User Interface System Based on Active Objects

Sungwoon Choi
Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

91-60-5

User Interface System Based on Active Objects

Sungwoon Choi and Toshimi Minoura
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
choi@cs.orst.edu, minoura@cs.orst.edu

Abstract

An *active-object user interface system* (AOUIS) is a user interface system implemented as an *active object system* (AOS). An AOS is a *transition-based* object-oriented system suitable for the design of various concurrent systems. In an AOUIS, *user interface objects*, which are sometimes called "widgets", are represented as *active user interface objects* (AUIOs). The behavior of an AUIO is defined by the transition rules, the equational assignment statements, and the event routines provided in its class definition. Furthermore, an AUIO can be constructed from its component AUIOs through structural composition as if it were a hardware object. Thus, AUIOs are better encapsulated and provide more flexible communication protocols than ordinary user interface objects. In addition, *declarative* descriptions of *multiple* views can be provided for each AUIO.

Key Words and Phrases: user interface management system, active object system, production system, structural composition, software IC, subject/view.

1 Introduction

Implementation of a sophisticated graphical user interface is still a laborious endeavor although object-oriented programming [GOLD80, MEYE88], with its higher levels of modularization and code sharing, has made this process significantly easier. The benefits of object-oriented user interface systems are well documented in [LIPK82, SIBE86, BART87, LINT88, KNOL89]. In this paper, we show that further improvement can be achieved through the use of an active object system as the framework of a user interface system.

The idea of active objects originated from the first object-oriented language SIMULA [BIRT73], where active objects are cooperating sequential processes that communicate with each other through procedure calls. Several active object systems [AGHA86, BLAC86, YONE87] have been designed since then by replacing procedure calls of SIMULA with message passing. Some AI systems allow us to create active objects by using active values [KUNZ84, KEHL84]. A KNOs object [TSIC87], which possesses an internal state, a set of operations, and a set of rules, is also active. One approach for implementing a user interface system from active objects is introduced in [COOT88].

Our *active object system* (AOS) uses *transition (production) rules*, *equational assignment statements*, and *event routines* for its behavior description. Transition rules (production rules) are *condition-action* pairs, and they have been known to be suitable for various concurrent systems that require flexible synchronization [DAVI76, ZISM78]. Equational assignment statements can maintain simple invariant relationships among object states. Event routines are activated by messages. Our AOS supports one-to-many message passing as well as ordinary many-to-one message passing.

We call transition rules, equational assignment statements, and event routines *transition statements*. The behavior of each active object is determined by the transition statements provided in the *class* definition of that object. One key feature of our AOS is that the transition statements provided for each active object can access the states of the other active objects known to it, realizing inter-object communication.

The major goal of the AOS approach is to construct a certain class of systems by the hierarchical composition of active objects, where software objects are constructed and modularized like hardware objects. This feature is very useful in constructing a user interface system because construction of a user interface is basically assembling component objects such as buttons, text fields, and indicators together.

Behavioral composition of objects is emphasized in some user interface systems [LOND85, MORI85, REIS86]. The parallel object-oriented language POOL-I addresses the composition of heterogeneous objects. Other related ideas are *contracts* [HELM90] and *collaborations graphs* [WIRF90].

In an *active-object user interface system* (AOUIS), user interface objects are implemented as active objects, which we call *active user interface objects* (AUIOs). The fact that the AOS approach is based on active objects with *structural* interfaces, whereas a conventional object-oriented user interface system is based on passive objects with procedural interfaces, makes an AOUIS easier to design, implement, and maintain than an ordinary object-oriented user interface system. Ellis claims that active objects provide a higher level of modularity than passive objects [ELLI89]. For example, an AOUIS rarely requires back pointers, which are frequently used by a user interface system written in a conventional OOP language.

In Section 2, we give an overview of an AOUIS by using a simple example. AOUIS features are further explained in Section 3, and implementation issues are discussed in Section 4. Section 5 concludes this paper.

