section won't gurantee a good design. But it may help solve the problem you face during the design.

8.2.1 Identify the high-level statechart

The first step in the statechart design is to identify the high-level states. These states usually corresponds to screen that a user can navigate between. Some screen can invoke subwindow. If a subwindow can only be called from a particular screen, the state for the subwindow is likely to be the substate of the state for the screen. In the other case, if a subwindow can be called from more than on screen, The state that coresponds to the subwindow would be at the highest level in the hierachy.

8.2.2 Screen rule

After the identification of the high-level state, the next steps includes

1. **Identify all the screen items in or related with the screen** Write down all screen items such as buttons, menu item, text item, scrolling list and so on. Also consider the keyboard shortcut.

2. **Entry and exit rules** Need to find out the events that cause a screen to be entered or exited, events that cause the application to be closed and different conditions that can affect the behaviour.

3. **Identify any modes** A mode is a state that the software can be enter. The same user event may have differnt effect because of the different mode that the user interface in. In this case, these modes are obvious state for states. A screen may have read-only mode or read-write mode. A screen may have different behaviour with different data displayed in it.

4. **Identify the screen items with varying behaviour** Identify not only the change in behaviour, but what events and conditions cause the change in behaviour.

5. **Identify the screen with constant behaviour**

8.2.3 Seperate unrelated behaviour

Seperating the independent behaviour in the statechart is very important for the user to understand and modify the part of design without knowing about the ohter parts of design. The designer should use depth to cluster the related states and model independent behaviour in different state. When designing the screen, try to model its independent behaviour with the concurrent parts.

8.2.4 Keep dependency out of independent concurrent parts

In figure 8.6, when in state A2, the event 2 occurs only if the other concurent part is in state B2. The condition (in B2) causes one concurent part to be dependent on the other one. Such dependencis make the testing and long-term maintainability more difficult. If the dependencis between concurrent parts are unavoidable, then consider merging the two concurrent parts.

8.2.5 Synchronize concurrent parts with simultaneous events

A single user event sometimes causes the simultaneous state transitions in concurrent parts. Usually actions will only be associated with one of the event arrows-the primary event arrow. The ohter event arrow should be seen as a way of synchronize the concurrent parts.

8.2.6 Be wary of action on states

Associating the action with state cause the software harder to be maintained. For example, add a new event to the state with actions, but these actions executed when in the state aren't desired by the new event, then some significant changes have to be made to the state. On the contrary, if the actions associated with the state have to change, then the change must be appropriate to all the transitions entering to this state. Note all actions should associate with events.

8.2.7 Avoid event arrows that transcend many levels in a state hierachy

Try to avoid event arrows crossing too many levels in the state hierarchy. When using depth to refine a state, try that each level in the hierachy doesn't have the event from the higher level, that is, make the event arrow move level by level. Thus, the statechart is easier to be understood. One exception is that the arrow can transcend the cluster state.

8.2.8 Naming state

Every state in a statechart should have a name to indicate its purpose. For parameterized state, the parameter enclosed with the brace must followed the state name such as 'Alert(message)'. Since Transient state rarely represent anything meaningful, it is usually labelled with 'Transient'. Those states used to cluster other states can be unnamed. But those high-level states should be given a meaningful name because they are abstract representation of part of system

Part III
Case Study: DChart Interface

The application DChart Editor provides a visual environment to create statechart diagram. It is a reactive system. The process of creating statechart diagram is actually the interaction between the user and the application. The behaviours of the application can be modelled by statecharts. This case study is about how to constructe the DChart Editor with statecharts. Before figuring out how the system responds to the user action, we must be sure about the main user tasks.

9 The main tasks of the users

The main tasks of the users will be as follows:

To draw the state chart diagram:

1. To add the basic states on the canvas, and connect the states to make the transition between them.
2. To add the composite state on the canvas and drop substate.
3. To add substate to the composite state.
4. To add orthogonal part to the composite state.
5. To add the histroy state to the composite state.
6. To create transitions.
7. To undo the operation

To operate on canvas:

1. To save the created diagram.
2. To open the exsiting diagram.

3. To clear the canvas.
4. To exit the system.

i**To manipulate with the exsiting diagram:**

1. To select part of the diagram.
2. To deselect part of the diagram.
3. To delete part of the diagram.
4. To cut part of the diagram.
5. To copy part of the diagram.
6. To paste part of the diagram.
7. To scale the selected part of the diagram.
8. To move the selected part of the diagram.
9. To edit the state.
10. To edit the transition.

