Mixing Visual and Textual Programming in a Functional Language

by

Gekun Song

A thesis submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the

requirements for the Degree of

MASTER OF SCIENCE

Supervised by Dr. T.C. Nicholas Graham

The Graduate Programme in Computer Science

Department of Computer Science

York University

North York, Ontario, Canada

Contents

1 Introduction **1**

1.1 Motivation for the Mixed-form Language 2

1.1.1 Why User Interfaces Are Hard to Develop 2

1.1.2 Why Build Support into the Language 4

1.1.3 Textual Languages vs. Visual Languages 5

1.1.4 Imperative Languages vs. Declarative Languages 5

1.2 The Mixed-form Language 6

1.3 Thesis Overview 10

2 Related work **12**

2.1 Purely Textual Approaches 12

2.1.1 Xt toolkit 12

2.1.2 InterViews 13

2.2 Visual Languages 15

2.2.1 Pict/D 15

2.2.2 ThingLab 17

2.3 Hybrid Approaches 18

2.3.1 NeXT Interface Builder 19

2.3.2 HyperCard 20

2.3.3 GVL: Graphical View Language 23

2.3.4 Pic++ 24

2.3.5 VisualWorks 26

2.4 Conclusion 27

3 Clock System and the Textual View Language **29**

3.1 The Clock Hierarchical Architecture 31
3.2 Component Specification 37
3.2.1 Event Handlers 39
3.2.2 Request Handlers 44
3.3 The Clock Textual View Language 45
3.3.1 The Underlying View Model 46
3.3.2 Basic Concepts of Coordinate Spaces 48
3.3.3 Display Primitives 49
3.3.4 The Views Primitive 54
3.3.5 Coordinates and Coordinate Spaces 55
3.3.6 The Positioning Primitive 58
3.3.7 Function Evaluation 59
3.3.8 Conclusion 60

4 Mixed-form Visual-textual Programming **63**

4.1 Motivation 63
4.1.1 Norman's Gulfs of Execution and Evaluation 65
4.1.2 Gap between Pictures and Their Textual Representations 67
4.1.3 Poor Comprehensibility of Textual Representations 72
4.1.4 Shortcomings of Graphical Primitives 74
4.2 Mixed-form Programming in Clock 76
4.2.1 An Equalizer Application 77
4.2.2 Advantages Offered by Mixed-form Programming 83
4.2.3 Mixed-form Programming in Clock 84
4.2.4 The Graphical Editor 85
4.3 Graphical Primitives in the Mixed-form Language 89

4.3.1 Canvas and Positioning Primitives 89
4.3.2 Display Primitives 91
4.3.3 Function Evaluation Primitive 92
4.3.4 The Grouping Tool 93
4.4 Design Tradeoffs 94

5 Syntax and Semantics of the Mixed-form View Language 96

5.1 Syntax of the Mixed-form Language 97
5.2 Semantics of the Mixed-form Language 99
5.2.1 Mapping from Graphical Primitives to Textual Equivalents 100

6 Mixed-form View Specification Examples 108

6.1 A Score Board 108
6.2 An Interactive Calculator 111
6.3 A BBE facility 118
6.4 Conclusion 120

7 Summary and Conclusion 122

7.1 Thesis Summary 122
7.1.1 Problems with Purely Textual or Visual Languages 122
7.1.2 The Mixed-form Visual/Textual Language 124
7.2 Future Works for the Clock System 125
7.3 Conclusion 126

List of Figures

1 An example of mixed-form Clock program. The display view function is specified with both textual and graphical primitives. The picture generated by this mixed-form specification is also illustrated. The four small magnets attached to each box are used to specify geometric constraints. 7

2 A graphical specification and its corresponding textual representation. 9

3 A graphical definition of a MidPointLine. The definition includes labels for the line's endpoints and midpoint, P1, P2 and M. The network on the right constraint the value of the midpoint to be equal to the average of the values of the two endpoints of the line. 18

4 The structure diagram of HyperCard. 22

5 An illustration of the HyperCard inheritance chain. 23

6 The overloaded plus operator and the Group function in Pic++. 26

7 A calculator application and its architecture tree. The calculator is composed of a display screen and a key pad. The key pad is made up of numerical keys, operator and functional keys. 32