2 Simple Examples

In this section, we introduce the AOUIS approach by using simple examples. An AOUIS program consists of a *main* program and *classes*. A class contains three parts: an *interface*, a set of *instance variables*, and a behavior description. Variables defined in the interface part are called *interface variables*. An interface variable is a special instance variable that contains a reference to another object. The main program is defined in the same way as a class but is an instance. As in C++ [STRO86], instance variables and functions can be *private*, *protected*, or *public*.

2.1 Labeled Push-Button

Fig. 1 shows a graphical view of a labeled push-button. The button is shown as a circle when it is not chosen, and it is shown as a filled circle when it is chosen.

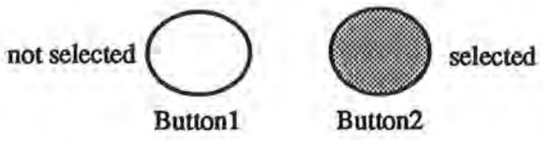

Figure 1: Labeled Push-Button.

We now show how the class for labeled push-buttons can be defined. Before class `PushButton` is defined, super-classes of `PushButton` must be defined.

```
class UIObject {
   UIObject *parent;
   Position position;
   Region region;
   Bool visible = true;

   /* event routine activated by a mouseDown() message */
   on mouseDown(Position location) from mouse {
      if (inside(location, region))
         say chosen();
   }
}
```

Figure 2: Class UIObject.

Class `UIObject` given in Fig. 2 is the root class for all the AUIOs. Every AUIO maintains a pointer `parent` pointing to its parent in the component hierarchy. The variable `position` indicates the location of a graphical object created as an instance of a subclass of `UIObject`. The variable `region` shows the region occupied by the graphical object. The variable `visible` indicates whether the graphical object should be visible or not. If a `UIObject` detects a `mouseDown()` event within the region of its graphical representation, it broadcasts message `chosen()`.

```
class Button: public UIObject {
   public:
      bool selected = false;
      char *label;
      Button(char *name) {label = name};
}
```

Figure 3: Class Button.

Class `Button` defined in Fig. 3 is a subclass of `UIObject`. It has two instance variables, `selected`, which indicates whether the button is *on* or *off*, and `label`, which stores the name of the button.

We now can define the class `PushButton` as a subclass of `Button`. Fig. 4 shows AOUIS description of `PushButton`.

```
class PushButton: public Button {
   public:
      PushButton(char *label) : (label) {};
      int radius = 20;
      /* a circle */
      Circle circle(radius) at (0, 0);
      /* draw a text on the bottom of the circle */
      Text text(label) at (0, -30);
      /* a filledCircle */
      FilledCircle fcircle(radius) at (0, 0);
      /* to be displayed when selected is true */
      always fcircle.visible = selected;

      /* event routine activated by the message chosen() from its superobject */
      on chosen() from this do
         selected = not selected;
 }
```

Figure 4: Class PushButton.

When each instance of `PushButton` detects message `chosen()` from itself (super-object), it changes its instance variable `selected`, which is defined in `Button`, from `true` to `false` or from `false` to `true`.

The graphical representation of an instance of `PushButton` is declaratively defined by instance variables `circle`, `text`, and `fCircle`. These graphical objects are instantiated when a `PushButton` is created. The graphical objects `circle` and `text` are displayed statically all the time. The graphical object

fcircle is displayed only while the labeled push-button is selected, as dictated by the equational assignment statement **always fcircle.visible = selected**, which always maintains **fcircle.visible** equal to **selected**.

2.2 Button Group

Radio buttons are similar to labeled push-buttons, but among a group of radio buttons, only one of them can be selected at any time. Fig. 5 shows a button group which has three radio buttons.

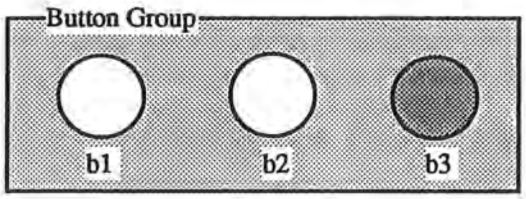

Figure 5: Button Group.