10 The screen of the user interface

This application has only one screen. This screen contains one toolbar which consists of those buttons such as basic state, composite state, transition and pseudostate such as the initial state, shallow history vetex, deep history vertex, fork vertex and join vertex. It also contains one menu bar which is made up with the menu file and edit**.**

The file menu is made up with menu items new, open, save, clear and exit. The edit menu contains menu items cut, copy, paste, delete and edit.

The screen is entered when the application is started and exited when the exit menu item is clicked. It's also exited when the window is closed.

11 Design

We use the user interface-control-model(UCM) to construct our reactive system. There are three layers in UCM architecture, which are user interface layer, control layer and model layer. The statechart diagram is used to model the behaviour of the control object(s) in the control layer.

11.1 Interface layer

The interface layer is implemented by class DChartEditorGUI. The class DChartEditorGUI constructs the GUI part of the application using the class MenuBar, ButtonBar and DrawArea. And it not only responds to the user action on the GUI to generate correponding event, but also keeps track of the graphical items defined by the class GraphItem on the drawing area, and provides the controller layer with the callback functions to operate on them. Figure 11.1 shows the relationships between the classe DChartEditorGUI and the other classes. Figure 11.2 is the UML symbol for class DChartEditorGUI.

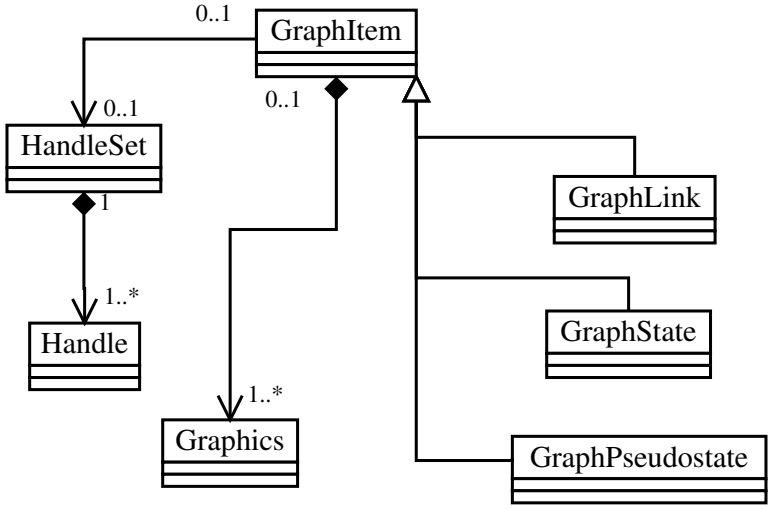

Figure 11.1 The UML association Diagram for interface layer

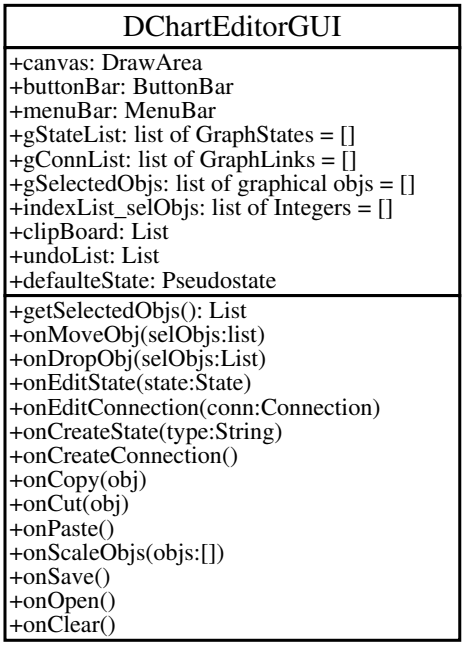

Figure 11.2 The UML representation for the class DChartEditorGUI

All the graphic objects, no matter whether they are part of the GUI or they are dynamically added to the drawing area of the GUI, are made up with one or more instances from the class Rectangle, Line, Oval, Text and so on. All these basic shape classes share the same super class Graphics (See figure 11.3).