8 Event handler "EqualKey" and request handler "Depressed". . . 33

9 An architecture tree for the calculator program showing request handlers. For each tree node, event handler is drawn at the left and request handlers at the right. 34

10 Architectural details of the "NkPushable" component. In this graphical representation, a line with one arrow denotes a update or input. A line with two arrows denotes a request. Single arrow lines on the right side of a component represent updates issued by the component and single arrowed lines on the left hand represent updates or inputs taken by the component. Double arrowed lines on the right side represent requests issued by the component and double arrowed lines on the left side represent requests handled by the component. 36

11 The interactive calculator program running under Clock. 38

12 "NumKey" subtree and pictures corresponding to each node. The views for "NkPushable" and "NumBox" are specific for instance when "Id" is "0". However, they are applicable to other Ids. . . . 40

13 Name boxes. 50

14 A picture generated using the "Box" primitive. 51

15 A box with stretching top-right vertex. 52

16 Two boxes with fixed position and size. 53

17 A picture that demonstrates the specification of relative size and position. 56

18 Mixed-form programming is somewhere at the middle point between purely textual and purely graphical programming. 65

19 Illustration of the stages of execution and evaluation. 66

20 A five-point star example. 69

21 A view function in graphical representation. v is a predefined display view, and *border* v adds a gray border to it. 73

22 A dynamic graphic equalizer. 77
23 Mixed-form specification for the equalizer display. 79
24 View specification for the increase button when it is not pressed. 80
25 Visual specification for the left arrow button when it is not pressed. 81
26 Mixed-form specification for the equalizer application. 82
27 The Clock graphical editor. 86
28 Handles and magnets 88
29 A simple score board example. The interface contains two huge digits showing the scores of two teams in a sports match. Two buttons labeled "+1" are used to update the scores. As no font is big enough to display characters as huge as that, we have to piece together gray boxes to compose the digit displays. 109
30 The graphical specification for digit "0". 110
31 An interactive arithmetic calculator. This calculator application allows the user to perform simple arithmetic operations. The user can enter operands by pressing the ten digit buttons or enter operators by pressing the five available operator keys. A "CA" key is used to reset the calculator. 113
32 The architecture diagram for the interactive calculator example. 114
33 The mixed-form view specification for subview "nkey". 115
34 The mixed-form view specification for "nkview" which adds a shadow to "nkey". 115
35 The mixed-form view specification for subview "numkey". 116
36 The mixed-form view specification for subview "dispad". 117
37 The interface of BBE audio facility 118

1 Introduction

Contemporary computing is evolving from batch-based applications to highly interactive, graphical applications. The use of graphical user interfaces has spread through almost all modern applications. However, due to several special characteristics, the design and implementation of user interface software is difficult and time-consuming. A study by Myers [19] shows that for application systems with highly interactive, graphical interfaces, the development time for the user interface averages about 50 percent. Over the years, many approaches have been proposed to reduce this development cost. Toolkits, user interface management systems (UIMS) and interface builders have all emerged as viable tools to ease the task of constructing graphical user interfaces [8]. Major progress has been made.

It is becoming generally accepted that many of the difficulties encountered in developing user interface software result from insufficient support of the underlying programming language. To date, most user interface software is still written using conventional languages designed for writing text-based applications. Conventional languages lack many features that are essential to support interactive computing, such as graphics, rapid prototyping support and effective representations for programs. Conventional languages contain only textual constructs, which make it difficult to specify the graphical aspects of the user interface. This accounts for the emergence of various graphical toolkits that are used to supplement the textual languages. A more fundamental problem with conventional languages is that their foundations are typically built on the imperative programming paradigm. Imperative programming demands that the programmer

be completely in charge of the management of the control flow. This is appropriate in batch applications in which control flow is centralized and fairly linear. However, due to the highly interactive nature of user interfaces, the control flow in these applications tends to be distributed, highly nonlinear and unpredictable. As the functionality of interactive applications increases, the potential combinations of operations and dialogues increases nonlinearly. The complexity of the control flow soon increases beyond the ability of the programmer to manage. As the problems are rooted from the paradigm upon which conventional languages are built, it is anticipated that a significant solution to these problems can only be obtained from a revolutionary approach, rather than from evolutionary improvement of conventional languages.

