EVA: AN EVENT DRIVEN FRAMEWORK FOR BUILDING USER INTERFACES IN SMALLTALK

by Jeff McAffer
and Dave Thomas

SCS-TR-129

November 1987

School of Computer Science
Carleton University
Ottawa, Ontario
CANADA K1S 5B6

This research was supported in part by DREO.

Eva: An Event Driven Framework for Building User Interfaces in Smalltalk

Jeff McAffer
Dave Thomas

Carleton University
School of Computer Science
Colonel By Drive
Ottawa, Ontario

McAffer@Carleton.Bitnet
F95THOMP@Carleton.Bitnet

Abstract

Eva is an event driven framework for implementing user interfaces in multiprocessor Smalltalk systems. Most current user interfaces systems (e.g., Smalltalk, MacApp and NeWS), while event based, use polling and coroutines to handle user interaction. This makes them hard to understand and difficult to use in multiprocessor environments. Eva uses events to drive the Actors (light weight processes) which make up the user interface. The components of the user interface are constructed from parts rather than by inheritance. This structure increases modularity and allows the interface to be distributed over several processors. Comprehensive event management and synthesis simplify the creation of new parts. Eva is useful in both uni- and multiprocessor systems.

1 Introduction

Eva is an event driven framework for implementing user interfaces in multiprocessor Smalltalk systems. Most current user interface software such as Smalltalk, the Macintosh and NeWS are event based but not event driven. They use polling and coroutines to handle user interaction. These interfaces are based on the multiwindow interface first implemented in Smalltalk-80[1] [Krasner, 1985] which was one of the first systems to clearly separate the user interface objects (view and controller) from the application (model).

The Eva user interface paradigm is an alternative to the Model/View/Controllers (MVC) found in Smalltalk. Eva was first thought of as a mechanism for building a new user interface for Smalltalk [Nickel, 1986]. The central goal of this work is to provide true toolkit facilities which will allow application programmers to rapidly prototype new user interfaces. Partitioning the user interface classes to take advantage of functional multiprocessing on advanced multiprocessor workstations as in ACTRA [Thomas, LaLonde and Pugh, 1986a] is a secondary objective. Multiprocessing capabilities would allow different parts of the user interface to run on different processors (e.g., a screen manager on a TMS34010[2]). Even though this goal has had a great influence on Eva's overall design, Eva is useful in both uni- and multiprocessor systems.

1.1 User Interface Tools Are Hard To Use

There are many multiwindow user interface toolkits available (e.g., MacApp, NeWS, SUNWindows, MS-Windows). Nevertheless, most application developers encounter major difficulties when trying to create window based systems. There are four major problems with existing systems, they;

- are closed,
- use of polling and coroutines,
- over use of specialization versus aggregation and
- have hidden view coordination.

User Interfaces Need To Be Open

The user interface behaviour is hidden from the user (programmer), in systems such as the Macintosh and MS-Windows. When creating user interfaces for complex and sophisticated applications, developers must conform to what they are given. In closed systems, programmers do not know the full extent of what is possible. Because of this, doing something which is out of the ordinary is difficult since programmers cannot see or modify how the system's components function.

In Smalltalk, browsing the user interface classes helps people see how the existing data types interact and how new data types can be built. Some may argue that such open systems violate user interface standards or are too inefficient. We believe that the ability to restructure/modify the system is essential when designing more complex user interfaces. The interface designer must be given both fine and coarse grain control over the system.

This control can be used in the design of an application which is to be used by both able-bodied and handicapped persons. User's with motor disabilities would like to be able to adjust such critical parameters as the hysteresis of the mouse. The same is true of systems which are used on both high powered graphics workstations and character only displays. Both of these situations occur frequently, yet few systems allow for adjustment of these parameters.

Polling and Coroutines

The use of polling and coroutines to handle asynchronous events places too much responsibility on the application programmer who must obtain and release control of the processor. Polling systems are simple but they prevent true multitasking, restrict the interface to a single focus of control and force the user interface to keep the notion of an *active window* to maintain context. Further, since a polling loop will test the supported devices in a specific order, some inputs have a slightly higher priority than the others.

In a system based on polling, programmers can inadvertently lockup the system while debugging the user interface. Such a lockup typically leaves them asking the

[1] Smalltalk-80 is a registered trademark of Xerox Inc.

[2] The TMS34010 is a specialized graphics microprocessor from Texas Instruments.

question so frequently asked by Smalltalk users, "When I am in this part of the window what code gets executed when I do this (e.g., move the mouse)?". These problems arise because Smalltalk's controllers (discussed below) poll for both input and control. When input is sensed by one controller, it asks the other controllers if they want control, the first controller that wants control processes the input. This confusion is indicative of the problems surrounding polling mechanisms.

It is often difficult to implement a clean solution to a problem in a polling-based system. For example, the decision to use a single event loop in systems like the Macintosh makes writing complex applications very difficult and multitasking is hard to introduce (e.g., MultiFinder). As a result, the event loops are complicated and monolithic.