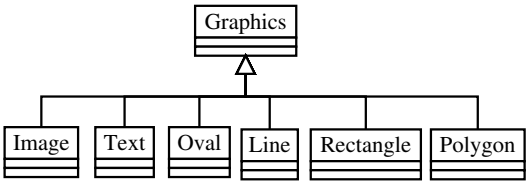

Figure 11.3 The UML class diagram for the class Graphics and its subclasses

The graphical item drawn by the user could be the instance of the class GraphLink, GraphState or GraphPseudostate. The class GraphState defines the visual representation of state. The class GraphPseudostate defines the visual objects for pseudostates including default state, shallow history state, deep history state, fork vertex and join vertex. The class GraphLink is for the transition. These classes are inherited from abstract class GraphItem. When the graph item is selected, it is to be surrounded by one HandleSet instance that consists of six instances of the Handle class. The existance of HandleSet helps the user to perform some operation like move, scale and etc on the graph item. The UML association diagram for all kinds of graph item classes is shown in figure 11.3. Figure 11.4 is the UML representation for class GraphItem, GraphLink and GraphState.

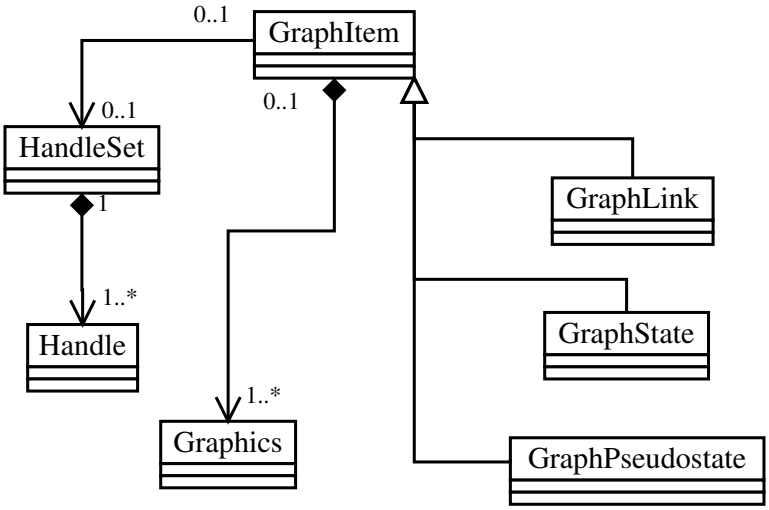

Figure 11.3 The UML association diagram for graphic item

Note the attribute itemObj of class GraphItem is used to store its corresponding object in the model layer.

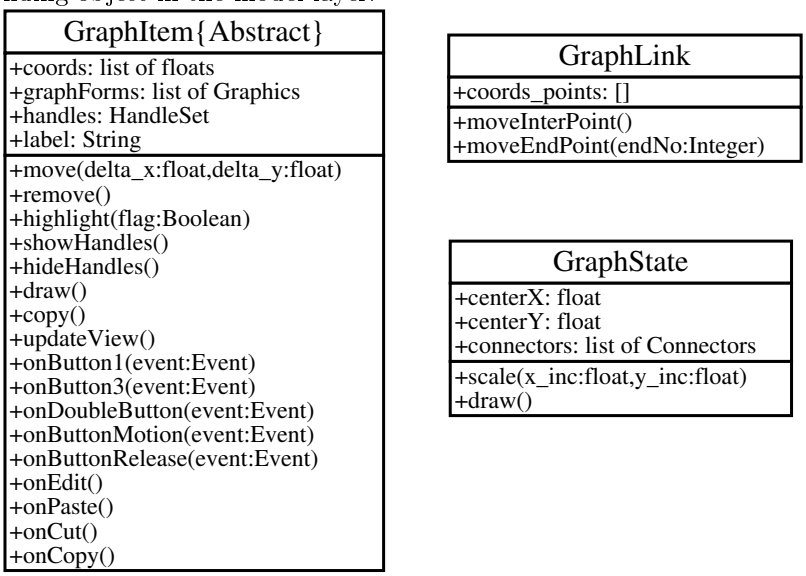

Figure 11.4 The UML symbols for the class GraphItem, GraphLink and GraphState

Note that the attribute coords of GraphItem stores the coordiates of the item's approximate bounding box. Beside the attribute coords, the GraphLink also has its own coordiate list coords_point that is used to record the coordiates of its intermediate points.

The class GraphLink has three different move methods. One is the method move() from its parent class, which moves the link anywhere. The method moveEndPoint() only moves the end point of the link, if the end point overlaps

with some connector of the graphic item, the system is to connect them together. The method moveInterPoint() is used to drag some part of the link using one of its intermediate point.

11.2 Model layer

The Model layer consists of model objects such as State, Transition and so on. The relationship of there objects is illustrated by the UML association diagram in Figure 11.5.