Fig. 6 shows the definition of class **RadioButton**, which will be used as a component class of the class for button groups.

```
class RadioButton: public Button {
   interface:
      int *i_selectedButton ;
   public:
      int myID;
      int radius = 20;
      RadioButton(int id, char *name) : (name) {myId = id, i_selectedButton = nil};
      Circle circle(radius) at (0, 0);
      Text text(label) at (0, -30);
      FilledCircle fcircle(radius) at (0, 0);
      /* draw a filledCircle whenever selected is true */
      always fcircle.visible = selected;

      /* event routine activated by the message chosen() from its superobject */
      on chosen() from this do {selected = true; *i_selectedButton = myID;};
      /* transition rule activated whenever myID is different from selectedButton */
      if (*i_selectedButton <> myID) selected = false;
}
```

Figure 6: Class RadioButton.

In the implementation of class `RadioButton`, each button is responsible for all of the operations of the button group. When each instance of `RadioButton` detects message `chosen()` from itself (superobject), it changes its instance variable `selected` to `true` and sets the variable referred to by the interface variable `i_selectedButton` to the value of its `myID`. As we see later, the variable referred to by `i_selectedButton` indicates the button selected within a button group. Then each of the other buttons sets its `selected` field to `false`, detecting its `myID` to be different from the value of the variable referred to by `i_selectedButton`. The graphical representation of a radio button is identical to that of a push button.

We now can define the class for button groups by using `RadioButton` as a component class. Class `ButtonGroupA` given in Fig. 7 is an AOUIS description of the class for button groups.

```
class ButtonGroupA {
   public:
      int selectedButton;
      RadioButton b1(1, "b1") at (0, 0) with i_selectedButton = &selectedButton;
      RadioButton b2(2, "b2") at (40, 0) with i_selectedButton = &selectedButton;
      RadioButton b3(3, "b3") at (80, 0) with i_selectedButton = &selectedButton;
}
```

Figure 7: Class ButtonGroupA.

The instance variable `selectedButton` stores the ID of the currently selected button. Three radio buttons, `b1`, `b2`, and `b3`, are defined as instances of `RadioButton`, and they are connected to `selectedButton` through interface variable `i_selectedButton`. This example clearly shows that we can declaratively construct an AOUIS class from its component AUIOs by specifying interactions among them.

3 AOUIS Features

In this section, we discuss further details of an AOUIS system. Our discussions centers around its key features, i.e., ***structural composition*** of AUIOs, ***inter-object communications*** among the structurally composed AUIOs, and ***multiple views.***

3.1 Composition of AUIO

We can compose a new AUIO from its component AUIOs by providing proper connections among them. *Interface variables*, which act like terminals of hardware components, are used for this purpose. Interface variables are pointers through which remote objects are accessed. The object bound to an interface variable may be dynamically changed. Dynamic bindings of objects to interface variables are needed to describe time-dependent structural relationships among active objects. As we discussed in [CHOI91], operations on dynamically bound objects cannot be handled as efficiently as those on statically bound objects. The latter can be compiled into efficient code.

AOUIS supports *component hierarchies* as well as *class hierarchies.* Although component hierarchies exist in ordinary object-oriented programming, we attach more importance to it. Fig. 8 shows the component hierarchy in the button group discussed in Section 2.

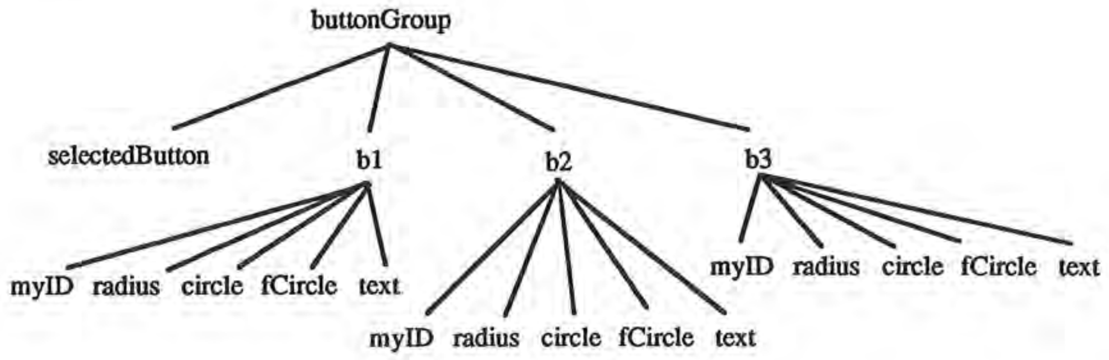

Figure 8: Component Hierarchy of the Button Group.