You Can't Do It All With Inheritance

There has been much written about the natural match between object oriented programming and user interface design. The design of abstract objects such as windows which do not correspond to any object in physical reality remains a challenge. We have found many programmers using Smalltalk's or Flavors' subclassing organization (inheritance) where an aggregation or parts organization would be better. Is it really true that a bordered window is a subclass of window? Should a text window with a scroll bar be different from one without? Subclassing leads to very cumbersome and confusing user interface datatypes, especially in a single inheritance model. Systems such as Animus [Duisberg, 1986] have clearly illustrated that it is better to think of a bordered window as a composite object [Lieberman, 1986] [Lalonde, Thomas and Pugh, 1986b] composed of a border part and a window part.

This means we need a system with lots of small parts such as buttons, text views, scroll bars, etc. which are assembled by the user interface designer. Combining these parts to form new parts should not require us to change existing code or write large amounts of new code. Examples of systems with this capability are ThingLab [Borning, 1979] and The Alternate Realities Kit [Smith, 1986].

Coordination Facilities Must Be Explicit

One of the most difficult parts of implementing an user interface is defining and maintaining the relationships between objects. Typically it is difficult to specify that when one object changes, some other object should be updated. Facilities for explicitly stating and maintaining these *dependencies* must be available so that the designer can clearly define what interactions are to take place.

Smalltalk contains a hidden and inflexible coordination scheme. Its *dependents* mechanism hides the relationships between objects by allowing them to interact behind the scene. When models think they have changed significantly, they indirectly send a generic update message to the views open on them. The interface (i.e., view) cannot specify what changes it is interested in nor can it control when it gets notified of these changes. The dependents mechanism is further restricted in that all messages go over a single message path which has a fixed format.

Other systems (e.g., the Macintosh) provide little or no coordination facilities that the user can access. These approaches are reasonable for simple applications which have few interacting objects. However, in a more general system which allows the user to compose arbitrarily complex objects from distinct parts, an explicit coordination scheme is needed.

1.2 Distributing The Interface And The Application

The interface facilities in existing systems like X [Scheifler and Gettys, 1986] are designed to have many users on many machines (processors). The machines have a client/server relationship with the interface running on the server and the applications running on the clients. Users require one server but may connect to any number of clients. In this model, which tasks should be performed by the client and which should be done by the server (i.e., the partitioning of the application and the interface) is not always clear. For example, who is responsible for updating the display when part of it is destroyed?

These systems require the designers to partition their product into application and interface parts which are distributed over the client and server. Eva, on the other hand, encourages but does not require this partitioning and can distribute both the application and the interface over many machines. Eva is used in a single user, multiprocessor environment similar to that found in Adagio [Tanner, *et al.*, 1985]. Because the components of Eva are Actors [Hewitt, Bishop and Steiger, 1973] (light weight processes) they are well defined and concrete. Each component is a self-contained object which has its own resources (including processing power) and communicates with other objects using a message passing protocol (e.g., Harmony [Gentleman 85]). This increased modularity makes it easy to incorporate new hardware into the system.

2 Models, Views and Controllers

Smalltalk uses the Model/View/Controller (MVC) paradigm in the design and implementation of its user interface. MVC is based on three kinds of objects; models, views and controllers. Models are data (e.g., text, numbers and collections). Views are visual representations of models. Controllers regulate interaction between the user and the view. (Note: We say that x is a view on y if a view x interprets the data of a model y, yielding some graphical representation.)

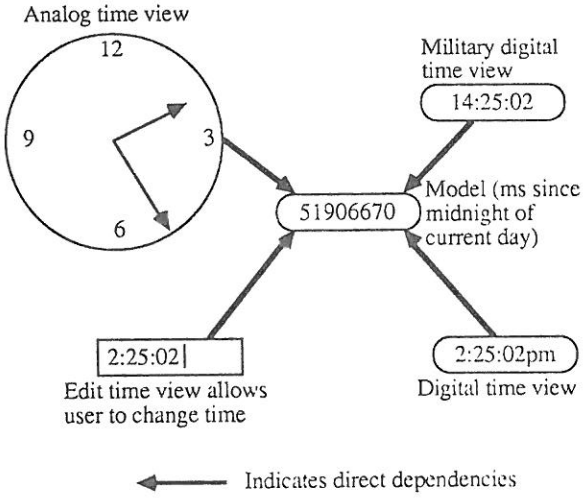

Figure 2.1. Time of day clock viewed several different ways.

The separation of data (i.e., model) and presentation (i.e., view) has several advantages. Each view is a particular interpretation of all or part of the data in the model. This means there may be several different views on the same model (e.g., a model containing the time of day could be viewed as an analog clock, a digital clock, etc.) (see Figure 2.1). This

separation improves the modularity and clarity of an application's code and allows code to be reusable. Designers can quickly prototype several user interfaces and switch between them without disturbing the application's code.