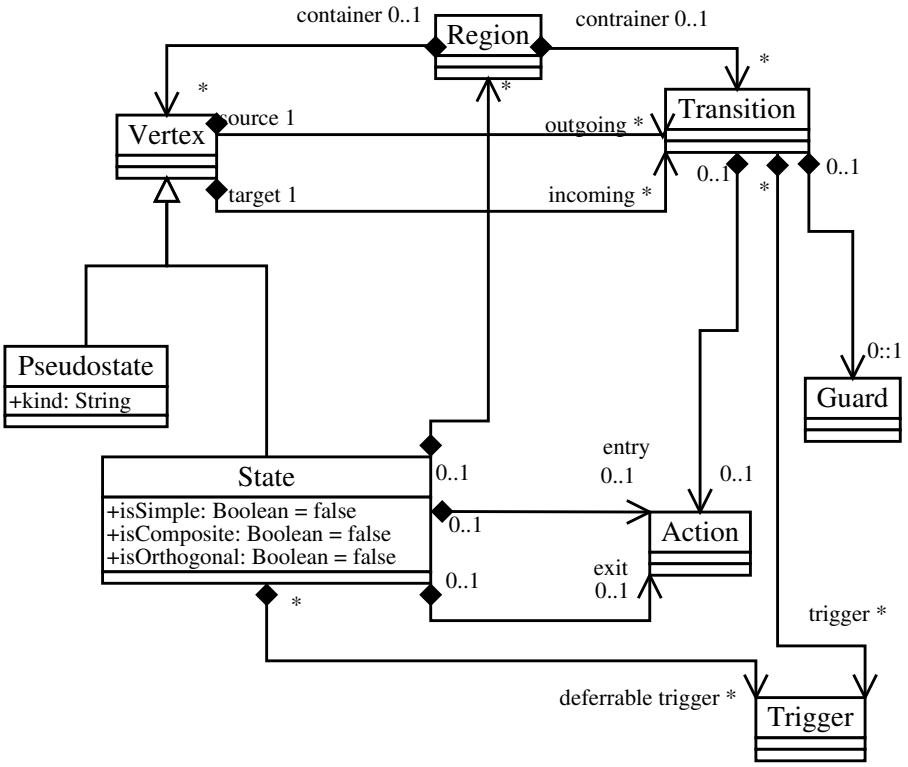

Figure 11.5 The association diagram: Model layer

The class DChartEditor is used to control the life cycle of the model objects, keep track of information of the diagram being created and modified in the GUI by the user. Like the class DChartEditorGUI, it also provide some functions to manipulate the other model objects. From its UML symbol in figure 11.6, we can understand its purpose more clearly.

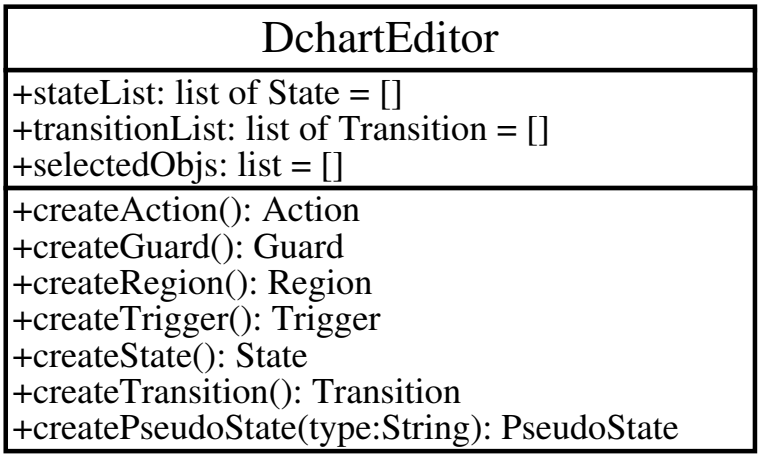

Figure 11.6 The UML symbol for the class DChartEditor

The vertex can be pseudostate and state. A state with isSimple=true is said to be a simple state. A simple state doesn't have any regions. A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint vertices and a set of transitions. The composite state with two or more regions is referred as the orthogonal composite state. Each state is associated with entry action and exit action. It may specify a set of deferrable event in that state. Each region of the composite state may or may not have the default state and/or history state. Each vertex may have Region(s) and/or State(s) as parents.

Each transition has one source vertex and target vertex. It is associated with Trigger, Guard and Action. The trigger is used to fire the transition. A guard is a contraint that provides a fine-grained control over the firing of the transition. The Action specifies an optional activity to be performed when the transition is fired.