As in ordinary OOP, instance variables and transition statements are inherited from a superclass to its subclasses according to the class hierarchy. Furthermore, according to the *static* component hierarchy, public instance variables are visible to its components. The inheritance according to the class hierarchy precedes the inheritance according to the component hierarchy. This mechanism allows us to provide global variables at different levels.

3.2 Behavior Description

Objects in Smalltalk or C++ are passive in the sense that they only respond to the messages sent to them. On the other hand, the behaviors of AUIOs, can be specified by three kinds of transition statements: ***transition rules***, which are ***condition-action*** pairs, ***event routines***, which respond to messages, and **always** ***statements***, which are ***equational assignment statements.***

3.2.1 Transition Rule

Each transition rule is a *condition-action* pair, and its action part is executed when its condition part is satisfied. An execution of a transition rule should be atomic. The implementation details of the execution mechanism of transition rules are discussed in [CHOI91].

3.2.2 Event Routines

The activations of transition rules are, at least conceptually, state-driven. Active objects can communicate with each other by directly accessing the states of other objects rather than sending messages to them. Although this mechanism often eliminates the necessity of explicit message passing, some events are more efficiently handled by messages. An AOUIS supports event-driven activations of procedures. We consider *messages* as extended events that include some data as parameters. One important feature of our message passing mechanism is that it supports *one-to-many* message-passing as well as *many-to-one* message-passing.

We provide two constructs that support message passing among objects. The statement **say** $message(p_1, ..., p_n)$ [to *receiver*] is used to send a message (to the receiver specified as *receiver*). The receiver may or may not be specified. The statement **on** $message(p_1, ..., p_n)$ [from *sender*] is used to receive a message (from the sender specified as *sender*). The sender may or may not be specified. For a pair of **say** and **on** statements between which a message is passed, if the sender is not specified in the **say** statement, the receiver must be specified in the **on** statement, and vice versa. The receiver object of a **say** statement and the sender object of an **on** statement, if specified, should be visible to the objects issuing these statements.

The message-passing mechanism in ordinary OOP can be regarded as *many-to-one* message-passing, where each **say** statement specifies exactly one receiver of the message, while one **on** statement can receive messages from different senders.

If the receiver is not specified in a say statement, the message can be received by any objects to which the sender is visible. Hence, we implement *one-to-many* message passing. In the next example, we show how this mode of message passing can simplify the structure of an AOUIS class.

```
class ButtonGroupB {
   public:
      Button b1("b1");
      Button b2("b2");
      Button b3("b3");
      /* Event routine activated by chosen() from  b1 */
      on chosen() from b1 {
         b1.selected = true;
         b2.selected = false;
         b3.selected = false;
      };
      /* Event routine activated by chosen() from  b2 */
      on chosen() from b2 {
         b1.selected = false;
         b2.selected = true;
         b3.selected = false;
      };
      /* Event routine activated by chosen() from  b3 */
      on chosen() from b3 {
         b1.selected = false;
         b2.selected = false;
         b3.selected = true;
      };
}
```

Figure 9: Class ButtonGroupB

In the implementation of class `ButtonGroupB` shown in Fig. 9, `ButtonGroupB` takes care of all the operations required for a button group. The `chosen()` message generated by any `Button` is caught by its parent `ButtonGroupB` instance. We could program `ButtonGroupB` without introducing a specialized class such as `RadioButton`, because on statements can designate their sources of messages. In this way, we can define classes without proliferating class definitions.

3.2.3 Always Statements

A simple mechanism for describing a behavior of an AUIO is an equational assignment statement that maintains an invariant relationship among the states of AUIOs. `always` statements are used for this purpose.