The MVC paradigm requires programmers to deal with the user's input on a low level. Controllers contain routines for sensing and handling user inputs such as; mouse movement, keypresses and button clicks. The controllers are constantly polling for input using these routines. When some event occurs, the controller must determine its context (e.g., a shifted mouse button down event may be different then an unshifted one) and do the appropriate action. This approach has two major problems, polling and lack of modularity.

Polling limits the capabilities of the MVC paradigm. The use of processor time to test for incoming events is the primary shortcoming, especially in a multitasking environment. Polling becomes very difficult and inefficient in a multitasking environment because it limits the system to sequential processing of user input and eliminates the possibility for multiple input foci. In addition, polling code is typically littered with special case tests and exceptions. The thread of control becomes confusing because of multiple nested polling loops in the controllers. Secondly, there is considerable processing overhead involved in finding the *active view* and the correct context for an event.

Since each type of view has its own input requirements and set of valid user interactions, each view has an associated controller which handles user input. In a typical system there are several different types of view and thus several different types of controller. These view/controller pairs normally fall into one of four broad groups; those dealing with text, lists or graphics and those dealing with aggregates of the above three.

Within a particular view/controller pair the code is modular, however the same is not true of the system as a whole. In each group, the view/controller pairs are organized hierarchically with subclasses differing only slightly from their superclasses and sharing most of their code. Because polling code tests a particular set of inputs, if a subclass recognizes one different input then it must reimplement the polling loop. This leads to a great deal of duplicate code and confuses the flow of control.

To illustrate some of these problems, consider the Smalltalk user who wishes to monitor all incoming data on a serial line and who does not want to create a separate task to manage this data. It is straightforward to create a text view which takes its input from the serial port. The problem is that its controller will only be active when the view is the *active view*. Using Smalltalk for anything else will prevent the controller's polling loop (the one that tests the serial port) from running. Solving this problem requires changing all of the polling loops in the system to include the serial port as one of the inputs.

Construction of new Model/View/Controller triads is complex and cumbersome despite the existence of tools such as Glazier [Alexander, 1987]. Views frequently interact with others in subtle ways causing one controller to retain control of the processor and lockup the system.

Typical implementations of Smalltalk do not use MVC for everything. This leads to confusion about what particular objects should be doing. For example, most controllers have processes associated with them. The code which handles user input for popup menus does not. In fact, this code, which plays the role of a controller, is not implemented as a controller. This inconsistency increases complexity and reduces compatibility between systems and between components within systems.

MVC in Smalltalk uses the dependents mechanism to maintain the correspondence between model and view. Objects can *depend* on one another such that when one is changed, all those who depend on it will be notified. This scheme is reasonable for simple applications but it allows only a single path between dependents (i.e., the changed: and update: messages). Unfortunately, an object's dependents rely on the object to decide when to send the changed: message and the dependents who receive the update: message must determine whether or not they are interested in the change.

3 Eva and NeWS

Although Eva and NeWS have evolved independently, they have a number of similarities. Both support the notion of events and multiple active views (processes) to support *light weight windows*. They both argue that complex user interfaces should be implemented in a portable interpretive language. This reflects an increasingly popular view that *reactive* applications consist of interacting objects which invoke application *devices* or stubs to perform work. They differ from traditional library based approaches such as GKS [ISO, 1982] [ANSI, 1984], where the user interface consists of a single application driven loop which invokes library procedures to perform the user I/O. It is encouraging to note that system designers from graphics, operating systems and object oriented programming have converged on the same idea.

We believe that Smalltalk provides a much better foundation for complex user interfaces. NeWS is based on POSTSCRIPT [Adobe Systems Inc., 1984], which is essentially Forth with a very flexible graphics imaging model. POSTSCRIPT (Forth), while good for device controllers and the like, does not meet the needs of large applications. POSTSCRIPT lacks a development environment (i.e., the browser and debugger), the garbage collector [Roberts, *et al.*, 1987] and the class library of Smalltalk. If we are to produce complex applications (e.g., a CAD system which conforms to the PHIGS [PHIGS] standard) then a rich programming environment is of utmost importance. We do not have to give up performance for sophistication. This despite the widely held view that Smalltalk systems are larger and less efficient than more traditional systems [Krasner, 1983]. For example, there are several Smalltalk systems (e.g., ParcPlace[1], Smalltalk/V[2]) which execute faster than the dedicated POSTSCRIPT interpreter in the Apple LaserWriter. Smalltalk/V requires only 600K to run and it is an entire programming environment complete with compiler, debugger and inspectors. Future high power, "crayola" class, workstations will make differences in system performance insignificant.

4 Eva

The Eva framework uses three classes; Events, Views and Models. Events describe actions. Views present a model's data graphically and handle high level user interaction with the model. Models are data, typically part of the application.