In this application, we only use five kinds of pseudostate, that is, the initial state, shallow history state, deep history state, fork vertex and join vertex. The initial psudostate is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a region. The initial transition may have an action. A composite state can have at most one shallow/deep history vertex, and at most one transition may originate from the shallow/deep history symbl to the default shallow/deep history state. Join vertex is used to merge several transitions with the same target state. Fork vertex serve to merge several transitions with the same source state. The existence of fork vertex and join vertex makes the hyperedge possible in the DChart Editor.

The UML class representation for these model object is shown in Figure 11.7.

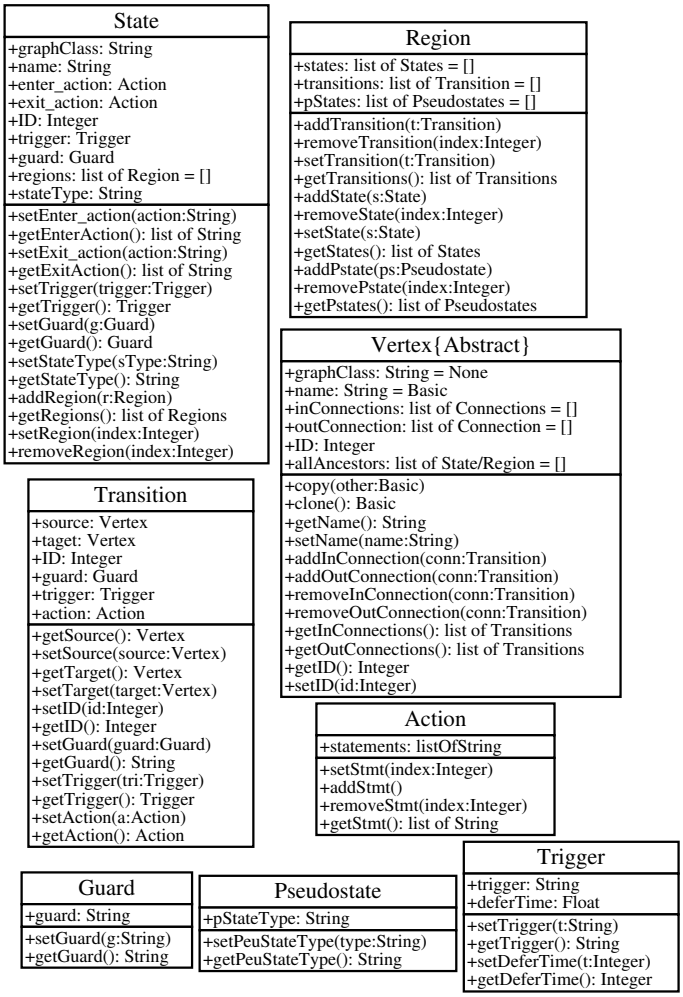

Figure 11.7 The UML symbol for model objects

11.3 Control layer

The control layer is the core of the application. It accepts the event passed from the user interface layer, i.e GUI, determines which transition the system should make and which output action the system will take, and then brings the system to the next state.

11.3.1 User Actions

The table 11.1 summaries the basic mouse binding. The application may generate different events to respond to the same mouse event. For example, based on the location, single clicking mouse button 1 is to create the different event that could be clickOnMenu, clickOnMenuItem, clickOnButton, clickOnCanvas

and clickOnObj. The system is to take the appropriate actions to respond to these events. Some event may correspond to more thant one actions. Which action will be taken depends on which state the system is in. Table 11.2 lists the basic key binding for the interface

11.3.2 Model Interactivity with Statechart

The control layer is responsible for invoking the appropriate actions in response to an event, maintaining the current state of the application and ensuring the user interface objects work together as a whole. Describing the control layer in terms of states and events nutually results in the use of statechart diagram.

In the application the interface objects don't share information with each other, that is, one interface object can't access the data that the other interface objects are using or are to use. It's the duty of the control layer to manage the global information. Each interface object is independent of the others, and it has its own behaviour. Some of them may respond to the same event, but they are act differently.

Figure 11.8 shows the top level of design. There are only three states. The application generates the start event, the system enters the state DChartEditor from the initial state. Only when the exit event is triggered, the system will be exited. Since each interface object acts independently, the DChartEditor state is defined as orthoganal composite state. Each orthoganal part represents the behaviour of one interface object.