1.1 Motivation for the Mixed-form Language

In this section, we first enumerate several key problems that account for the difficulties in user interface development. The necessity of new user interface programming languages is discussed. Comparisons between textual languages and visual languages, imperative languages and declarative languages are presented in the last section.

1.1.1 Why User Interfaces Are Hard to Develop

A number of reasons have been identified by several researchers [18] explaining why user interface software is hard to develop.

- Iterative design: It is difficult to thoroughly understand the tasks of the end users in designing interactive software. The developer usually has to resort to empirical testing to determine effectiveness of the interface. Considerable redesign following testing is essential to good user interfaces. This iterative design requires the separation of the user interface part from the functional part of the software.

- Difficult to get desired graphical appearance: Using conventional languages and supplementary graphical toolkits to achieve the desired graphical appearance of the interface is a challenging task. The developers have little idea about what the interface really looks like when they use textual languages to specify graphical views.

- Distributed, nonlinear control flow: Highly interactive user interfaces have the characteristic that the user is in control of the action sequence. The user can initiate an arbitrary interaction at any time, unlike in the imperative model where the program dictates when to receive input and what type of input it should be. It is not easy to use conventional languages to construct software with this kind of distributed, nonlinear control flow.

- Too many complex constraints: In interactive software, events, feedback and the data states of the application are tightly coupled by a large number of complex constraints. Using conventional languages, the programmer has to explicitly program these constraints. This leads to messy and error-prone programs. Ideally, these constraints should be maintained automatically by the system after being specified by the programmer using some high-level languages.

1.1.2 Why Build Support into the Language

Many researchers have realized that the problems encountered in constructing user interfaces are mainly due to insufficient language support [9]. For example, most currently available languages only have text-based I/O primitives: read or write a string, and no graphical I/O primitives are supported. To address this problem, a large number of toolkits have been developed, eg, X, Interviews. The idea is: use a large external library to handle the graphical layout of the interface. These toolkits typically provide little help to achieve desired graphical effects and are usually very hard to use. More critically, most toolkits don't tackle the crucial problems in maintaining data consistency and constraints.

Several kinds of interface builders have emerged to aid in GUI development [10]. With interface builders, the interface part and the functional part of the application are completely separated. The developer can construct the actual views of the interface through direct manipulation. This avoids the problems of using textual constructs and absolute coordinates to specify graphics. However, since the interface part and the functional part are sharply separated, extensive low-level programming is required to implement their interaction. This results in difficult development and maintenance.

A language augmented by built-in graphical primitives and automatic constraint maintaining mechanism can eliminate the root of these problems.

1.1.3 Textual Languages vs. Visual Languages

In most current systems, the graphics of the interface is specified using text. However, specifying two-dimensional graphical entities using one-dimensional notion (such as text) imposes considerable inconvenience on programmers. The significant gap between the programmer's intention and the representation used to carry out that intention easily leads to errors. The graphical part of the interface can be much more easily specified in some graphical notation.

In visual languages, views can be directly defined using graphical primitives. Such primitives can be easily manipulated to comprise the desired graphics. However, user interfaces are not merely graphical views. Their highly interactive nature must be backed up by the functional part of the interfaces. Textual languages are more appropriate to program the functional part of user interfaces. For example, it is easy to denote variables and parameters in textual languages, which seems to be an unsolvable problem in visual languages. Ideally, a language should consist of both graphical constructs and textual constructs which can be used respectively to specify the graphical part and the functional part of user interfaces.

1.1.4 Imperative Languages vs. Declarative Languages

Most conventional languages are based on the imperative paradigm, in which control flow is completely managed by the programmer. In highly interactive interface software, the control flow is determined by the user not the program. It is not easy to implement this style of interaction using imperative languages,

because all the complex constraints have to be maintained through explicit programming. In declarative languages, it is only necessary to specify the results we want. How to achieve the results is left to the underlying mechanism supported by the system. Thus, we may use declarative languages more easily to implement the interactive aspects of user interfaces. The declarative paradigm is also appropriate to visual languages, because it eliminates the problem of tangled control flow and variable naming.

1.2 The Mixed-form Language

This thesis is based on the Clock system - a functional programming environment for user interface construction. For a detailed description of the Clock system, please refer to [7].