4.1 Event Management

The mainstay of Eva's control mechanism is the Event. Rather than having one controller handling the user input for each view, Eva has one manager for the entire system, the

[1] ParcPlace Smalltalk is a product of ParcPlace Systems, Palo Alto, CA

[2] Smalltalk/V is a registered trademark of Digitalk Inc

event manager. This event manager can be thought of as managing an interrupt table. Views register with the event manager telling it which events the view wants to handle and the manager interrupts the view when one of these events occurs. If a view wishes to know when the left mouse button is clicked inside its bounding box, it would register for the synthetic event leftButtonClickedIn: (synthetic events are discussed below). Whenever the left button is clicked in the view a leftButtonClickedIn: message will be sent to the view which will then process the event (see Figure 4.1 below).

The components of a user interface are more clearly defined in the Eva paradigm than in MVC. Programmers using MVCs are continually facing the question, "should this operation be done by the controller or the view?". In Eva this problem is reduced because the responsibilities of the event manager are more clearly defined and intuitive. It is the event manager's job to maintain the view's interests in events and inform interested views when these events occur. It is the view's job is to handle the events from the event manager by modifying the model and updating itself as appropriate.

The task of maintaining and interpreting the interface context is split over several managers, the event manager, the screen manager and the mouse manager to name a few. Typically there is one manager for each input or output device although. Each of the managers is an Actor and so executes in parallel with the others. It is the job of these managers to combine user input and interface context creating a meaningful stream of events for each view.

In Eva it is easy to understand how a particular view behaves. Each event has a corresponding handler (method) in each of the views which are interested in it. These views must explicitly state which events they are interested in by registering for them. When an event occurs, control is passed to the corresponding handling methods for each of the registered views.

A view can be *registered* for an event which means that it is possible for the view to handle that event. If a view is *enabled* for a particular event then that view currently wants to handle occurrences of that event. Only views which are enabled for an event are informed when the event occurs.

The event manager keeps track of all views and the events for which they are registered in its *registry*. Events and the views enabled for them are kept in the event manager's *event table*. The event table is used in the same way as an interrupt vector table. When an event occurs the list of handlers (views enabled) for that event is retrieved from the event table. Each of the handlers in the list are informed of the event's occurrence.

The set of events which are valid at a given time are those which are in the manager's event table. Invalid events will be ignored. Old and new event types can be added or removed at any time with no adverse effects on the rest of the system. The underlying code for the event manager can also be changed without effecting existing views. Ideally the host system would not require the event manager to poll for primitive events (e.g., mouse button down and up, etc.). Polling, if required, can be limited to a few device classes.

In practice the concept of a single event manager would be broken down into the its component managers. For example, a mouse manager would be responsible for distributing the mouse related events. Because there are many input (event) sources individual views can receive input from many sources at the same time. In addition, consecutive events in a view's event stream need not be related or have been generated by the same device. Therefore, in Eva, it is possible to have concurrent active views with each one having multiple input foci.

4.2 Synthetic Events

A more meaningful and intuitive stream of events is provided by using synthetic events. Synthetic events [Cardelli, 85] are logical events which occur as the result of a specific sequence of real or physical events. For example, a leftButtonClick: event occurs whenever a leftButtonDown:, leftButtonUp: event sequence occurs within a given time frame. Further, using the state of the user interface, more complex event synthesis can be done. The leftButtonClickIn: is generated when a leftButtonClick: event (a synthetic event) occurs inside a view (see Figure 4.1). Sophisticated event management and synthesis improves the simplicity and modularity of the view code.

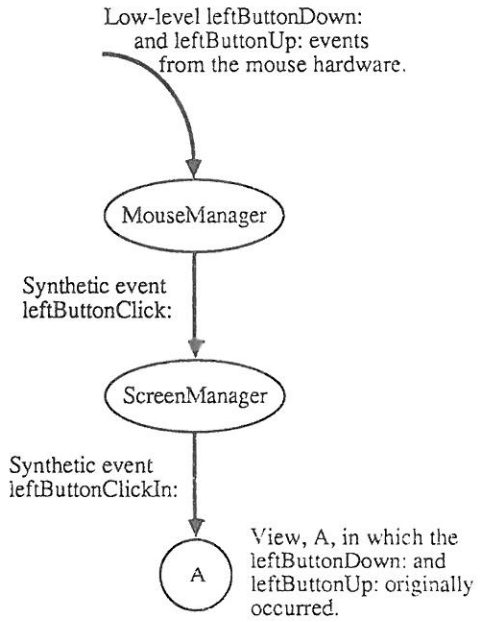

Figure 4.1. leftButtonClickIn: event creation/distribution.

In addition to the events generated by user interaction and the event manager, the application can create and post events (software events). Software events have higher priority than normal events. This allows applications to synthesize events and still maintain proper temporal ordering. Suppose four leftButtonClick: events were generated in a particular view and the related application dictates that three consecutive button clicks defines some event, x:. When the application receives the first three events it will create and post a x: event. This new event should be processed before the fourth button click since it happened before the click. Although not necessary, a sophisticated event manager can have an event priority scheme to attach levels of importance to particular events and thus particular input sources.