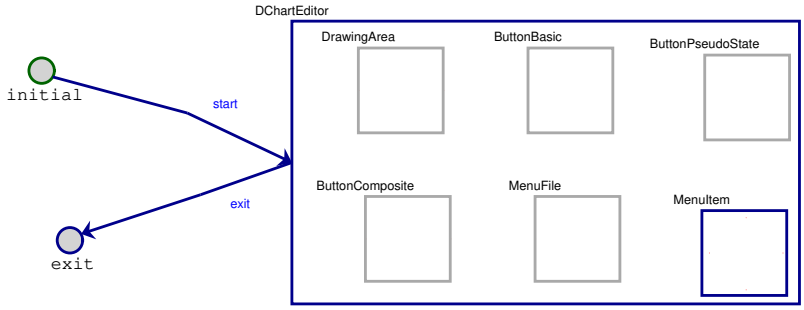

Figure 11.8 Statechart diagram: top level of the application

Let's look at each orthoganal part of state DChartEditor in detail.

(1) DrawingArea State

The drawing area is either in the empty state or in the not-empty state. By default, the system is in the state empty. When the event openFile occurs, the system enters the substate notEmpty. The event saveFile and openFile won't lead to the state change, on the contrary the event clear will bring the system back to the default state empty. In addition, when the system is in state empty, if there is no event triggered, the system will automatically goes into the notEmpty state. Figure 11.9 shows the detail of the state DrawingArea.

Mouse Event	GUI Generated Event	Action(s)
ButtonPress-1	clickOnCanvas	insert object start multiple selection paste hide the submenu
	clickOnMenu	get menu type
	clickOnMenuItem	generate corrsponding event
	clickOnButton	get button type
	clickOnObj	select object
buttonMotion-1	moveObj	move the selected object(s) move one end of link
	scaleObj	scale the selected object(s)
buttonRelease-1	dropObj	drop the object finish scaling connect one end of link to object
	finiMultiSelection	finish multiple selection
Double-buttonPress-1	editObj	popup the edit dialog
buttonPress-3	popuMenu	show context menu
any motion	moveOver	move over the menu item
	moveOff	move off the menu item

Table 11.1 Basic mouse event binding

Key Event	GUI Generated Event	Action
key-Delete	deleteObj	delete object(s)
key-Z	undo	undo operation
key-X	cut	cut object(s)
key-C	copy	copy object(s)
key-V	paste	paste object(s)
shift-Button-1	multipleSelection	select multiple objects

Table 11.2 Basic Key event binding

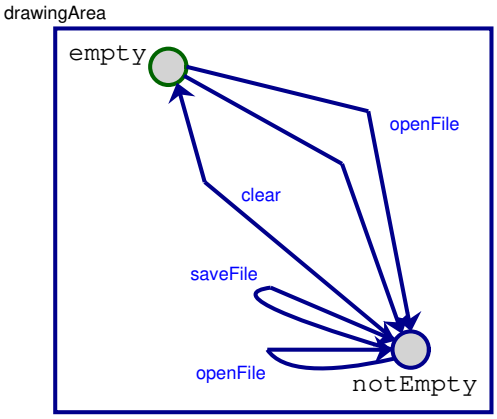

Figure 11.9 Statechart diagram: state DrawingArea

When the system enters the state notEmpty, it automatically goes into the default substate. Figure 11.10 illustrates the interactivity between the user and the application. Table 11.3 is an event-action table associated with the state-chart diagram in Figure 11.10. This table provides the precise detail of every state transition in the diagram. Note that in the table gui represent the instance of DChartEditorGUI, and editor represent the instance of DChartEditor.