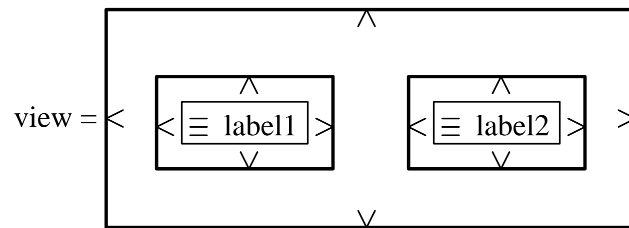
The Clock language is a purely functional language with no side effects. It only has textual primitives. Thus, graphical views of the user interface can only be specified textually. This thesis extends the Clock view language by incorporating a set of appropriate graphical primitives into it. With this extended language, the Clock programmer can choose either form of primitives where it is appropriate in developing user interfaces. Figure 1 shows a piece of mixed-form Clock program and the picture it renders.

The purpose of this thesis is to demonstrate that a functional style language comprised of both textual and graphical primitives is a useful alternative to the conventional languages in user interface development.

As the goal is to maximize the expressive power, we identify a set of appro-

label1 = Text "Ok"

label2 = Text "Cancel"



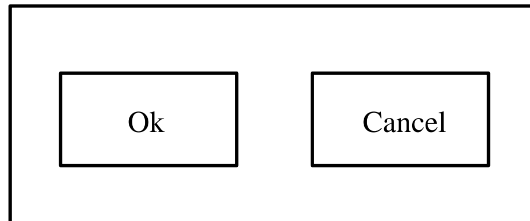
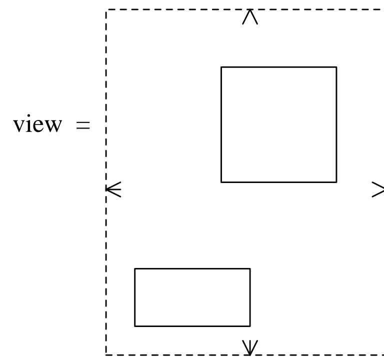


Figure 1: An example of mixed-form Clock program. The display view function is specified with both textual and graphical primitives. The picture generated by this mixed-form specification is also illustrated. The four small magnets attached to each box are used to specify geometric constraints.

priate graphical primitives that can be mapped to the textual primitives in the original language. The purpose of incorporating graphical primitives into the language is to allow graphical specifications of component views of an interface and their spatial relations. A prototype graphical editor has been implemented to let the developer directly draw graphical primitives and arrange their spatial relations to achieve the graphics of the interface. The editor then translates the visual specification into the corresponding textual representation which can be either fed into the Clock compiler for executable code generation or stored in a text file for further use. For compilation purpose, mixed-form programs need to be translated into an intermediate form with uniform primitives. In our case, the original language assumes this responsibility. It serves as an assembly language to the mixed-form language. Figure 2 shows a graphical specification and its corresponding textual representation.

Our approach offers a number of advantages. First, the developer is allowed to specify the view part of the interface intuitively using graphical representation rather than being forced to use textual representation. Thus, the desired appearance of the interface can be more easily achieved. More importantly, in our approach, interactive views are treated as first class values. Therefore, we can insert a picture wherever a value is allowed. On the other hand, text can also be inserted into pictures as long as the computation of the text yields a picture. In this way, a fluid mixture of textual and graphical primitives can be achieved. The problems caused by the sharp separation between the graphical and functional part of the interface in some other approaches can be avoided. We use a language based on the functional paradigm. In functional languages,



```
view =   Views [
              At  ( x 10, y 10) (LeftStretching 50 ,  BottomStretching 30)  Box (noView) ,
              At  ( x 40, y 60) (LeftStretching 80 ,  BottomStretching 100)  Box (noView)
            ] .
```

Figure 2: A graphical specification and its corresponding textual representation.

there is no notion of variables and control flows. Thus, the tangled control flow of imperative visual languages can be avoided. In our approach, pictures make no reference to external variables. The problem of representing variables in visual languages is therefore eliminated.