4.3 View management

It is quite common for a user to want to treat a collection of different views as one logical view. The Smalltalk class browser for instance contains several different subviews (a class list, selector list and code view, etc.). These simple subviews are all part of the same complex view, a browser. This relationship must be well defined.

Since individual views define which events they are interested in, and are responsible for their own operation combining several views to create arbitrarily complex views is easy. We need only to provide a general mechanism for grouping views together, a *complex view*. A complex view is a collection of views which acts as one. It is a container and a coordination mechanism for these views. Complex views respond to typical view methods (e.g., display, open, close, etc.) such that sending the open message to a complex view is like sending open to each of its subviews.

Complex views allow a user interface designer to create views with several different parts without modifying any code, he simply adds components to the collection. The components can be anything from a border or label for the view to another complex view. By adding one complex view to another complex view several times, the user can create an arbitrarily complex hierarchy. Suppose the designer wanted to combine a debugger and a class browser. He would create a complex view and add the debugger's view and the class browser's view, both of which are complex. Neither the code for the debugger nor the browser has to be modified.

Complex views are created from parts (see Figure 4.2) and have no displayable form of their own. They create a logical grouping, not a physical or graphical one. For this reason, the subviews of a complex view need not be grouped together and may overlap on the screen.

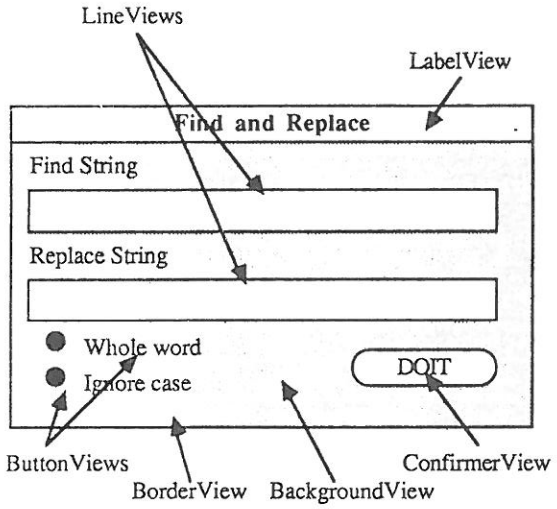

Figure 4.2. Example complex view (a FindAndReplaceView).

A complex view also serves as a coordinator for its subviews. The subviews can communicate with each other through the complex view. One kind of complex view is a screen manager (the screen is just a view with several subviews). The screen manager ensures that the proper views are visible and updates the screen when views are moved or deleted.

4.4 Model/View Interaction

As in MVC views are dependent on their models. Views can know about the models they represent but a model must not require knowledge of the views which may be on it. A model must be entirely independent of its view. This means that implementing a view for a particular object should not require any modification of that object's behaviour. This leaves us with the question, "How does the view know when the model has changed?". There are several ways of dealing with this problem.

What is needed is a coordination mechanism for the view and model. This coordinator can tell when some action will cause the value of the view to be changed. Sensing that the model has changed, the coordinator tells the view which takes the appropriate action. Since the coordinator forms a much used link between the model and view it must be light weight.

A constraint satisfaction system such as the Filters Paradigm [Ege, 1986] [Grossman and Ege, 1987] is one such system. Filters combines light and heavy weight constraints (filters and logic) to coordinate user-object and object-object interaction respectively. Constraint Hierarchies [Borning *et al.*, 1987] provides more complete but expensive constraint specification. Both are built on top of ThingLab and allow the user to explicitly define what the interactions between entities are.

Another approach to this problem is the use of encapsulators [Pascoe, 1986]. An encapsulator is an object which surrounds another object intercepting all messages to it. In this way, the encapsulator can sense when a "significant" change occurs and directly inform interested objects. In both solutions a nice way of specifying what constitutes a significant change remains a challenge.

5 Implementation

A non-Actor uniprocessing prototype of Eva is implemented in Smalltalk/V. Eva has three logically separate parts: events, views and models. The event manager consists of three main classes: Event, EventStream and EventManager. Views in Eva are different from those in MVC because they handle the user's interaction directly, that is, Eva does not require controllers. This section briefly describes each of the major Eva classes.

In Smalltalk/V the View and Controller of MVC have been renamed Pane and Dispatcher respectively. In the following discussion these terms are used interchangeably and the word *view* is to be associated with Eva views unless otherwise stated.

5.1 Events

Event

Events describe actions carried out by the user. Control, activation and deactivation is dispensed on the basis of event occurrence. There are many kinds of events; keypresses, mouse movement, button clicks in addition to software generated events. All have specific types and may optionally have a timestamp, an originator (i.e., who caused the event) and data related to the particular event. See Figure 5.1 for an example event. There may be arbitrarily many kinds of events in a system. Table 5.1 contains a list of some of the events found in an Eva system.

event type	originator	time (ms)	data
<#leftButtonDown:	nil	31276152	230 @ 400>

Figure 5.1. An example of a leftButtonDown: event.