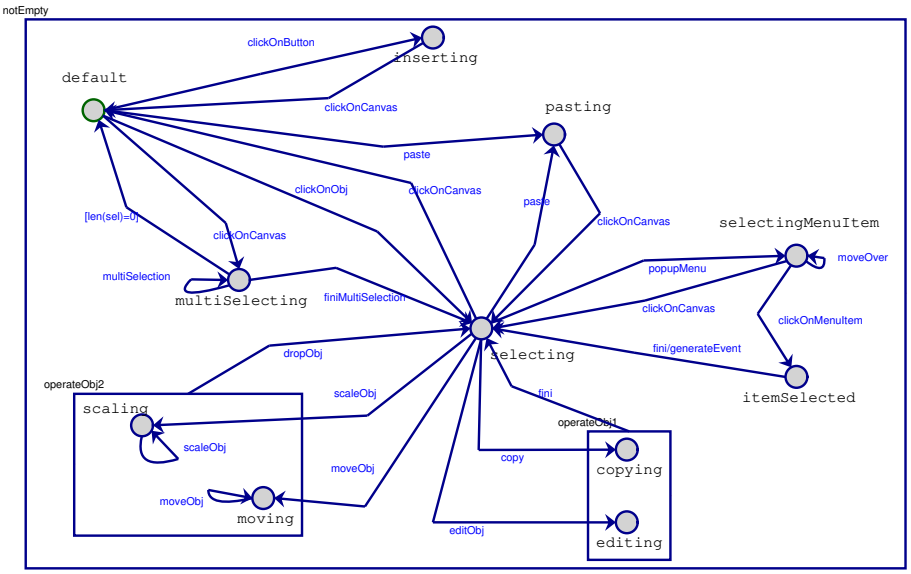

Figure 11.10 Statechart Diagram: substate not-empty of state DrawingArea

Note: The event fini isn't triggered by the user. It's generated only when

the system exits the some state. It can be seen as the exit action of that state.

Table 11.3 Event-Action table associated with the statechart in figure 9.3

Current state	Event & Condition	Actions	Next State
default	clickOnButton	gui decides the type of the clicked button	inserting
inserting	clickOnCanvas	gui draws the graphic object with given type; add it to the graphic object list. If state, gui determines whether or not it is substate based on the location	default
		editor creates new instance of state/Transition; add it to state/Transition list; if it's substate , editor updates its parent.	
default	clickOnObj	gui decides which graphic object is selected, highlight it	selecting
selecting	clickOnCanvas	lowlight the selected graphic object	default
selecting	deleteObj	gui removes the selected graphic object from the list	default
		editor removes the selected state/Transition from the list	
selecting	editObj [1 obj selected]	gui shows the edit dialog for the selected graphic object	editing
		editor assigns or updates the values of the selected state/Transition	
selecting	copyObj	editor clones the selected state/Transition; add it to state/Transition list	copying
		gui draws the clone and add its graphic form into graphic object list	
selecting	moveObj	If the selected graphic object is state, gui keeps track of the location of the object and its in_connections. If it's the end or the intermediate point of the link, just keep track of the location; update the view	moving
selecting	scaleObj	gui keeps track of the size of obj; and update the view	scaling

operateOnObj	dropObj	gui determines what is selected, state, link, the end of the link, intermediate point of the link; gui judges if the drop location is valid, drops the graphic object; if it's state, determines whether or not it is inside the other state. If it's end of the link, determinses whetehr or not the end of the link overlaps with one connector of the state	selecting
		If the selected graphic object is state and dropped inside the other state, editor updates its and its parent's information; If the selected graphic object is one end of the link and it overlaps with one connector of the state, then add the corresponding Transition object into the in_connection list of the state.	
selecting	deleteObj	gui removes the selected object(s) from its list. If there is only one selected object and it has other objects inside, then remove them too; update the view.	deleting
		editor removes the state/Transition from its list. If it's composite state, remove all its descendant too.	
default	multipleSel	gui records the list of selected object(s)	multiSelecting
multiSelecting	finiMultiSelection	gui updates the list of selected object(s)	selecting
default	paste	gui draws the objects in the clipboard list	pasiting
		editor creates the corresponding State(S)/Transition(s)	
selecting	paste	as above	default
selecting	popupMenu [1 obj selected]	gui shows the context menu	selectingMenuItem
selectingMenuItem	clickOnMenuItem	gui gets the selected menu item; generates the corresponding event	

(2) ButtonBasic State

The system is either in the normal substate or the pressed substate. By default, the system is in the substate normal and the button shows its normal view. Only after it's clicked, it is displayed in its pressed view. When the user clicks on the drawing area to add the basic state, the event clickOnCanvas brings the button back to its normal visual representation. Similarly when the user clicks on the other button or menu item, it also restores its normal view. The other button acts in the same way as the basic button.

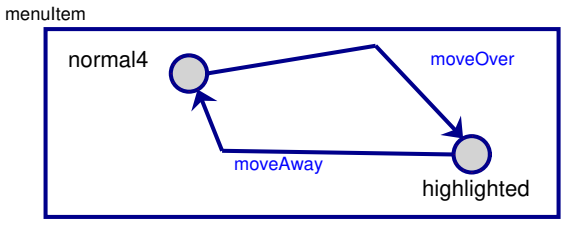

Figure 11.11 Statechart Diagram: ButtonBasic state

The other button in the GUI acts in the similar way.