This thesis attempts to demonstrate that a functional language with mixed-form constructs solves the problems of a sharp separation. The major contributions of the thesis are: identifying a set of appropriate graphical primitives for view specifications, incorporating them into the Clock view language and developing a prototype graphical editor which supports the editing of mixed-form programs and the translation of graphical primitives to their textual equivalents. The translation is based on the formal semantics of the mixed-form language.

1.3 Thesis Overview

The following is an overview of the contents of this thesis.

- Chapter 2: Related work. The idea proposed in this thesis is inspired by the analysis of advantages and shortcomings of a variety of related research work. Chapter two briefly describes several related research projects and systems, and shows how the ideas proposed and demonstrated in those projects contribute to the emergence of our approach.

- Chapter 3: Clock system and the textual view language. This chapter first introduces the Clock system and the Clock program development methodology. The description of the Clock textual view language sets the context

for the work in this thesis.

- Chapter 4: Mixed-form visual-textual programming. This chapter discusses motivations for mixing textual and graphical primitives in a functional language for user interface development. The mixed-form Clock view language is introduced in an informal way.

- Chapter 5: Syntax and semantics of the mixed-form view language. A formal syntax and semantics of the mixed-form Clock view language is presented. The semantics description is based on a correspondence mapping from graphical primitives to their textual equivalents.

- Chapter 6: Mixed-form view specification examples. A series of user interface examples are specified using the mixed-form Clock view language. Evaluation of the language in terms of usefulness is carried out.

The final chapter (chapter 7) summarizes the thesis and presents conclusions. A discussion of directions for future research is also presented.

2 Related work

The idea of mixing textual and graphical programming in the development of user interfaces is based on close observation of several existing systems. This chapter enumerates a variety of research projects which served as motivation and basis for the work in this thesis. For each related research project, an overview of the project and its relevance to the work in this thesis will be presented.

Related work fall into three categories: purely textual, purely visual and mixed textual/visual approaches. This chapter covers nine existing systems that are representatives in the three categories.

2.1 Purely Textual Approaches

Purely textual languages usually need to depend on some external graphical libraries in programming graphical user interfaces. The developer has to specify the graphics in terms of textual primitives. Lots of effort and iterations are needed to achieve the desired result.

2.1.1 Xt toolkit

The popular *Xt toolkit* for the X window system [22] is a typical example of systems taking the purely textual approach.

The Xt toolkit represents an application programming interface (API) and a run-time library. A set of primitive widgets and composition widgets are provided. Primitive widgets are used to render predefined user interface components

while composition widgets are used to manage the geometric layout of primitive widgets. The Xt Intrinsics provides basic abstractions that are necessary to organize the graphical input and output of an application. With the toolkit, user interfaces are constructed and manipulated in terms of graphical widgets. This remedies the defect that most user interface development languages (usually C or C++) lack graphical I/O facilities. In addition, the Xt toolkit provides a quick means for prototyping user interfaces.

However, the Xt toolkit suffers from several major problems. Programming in X requires the use of absolute coordinates. This makes it hard to get the desired graphics of the interface. The developer has to specify graphical components and their layout arrangement in a textual language. It is difficult for the developer to picture what the interface really looks like when writing textual code. In X applications, management of interaction is achieved through a convoluted control structure. An event dispatcher is set up to invoke a set of predefined "call-back" procedures which in turn update the data state of the application. The maintenance of data consistency is achieved totally through low-level, explicit programming. Furthermore, the code that handles graphical layout of the interface is interwoven with the code that handles functionality of the application. These problems make the program difficult to develop and to maintain and work against the goals of rapid prototyping and iterative refinement.

2.1.2 InterViews

The *InterViews* [14] system is a graphical user interface toolkit developed at Stanford. The major contributions of InterViews are several flexible composition

mechanisms and a uniform protocol to define the behavior of the primitive user interface objects. These facilitate the construction of customizable and dynamically extensible user interfaces. Above all, InterViews provides a novel approach to separating the interface from the application while maintaining a tight connection between the two.

Separating user interfaces from application programs is important in many aspects of user interface construction. However, the highly interactive nature of interface software requires tight coupling of the two parts. Toolkits enable separation of interface code and application code at the level of individual components. But most toolkits do not enforce the use of a uniform separation. In InterViews, a base class called *glyph* is used to specify a common protocol for communication among objects. This protocol allows uniform treatment of interface objects and flexible composition of these objects in building up the complete interface. In InterViews, a *monoglyph* is responsible for specifying the behavior of an object, while the inner *glyph* is responsible for specifying the appearance of the object. Thus, the common protocol fluidly glues the view part and the behavior part of interface components.