An `always` statement can be implemented like a transition rule. The execution of the assignment statement should be triggered whenever any of the variables used in the expression of the `always` statement changes.

3.3 Multiple views

Separation of a *view* object from a *subject* (or *model*) object is often important for easy modification of a user interface. In an AOUIS, these objects can be easily separated into different AUIOs, and the necessary interconnections between them can be provided by interface variables. Multiple views can be supported in the same way.

Fig. 10 shows a *subject*, which contains a value, and three *view* objects, which represent the value as a *barchart*, a *piechart*, and a *text*. Each view always reflects the current value of the subject.

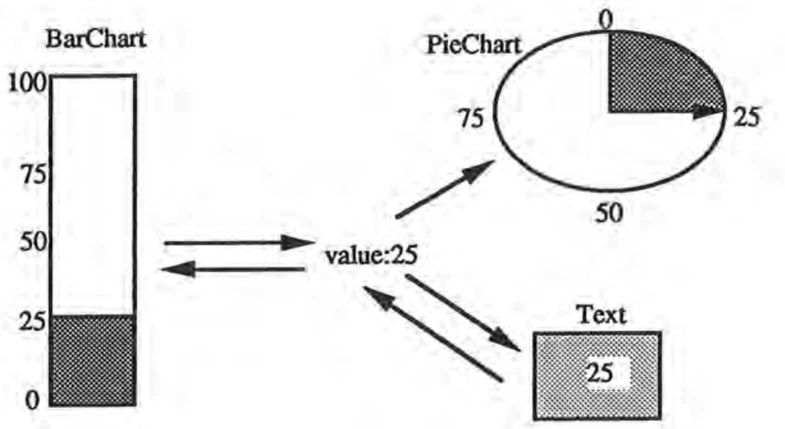

Figure 10: Multiple Views.

Furthermore, the barchart and the text can be manipulated by the user. If the user clicks the mouse button inside the barchart, the height of the bar is adjusted to the point where the mouse-down event occurred. The new height of the bar is then reflected in the value of the subject. This change further causes chain reactions in other view objects. Conversely, the user can directly edit the text representation of the value. Then the barchart and the piechart will be updated accordingly. We assume that the piechart only displays the value of the subject as a filled arc and cannot accept user input.

We now define class `MultipleViews` in Fig. 11.

```
class MultipleViews {
   public:
      int subject;
      BarChart barChart(100, 20) at (0, 0) with i_subject = &subject;
      PieChart pieChart(100) at (100, 50) with i_subject = &subject;
      Text text(20, 20) at (50, 0) with i_subject = &subject;
}
```

Figure 11: Class MultipleViews.

`MultipleViews` shows `subject` as a *subject* object and `BarChart`, `PieChart`, and `Text` as its *views*. `subject` is connected to the views through the interface variable `i_subject`.

In the following, we show how required classes for this example of multiple views can be defined. The class `MultipleViews` uses three component classes, `BarChart`, `PieChart`, and `Text`.

```
class BarChart : public UIObject {
   interface:
      int *i_subject;
   public:
      int height, width;
      BarChart (int h, int w) {height = h; width = w;};
      Rectangle frame(height, width) at (0, 0);
      FilledRectangle bar(width) at (0, 0);

      /* event routine activated by a mouseDown() message */
      on mouseDown(Position location) from mouse
         if (inside(location, region))
            *i_subject = location.y / height * 100;
      /* bar.height should always reflect the state of i_subject */
      always bar.height = *i_subject / 100 * height;
}
```

Figure 12: Class BarChart.

Class `BarChart` shown in Fig. 12 is a subclass of `UIObject`. The graphical representation of a `BarChart` consists of a rectangle `frame` and a filled rectangle `bar` which is dynamically updated. If an event `mouseDown` is received from `mouse` by an instance of `BarChart`, the variable pointed to by `i_subject` is updated to reflect the current position of the `mouseDown` event. The superclass of `BarChart` also contains a event routine for the `mouseDown` event. However, it is overridden by the event routine of `BarChart`. The height of `bar` is changed according to value of the variable pointed to by `i_subject`, as specified by the equational assignment statement `always bar.height= *i_subject / 100 * height`.