(3) MenuFile state

Not like the button, the system changes the menu's appearance not only becuse of it being pressed. The system is to highligted it when the cursor moves inside it. The system is to pop up its submenu when it's pressed. The following figure shows the lower level of state MenuFile.

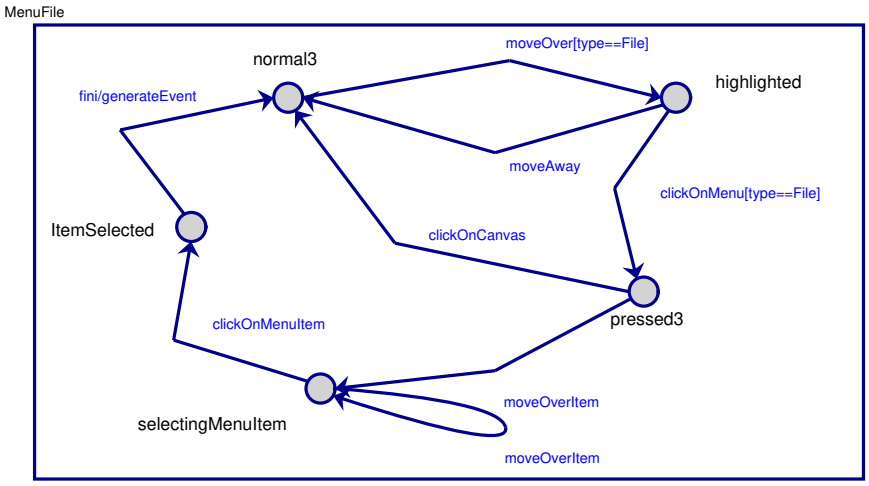

Figure 11.12 Statechart Diagram: MenuFile state

Note: the default substate is normal3. The event fini isn't triggered by the user. It's generated when the system exits the state itemSelected.

The following event-action table describes each transition of statechart in figure 11.12 in detail.

current state	evnet & condition	action	next state
normal	moveOver	gui highlights the menu file	highlighted
highlighted	moveOff	gui lowslights the menu file	normal
highlighted	clickOnMeun	gui show its pressed view; pop up its sub-menu	pressed
pressed	clickOnCanvas clickOnButtons clickOnMenu[other]	gui show the menu's normal view; hide its submenu	normal
pressed	moveOver[menu item]	gui gets the current menu item;	selectingMenuItem
selectingMenuItem	clickOnMenuItem	gui gets the selected menu item	itemSelected
itemSelected	fini	gui hides the submenu; show menu's normal view; generates corresponding event	normal

Table 11.4 Event-Action table associate with the statechart in figure 11.11

(3) MenuItem State

Figure 11.13 shows the statechart of the lower level of the state menuItem. When the system is in the state MenuItem, it is either in substate normal or in the substate highlighted. When the cursor move over menu item, gui highlights it and the system is in substate highlighted; otherwise if the cursor moves over the other menu item, that is, leaves it, then it is to be lowlighted and the system goes back to the substate normal.

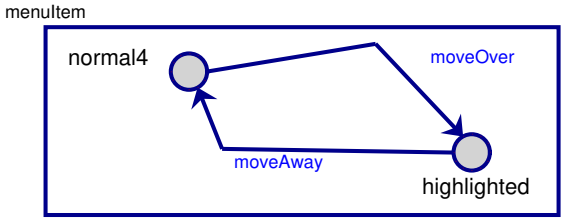

Figure 11.13 Statechart diagram: MenuItem state

11.4 Summary

The statechart provides a precise specification of the behaviour of the user interface, which is more concise and less ambiguous than any natural language equivalent. The use of concept of the depth and the concurrency makes it easy to divide the DChart Editor application into simple parts. We can enrich the behaviours of each part or add more interface component into the whole application easily.

Reference

1. G.Costagliola, A. Delucia, etc. *A Classification Framework to Support the Design of Visual Languages.* Journal of Visual Languages and Computing (2002) 13, 573-600

2. J. Feder(1971) Plex language. Information Science 3, 225-241

3. David H. and Michal P. *Modeling Reactive Systems with Statecharts.* McGraw-Hill

4. Ian Horrocks(1999). *Constructing the User Interface with Statecharts.* Addison Wesley

5. *UML2.0 Superstructure Specification*, 455-510, http://www.omg.org/docs/ptc/03-08-02.pdf