In InterViews, *interactive objects*, which implement the interface, and *abstract objects*, which encapsulate the data underlying the interface, are distinguished as *views* and *subjects*. A *view* in InterViews corresponds to a combination of a controller and a view in the MVC model [13]. Views are usually implemented by composing primitive objects, while subjects are typically derived from the *subject class*. A subject maintains a list of its views. An *Update* operation is defined on views to refresh their appearance in response to changes in the subject. Invoca-

tion of the *Calling Notify* method on a subject caused by any data change will in turn call the *Update* method on its views.

InterViews takes advantage of the object-oriented paradigm to allow tight coupling between the views and behavior of interface components while separating them from each other. The scaling up of interface components to the complete interface is made more flexible and more fluid. InterViews separates the interface part from the application part in such a way that necessary connection between the two can be easily maintained. However, with InterViews, the developers still have to specify graphics in terms of textual primitives, which is not an easy task for the programmers unless they use the InterViews interface builder.

2.2 Visual Languages

In visual programming languages, syntax and semantics are defined and reflected by the appearance of the program, which is specified by the combination of graphical elements and their spatial configuration. We concern ourselves with visual languages because graphical primitives are easier and more intuitive to use in specifying pictures. However, the following two visual programming systems reveal some inherent problems with purely visual languages.

2.2.1 Pict/D

Pict/D [4] is a purely visual programming environment. In Pict/D, the programmer never touches the keyboard. Instead, they can draw the program using

pointing devices such as a joystick. A set of high level graphical entities are made available to users as atoms which they may manipulate in the programming environment. The prototype of Pict/D system allows the programmer to compose simple numeric calculation programs.

Pict/D is an icon-based system in which all keywords have been replaced by iconic representations. The user can program an icon, edit it, or run its associated program. In Pict/D system, program and procedure names, parameter passing modes, data structures, and variables are all represented by icons of various shapes and colors. The control structures such as repetition and selection are represented by colored, directed paths that can be actually seen. Programming in Pict/D involves four major steps:

- Select appropriate icons that visually represent the data structures and variables needed in the program.

- Draw the desired algorithm as a logically structured picture-like flow chart.

- Watch the animated program execution and see the results being generated.

- Errors can be seen directly during the program execution.

Pict/D is a good demonstration of the problems associated with imperative visual languages. Since variables are denoted by colors in Pict/D, the number of variables in each program or subprogram is restricted to four. This restriction severely limits the functionality of Pict/D programs. Pict/D is based on a flow chart metaphor. As the size of the program increases, the flow chart easily becomes too big and messy to manage. Thus, Pict/D is bound to be able to handle

only small applications.

The approach presented in this thesis avoids the above problems by employing the functional paradigm. In a functional language, there is no notion of control flow or global variables. Thus, the limitations imposed by the use of pictures in imperative languages are totally eliminated. Meanwhile, a visual language fits well with the functional paradigm.

2.2.2 ThingLab

ThingLab [1] is another example of purely visual programming environment. It introduces the idea of graphically specified constraints.

In most constraint-based systems, constraints are typically defined using an underlying implementation language. Therefore, in order to specify a new constraint, the user must leave the graphical environment and write some textual code. ThingLab is a constraint-based simulation system in which the developer can define constraints graphically.

For simulation in a certain domain, a set of building blocks are first constructed using ThingLab. Then, other users may scale up these building blocks to explore particular simulations. In ThingLab, a constraint is composed of a predicate and several methods. The predicate is used to test whether the constraint is satisfied. The attached methods are used to modify states of the objects involved in the constraint. Once a predicate is defined, the system will automatically derive the necessary methods for satisfying the constraint. In ThingLab, a kind of network is employed to define constraints. The networks look just like