The class of the instance variable `bar` is shown in Fig. 13.

```
class FilledRectangle: public UIObject {
   public:
      int oldHeight = 0, height, width;
      FilledRectangle(int w) {width = w};

      /* transition rule which update shape of the filled rectangle */
      if (updated(height) && (oldHeight != height) {
         if (oldHeight > height))
            eraseRectangle(height, oldHeight, width);
         else
            filledRectangle(oldHeight, height, width);
         oldHeight = height;
      }
}
```

Figure 13: Class FilledRectangle.

Class `FilledRectangle` is defined as a subclass of `FilledRectangle`. If the interfacē variable `height` is updated and has a new value, `FilledRectangle` erases the region of the rectangle specified by `oldHeight` and `width` and draws a new rectangle specified by `height`. Instance variable `oldHeight` maintains the value of current `height`.

```
class PieChart : public UIObject {
   interface:
      int *i_subject;
   public:
      int radius;
      PieChart(int r) {radius = r};
      Circle circle(radius) at (0, 0);
      FilledArc fArc(radius) at (0, 0);
      always fArc.degree = *i_subject / 100 * 360;
}
```

Figure 14: Class PieChart.

Class `PieChart` is shown in Fig. 14. Graphical representation of `PieChart` consists of a `circle` and a `fArc`. It does not accept user input and only draws a piechart which reflects the current state of `i_subject`.

Fig. 15 shows class `Text`. `Text` accepts user input and echos the input to the display. The instance variable `eText` of class `EditableText` takes care of user input. `i_subject` and `text` of `eText` always reflect each other's state constrained by the equational statements `always *i_subject = atoi(eText.text)` and `always eText.text = itoa(*i_subject)`.

```
class Text {
   interface:
      int *i_subject;
   public:
      int height, width;
      Text (int h, int w) {height = h; width = w;};
      Rectangle rectangle(height, width) at (0, 0);
      EditableText eText at (0, 0);
      always *i_subject = atoi(eText.text);
      always eText.text = itoa(*i_subject);
}
```

Figure 15: Class Text.

4 Implementation Issues

In this section, we describe the implementation issues of the AOUIS runtime environment and the translator. The major problem is the activation mechanism for the behavior description routines such as transition rules, *always* statements, and event procedures. As these routines are activated by triggers, setting up the *triggers* for them is the main subject of this section. Further details can be found in [CHOI91]. A prototype AOUIS is being developed on top of the X window system. AOUIS programs are translated into C++.

4.1 Event Scheduling

An AOS computation is a sequence of executions of transitions, which transform the states of objects and generate events. These new states of objects and events may trigger executions of other transitions. We call an object that contains a transition a *source object*, since a transition contains the sources of references to other object states. The objects referenced by the condition parts of transitions are called *trigger objects*, since their state changes must trigger the executions of the transitions in source objects. An AOS computation can be best understood in terms of trigger objects and triggered transitions.

Fig. 16 shows this interaction between the trigger objects and the triggered transitions in a `ButtonGroupA` AUIO, whose class definition is given in Section 2.

If a `RadiobuttonA` receives a message `chosen()` from its super object, it modifies `selectedButton` in its parent component. Modification of `selectedButton` triggers the transition rule which has `selectedButton` in its condition part. When the transition in `RadiobuttonA` is executed, it modifies local variable `selected`. This modification triggers a further sequence of transitions inside the