Events are named uniformly throughout the system. The leftButtonDown: event means the same thing to all objects in the system. This improves readability and portability.

```
left/rightButtonDown:
left/rightButtonUp:
left/rightButtonClick:
left/rightButtonClickIn:
left/rightButtonFirstClickIn:
left/rightButtonFirstClickOut:
left/rightButtonDrag:
keyPressed:
cursorMove:
hotSpotEnter:
hotSpotExit:
updateModel:
```

Table 5.1. Some standard Eva events.

EventStream

An EventStream supplies a continuous sequence of Events to the EventManager. The EventStream reads system-specific events from the Smalltalk kernel and transforms them into Events. This is so the Event data is consistent from system to system. In the current implementation of Eva it is possible to specify priorities for particular events.

EventManager

The EventManager is the central dispatcher of events. A registry of objects and their associated events is maintained as is an event table which holds events and their known handlers. In contrast to MVC's controllers, which contain many control loops, Eva has one event dispatcher. By using the state of the interface (e.g., the mouse and the screen), Eva can create very high-level synthetic events. When events arrive from the EventStream the EventManager first tries to use them to make a synthetic event by looking at the new event in the current context of the interface (see Figure 5.2). This context includes the layout of the views on the screen (from the ScreenManager), the events which immediately proceeded the new one (from the EventManager) and information regarding the events views are enabled for (from EventManager's eventTable). The synthetic event generated, or the original if synthesis was not possible, is then dispatched to its associated handler(s). An event is distributed to all of the handlers enabled for it.

It should be noted that Figure 5.2 depicts the non-Actor version of Eva. The use of Actors makes it possible to reduce the number of events the EventManager processes. For example, mouse events can be handled and distributed by the MouseManager, not the EventManager. Views may register directly with the MouseManager for these mouse events. This will eliminate any potential for a performance bottleneck at the EventManager.

Synthesizing events helps to reduce the number of views that get activated. Suppose that the highest level of mouse button event available is leftButtonUp: and leftButtonDown:. When the left button is clicked, every view is informed of the button down and up events. This means that each view would have to see if the mouse is in its bounding box and combine the events to make a click. Using event synthesis and the state of the interface, the leftButtonClickIn: event can be generated instead. This greatly simplifies the view's event handling code and increases efficiency.

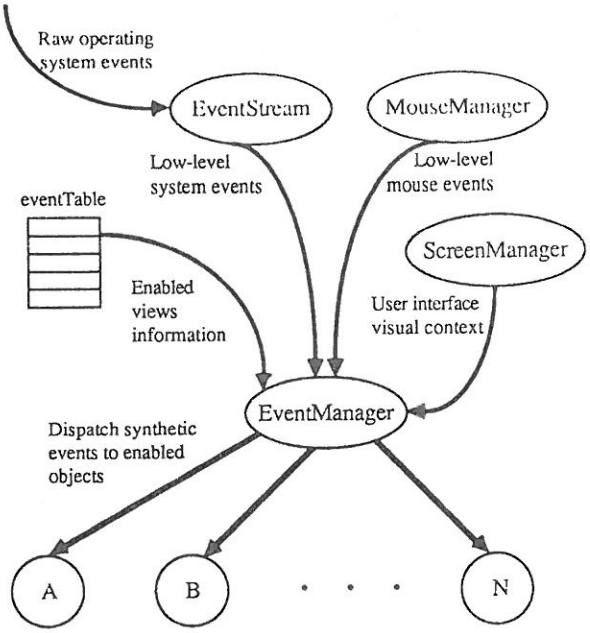

Figure 5.2. Chronology of event synthesis and distribution.

When the EventManager is distributing a particular event, say leftButtonClickIn:, it looks up the event's handler list in its event table. Each of the handlers in the list is informed that the leftButtonClickIn: event has occurred. An example leftButtonClickIn: event handling method is shown in Figure 5.3.

```
leftButtonDownIn: anEvent
    "TextView method - put gap selector before character
    nearest the point where the button went down."

    | downPoint |
    downPoint := anEvent data.
    model selectAt: (self stringCoordinateOf: downPoint)
```

Figure 5.3. Example of an event handling routine

For an object to receive a particular event, say leftButtonUp:, it must do the following:

- implement a leftButtonUp: method which will handle the event, and
- *register* for leftButtonUp:,
- *enable* itself for leftButtonUp:

Registering (deregistering) informs the EventManager that the object is (not) interested in a particular set of events. Enabling (disabling) an object insures that the object will (not) be informed when the events it is registered for occur.

Synthetic events, as discussed above, improve the readability and performance of event handling code. The EventManager, in conjunction with the ScreenManager and the MouseManager (see below), is responsible for generating synthetic events. When the EventManager synthesizes events, it attempts to create the highest level event possible in the state of the interface. For example, leftButtonFirstClickIn: will be generated instead of leftButtonClickIn: if it is the first time the

button is clicked in a window and the window is enabled for leftButtonFirstClickIn:.

In the current implementation of Eva, event synthesis performed by having the various managers which register for the low-level device events (e.g., mouseButtonDown: for the MouseManager) and generate higher level synthetic events based on their state. In this way the event generating portion of Eva is a state machine.

Event synthesis would be handled better by a series of simple state machines. One or more state machines would be attached to a particular event source (e.g., mouse or keyboard) to define all of the synthetic events. It is important to have a concrete way of creating these state machine because user interaction is complex. Squeak [Cardelli, 85] is a language for defining such state machines. Squeak fits nicely with our design and may be a useful future addition.

5.2 Views

View

Simple views (e.g., text or list views) in Eva are subclasses of the class View. Views must have code associated with them which perform all of the necessary display operations and handle any associated events. The implementation and use of these classes is much simpler and smaller than that of similar view/controller pairs in MVC.

ComplexView

The complex views discussed above are implemented in the class ComplexView, a subclass of View. A ComplexView maintains a list of its subviews and serves as their coordinator. Subviews can send messages to all of the other subviews by asking its superview (a ComplexView) to distribute the message. ComplexViews also coordinate external input. For example, sending the display message to a ComplexView would cause it to send display to each of its subviews.

ComplexViews equal the sum of their parts and have no displayable form of their own. That is to say, displaying a ComplexView which had no subviews would draw nothing on the screen. Everything shown on the screen is a subview, including the label and the border. This removes the need for complex class hierarchy trees and, if multiple inheritance is not available, duplicated code.

ScreenManager

ScreenManager, a subclass of ComplexView, is responsible for maintaining the appearance of the display screen. Views which are to be displayed are added to the current ScreenManager and as a result, are drawn on the display screen. The EventManager uses the ScreenManager during event synthesis to determine the state of the user interface. This aids in identifying the synthetic events which can be generated.

Views on the screen (in the ScreenManager) can be moved, sized, collapsed or closed. The ScreenManager takes care of the low level updates of the physical screen. When a view is removed or repositioned, the ScreenManager is responsible for redrawing the views which were under the changed view thus, views can overlap arbitrarily. Moving or resizing a ComplexView which has a border will affect all of its subviews.

ViewEditor

In the current implementation of Eva there is also a view editor, the ViewEditor. This editor allows the user to create, remove and modify views on the screen. New views are added by selecting the view type from a menu and placing the created view on the screen. In most cases, creating a view in the ViewEditor will also create a default model specific to that type of view. The new view can be put anywhere on the screen or associated with an existing ComplexView. Existing views are edited by selecting the view and carrying out the desired operations on it. This simple ViewEditor has been very useful in creating application interfaces. A system such as Interacticons [Smith, 1987] would be even more useful when designing complex interfaces.

5.3 Models

In the current implementation of Eva, models do not communicate with their views directly. Models are encapsulated as described above and these encapsulators inform the views when a model's value has changed. The encapsulator is like an EventManager in that a view can specify which messages it is interested in. When this message is sent to the encapsulated object the view is notified. The view can then determine if the change was significant enough to warrant updating itself.

In this way, the model does not need any knowledge of the views. Another way of maintaining the correspondence between model and view is to have the views update themselves when they change the model. There are two problems with this. First, a view would then require knowledge of how particular operations affect its model (i.e., in depth knowledge of the model's structure). If the model's behaviour is changed then the code for all of the views on it must also be changed. Secondly, how does one view find out about changes made to its model by some other view? This requires some sort of dependency mechanism. We are back where we started.

It is not clear whether a constraint system as described in section 4.4 would be better than encapsulators alone. Both allow arbitrary objects to be used as models for views and both would make the relationship between model and view clearly defined. The choice depends on the amount of constraint satisfaction required. If little is required then the two methods are roughly equivalent.

6 Conclusions

The Eva paradigm is logically much more object oriented than the MVC model. Events, Views and Models are all very clearly defined and independent of each other. Elements of Eva communicate via explicit messages rather than subtle interactions. Views are independent, self-contained objects, Actors and each of the parts in a complex view is responsible for its own functioning. More complex objects are constructed by composing parts rather than by inheritance. Views are independent of one another in the sense that each may be combined with any of the others. Complex views within complex views maintain these properties.

The implementation discussed above has shown that Eva is feasible. More over it has proven to be a reasonably good alternative to the Model/View/Controller paradigm. MVC is difficult to use in a multiprocessor environment, less modular and more restricted. Eva is easy to understand, less complicated to use/understand and more intuitive.

To date Eva has been used in small applications [McAffer, 1987] but has yet to be tested in a more complex multiprocessor environment. Eva's Model/View coordination scheme is currently being improved so dependencies are more explicit and maintainable.