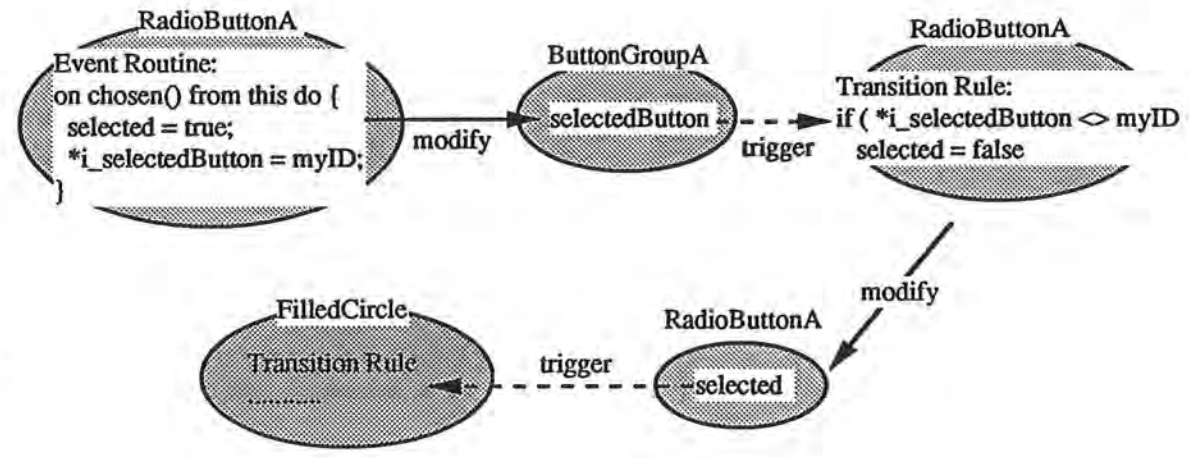

Figure 16: Interaction between trigger objects and triggered transitions.

graphical object `fcircle`, since `fcircle` should reflect the state of `selected`.

4.2 Trigger Setup

In order for the transitions in an AOUIS to work properly, whenever the state of an object changes, all the transitions that refer to that state in their condition parts should be activated. Hence, triggers should be setup for the objects whose states are referenced by transitions.

We now introduce some definitions used in our discussions. A *reference path* is a set of pointer variables which are involved in designating a trigger object from a source object. For example, the reference path for the reference expression `x.p->q->y.z` is (`x`, `x.p`, `*(x.p)`, `x.p->q`, `*(x.p->q)`, `x.p->q->y`, `x.p->q->y.z`). A pointer object that is an element of a reference path is called a *path pointer*. When the value of a path pointer is changed during the execution of an AOUIS program, it is called a *dynamic path pointer*.

In most cases, the reference path for a reference does not involve any dynamic pointers. In this case, the trigger for that reference need not be changed during the execution of the program. Since we can know the exact object that will be accessed when the source object is instantiated, the trigger for the reference need be setup only once when the source object is instantiated.

When the reference path of a reference involves a dynamic pointer, the trigger object for that reference changes when the value of that pointer is manipulated. If this happens, the trigger in the old trigger object should be dropped, and the new trigger should be added to the new trigger object. It is generally impossible to tell which trigger object is accessed until the operation that modifies the dynamic trigger object is actually executed.

When the value of a dynamic path pointer is changed, all the reference paths that involve the dynamic path pointer are affected. The details of the dynamic trigger setup mechanism is discussed in [CHOI91].

5 Conclusions

We proposed a new framework for constructing an object-oriented user interface system. The new framework, AOUIS (*active-object user interface system*), is based on active objects whose behaviors are defined by the transition statements provided in their class definitions.

We can construct an AOUIS class from its component classes as follows. First, pick-and-place instances of component classes. Second, to establish proper connections among the component objects, set their interface variables properly. Finally, specify the additional behavior of the new class by providing transition statements. This design process is similar to the process of constructing a hardware circuit from its components.

In this paper, we introduced the concept of an AOUIS, described its key features, and addressed its implementation method. In one of our experiments, we implemented a radio-button widget as an AOUIS class, and found that its code size was reduced to two thirds of that of the InterViews implementation of the same functionality. InterViews is a user interface system implemented in C++ according to a conventional approach.

We obtained the following insights on the effectiveness of the AOUIS approach. First, active objects, with three kinds of transition statements, show better encapsulation and achieve more flexible inter-object communication than ordinary objects. Second, structural interfaces help to modularize software objects like hardware objects. Finally, the AOUIS approach can support with ease such desirable features as multiple views and separation of views and models.