Work in related areas includes the development of SmallScript, [Haaland, 1988] and the implementation of a POSTSCRIPT interpreter for Smalltalk [Nguyen, 1988]. SmallScript is a Smalltalk facility which uses the POSTSCRIPT imaging model and allows for the creation of NeWS-like client/server relationships between Smalltalk running on several machines.

7 References

Adobe Systems Inc. (1984) *POSTSCRIPT Language Manual*, Adobe Systems Inc., Palo Alto, CA.

Alexander, J. H. (1987) Painless Panes for Smalltalk Windows, *Proceedings of OOPSLA '87*, ACM SIGPLAN Notices, volume 22, number 12, p.287-294.

ANSI (1984) *dpANS GKS*, X3H3/83-25r3, ANSI X3H3 Project 362, January 5th, 1984.

Borning, A. (1979) *ThingLab - A Constraint-Oriented Simulation Laboratory*, PhD Thesis, Stanford University.

Borning, A., *et al.* (1987) Constraint Hierarchies, *Proceedings of OOPSLA '87*, ACM SIGPLAN Notices, volume 22, number 12, p.48-60.

Cardelli, L., Pike, R. (1985) Squeak: a Language for Communicating with Mice, *Proceedings of SIGGRAPH '85*, volume 19, number 3, p.199-204.

Duisberg, R. (1986) *Constraint-Based Animation: The Implementation of Temporal-Based Constraints in the Animus System*, PhD Thesis, University of Washington.

Ege, R. (1986) *The Filter - A Paradigm for Interfaces*, Technical Report No. CSE-86-011, Oregon Graduate Center, Beaverton, OR.

Pascoe, G. (1986) Encapsulators: A New Software Paradigm in Smalltalk-80, *Proceedings of OOPSLA '86*, ACM SIGPLAN Notices, volume 21, number 11, p.341-346.

Grossman, M., Ege, R. (1987) Logical Composition of Object-Oriented Interfaces, *Proceedings of OOPSLA '87*, ACM SIGPLAN Notices, volume 22, number 12, p.295-306.

Haaland, K. (1988) *SmallScript: Interfacing Smalltalk and Display POSTSCRIPT*, Honours project, Carleton University, School of Computer Science, Ottawa, Ontario, Canada.

Hewitt, C., Bishop, P. and Steiger, R. (1973) *A Universal Modular Actor Formalism for Artificial Intelligence*, IJCAI-73 Stanford, CA.

ISO (1982) *Graphics Kernel System (GKS) Functional Description*, Draft International Standard ISO/DIS 7942, ISO TC 97/SC5/WG2 N 163, November 14th, 1982.

Krasner, G. (1983) *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, 1983.

Krasner, G. (1985) Personal communication with Dave Thomas, June 1985.

LaLonde, W., Thomas, D. and Pugh, J. (1986b) An Examplar Based Smalltalk, *Proceedings of OOPSLA '86*, ACM SIGPLAN Notices, volume 21, number 11, p.322-330.

Lieberman, H. (1986) Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, *Proceedings of OOPSLA '86*, ACM SIGPLAN Notices, volume 21, number 11, p.214-223.

McAffer, J. (1987) *The User Interface Construction Kit*, Honours project, Carleton University, School of Computer Science, Ottawa, Ontario, Canada.

Nguyen, T. (1988) *A POSTSCRIPT Interpreter for the TMS34010*, Honours project, Carleton University, School of Computer Science, Ottawa, Ontario, Canada.

Nickel, R. (1986) *Towards a New User Interface for Smalltalk*, Masters thesis, Carleton University, School of Computer Science, Ottawa, Ontario, Canada.

PHIGS, *Programmer's Hierarchical Interactive Graphic System*, B.S.R. X3.144, Global Engineering Documents, Santa Anna, CA.

Roberts, W., *et al.* (1987) *First Impressions of NeWS*, Department of Computer Science, Queen Mary College, London, England.

Rosenthal, D. (1987) *Window Systems Implementations*, Washington DC Usenix Course Notes.

Scheifler, R., Gettys, J. (1986) The X Window System, *Transactions on Graphics #63*, Special Issue on User Interface Software, ACM.

Smith, D. (1987) *Interacticons*, OOPSLA '87 Video Show.

Smith, R. (1986) The Alternate Realities Kit: An Animated Environment for Creating Interactive Simulations, *Proceedings of: 1986 Computer Society Workshop on Visual Languages*, Dalas, TX.

Tanner, P., Wein, M., Gentleman, W. M., MacKay, S., Stewart, D. (1985) The User Interface of Adagio, A Robotics Multitasking Multiprocessor Workstation, *Proceedings of the 1st International Conference on Computer Workstations*, p.90-98, San Jose, California, November 11-14, 1985.

Thomas, D., LaLonde, W. and Pugh, J. (1986a) *ACTRA, A Multitasking/Multiprocessing Smalltalk*, Technical Report No. SCS-TR-92, Carleton University, School of Computer Science, Ottawa, Ont.