References

[AGHA86] Agha, G. A. *Actors: A model of concurrent computation in distributed Systems.* The MIT Press, 1986.

[BART87] Barth, P. S., An Object-Oriented Approach to Graphical Interfaces. *ACM Tran. on Graphics, 5*, 2, 1987, 142-172.

[BIRT73] Birtwistle, G., Dahl, O. J., Byhrhang, B., and Nygard, K. *SIMULA BEGIN*, Auerbach, 1973.

[BLAC86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald System. In Proc. OOPSLA'86 Conf. on Object-Oriented programming, 1986, pp. 78-86.

[CHOI91] Choi, S., and Minoura, T. Active Object System. Tech. Report 91-40-1, Computer Science Dept., Oreon State Univ., 1991.

[COOT88] Coote, S., et. al., Graphical and Iconic Programming Languages for Distributed Process Control: An Object Oriented approach. In Proc. IEEE Workshop on Visual Lnaguage, 1988, 183-189.

[DAVI76] Davis, R., and King, J. An overview of production systems. *Machine Intelligence, 8*, 1976, 300-332.

[ELLI89] Ellis, C. A., and Gibbs, S. J. Active objects: realities and possibilities. In *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. H. Lochovsky (Eds), ACM Press, 1989, 561-572.

[GOLD80] Goldberg, A., Robson, D. *Smalltalk-80 The language and its implementation*, Addsison-Wesley, 1983.

[HELM90] Helm, R., Holland, I. M., and Gangopadhyay, D. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, In Proc. ECOOP/OOPSLA'90 Conf. on Object-Oriented programming, 1986, pp. 169-180.

[KEHL84] Kehler, T. P., and Clemenson, G. D. An Application Development System for Expert-Systems. *Systems and Software 34*, 1984, 212-224.

[KNOL89] Knolle, N. Why Object-Oriented User Interface Toolkits Are better *JOOP, 2*, 6, 1989, 63-67.

[KUNZ84] Kunz, J. C., Kehler, T. P., and Williams, M. D. Applications development using a hybrid AI development system. *The AI Magazine, 5*, 3, 1984, 41-54.

[LINT88] Linton, M. A, Calder, P. R., and Vlissides, J. M. InterViews: C++ Graphical Interface Toolkit. Tech. Report CSL-TR-88-358, Stanford Univ., 1988.

[LIPK82] Lipkie, D. E., Evans, S. R., Newlin, J. K., and Weissman, R. L. Start Graphics: An object-oriented implementation. *Computer Graphics, 16*, 3, 1982, 115-124.

[LOND85] London, R. L.., and Duisberg, R. A. Animating programs using Smalltalk. *Computer, 18*, 8, 1985, 61-71.

[MEYE88] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

[MORI86] Moriconi, M., and Hare, D. F. Visualizing program designs through PegaSys. *Computer, 18*, 8, 1986, 72-85.

[REIS86] Reiss, S. P. An object-orietned framework for graphical programming. *ACM SIGPLAN Notice, 21*, 10, 1986, 49-57.

[SIBE86] Sibert, L., Hurley, D., and Bleser, T. An object-orietned User Interface Management System. *SIGGRAPH, 20*, 4, 1986, 259-268.

[STRO86] Stroustrup, B. *The C++ Programming Language*, Addison-Wesly, 1986.

[TSIC87] Tsichritzis, D., Flume, E., Gibbs, S., and Nierstrasz, O. KNOs: Knowledge acquisition, disemination, and manipulation objects. *ACM Trans. on Office Information Systems, 5*, 1, 1987, 96-112.

[WIRF90] Wirfs-Brock, R. J., and Johnson, R. E., Surveying Current Research in Object-Oriented Design. *CACM, 33*, 9, 1990, 105-124.

[YONE87] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. Modelling and programming in an object oriented concurrent language ABCL/I. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (Eds), The MIT press, 1987, 55-90.

[ZISM78] Zisman, M. D. Use of production systems for modelling asynchronous, concurrent processes. In *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (Eds), Academic Press, 1978, 53-69.