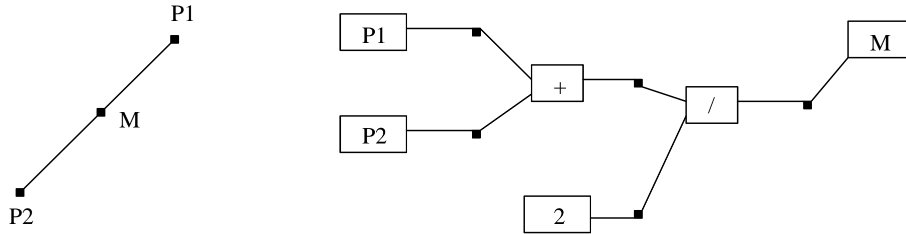


Figure 3: A graphical definition of a MidPointLine. The definition includes labels for the line's endpoints and midpoint, P1, P2 and M. The network on the right constraint the value of the midpoint to be equal to the average of the values of the two endpoints of the line.

digital circuits composed of logical gates. Variables of the involved objects are denoted as small labeled boxes. Such labeled boxes are connected to the inputs and outputs of certain operators to specify the relationships among them. Figure 3 [1] shows a graphical specification of a MidPointLine in ThingLab.

ThingLab presents an intuitive way to graphically specify constraints. It demonstrates that, in a visual programming environment, graphical constructs are easier and more intuitive to use than textual constructs in the specifying of constraints. In this thesis, we will explore a way to visually specify the sizing and positioning constraints of graphical components of user interfaces.

2.3 Hybrid Approaches

The hybrid textual/visual approaches are meant to somehow combine the merits of both purely textual and visual programming. This section reviews several representatives of such approaches.

2.3.1 NeXT Interface Builder

NeXT Interface Builder (IB) [10] is a system attempting to combine the advantages of visual and textual programming. It allows the developer to specify the graphical part of a user interface in a direct manipulation manner. Then, this graphical part is linked to the functional part written in a traditional textual language.

In IB, programmers can build an interface by simply dragging predefined graphical objects from the palette of NeXTSTEP objects directly into the application they are building. A number of methods are provided to customize these graphical objects. The programmer can also directly manipulate the spatial relationships among the graphical objects to create the desired view of the interface. IB's technique of direct manipulation of programming objects is not limited to predefined objects in NeXTSTEP. The palette can be extended by new objects created by the developer.

IB adheres to the principle of completely separating the user interface part and the functional part of application programs. The functionality of an application is handled by computational objects which are written in Objective C. IB uses a message-passing mechanism to deal with the interaction between the interface part and the computational part. Such connections are made through an object's *outlets*. An object can send certain messages through its outlets. The developer can initialize an outlet to establish a connection by simply dragging a line from the source object to the destination object. Then, IB allows the developer to choose the message to send from a list of action messages which the destination object can respond to. The developer can define the outlet vari-

ables and actions of the computational objects so that they can issue messages to certain interface objects and respond to messages received from some interface objects. In this way, the interface objects and the computational objects are interconnected so that they can communicate with each other.

IB allows the actual view of an interface to be constructed through direct manipulation. This avoids the problems of using one-dimensional textual constructs to specify two-dimensional graphical components and their composition. Therefore, the developer can easily attain the desired graphics. This graphical part is then connected to the computational part through message passing. There is a sharp separation between the two. This strict cut between the functional part and the graphical part results in a low-bandwidth between them as well as communication overhead. More critically, implementation of the communication between the two parts still relies on explicit low-level programming, which is not a trivial task. It is the programmer's duty to manage message passing to ensure data consistency and to satisfy various constraints. The problems caused by imperative programming in user interface development are therefore not solved.

2.3.2 HyperCard

HyperCard [5] is another system attempting to combine the benefits of visual and textual programming. The graphical part of a HyperCard program is drawings on cards or backgrounds, as well as attached buttons and text fields with a variety of appearances and styles. A textual script language, *Hypertalk*, is equipped to define the interactive behavior of the graphical part.

A HyperCard program is called a *stack*, which is composed of *cards* and *backgrounds*. Pictures can be drawn on cards and backgrounds using an integrated painting facility. A background behaves very much like a card except that the items on a background can be shared by multiple cards. In addition, cards and backgrounds can have *buttons* and *text fields*. A button is simply a location that can be clicked to invoke some behavior. A text field is a rectangular area in which text can be edited. A variety of presentation styles for fields are available. A field is stored in each individual card, even if the field is drawn on the background. HyperCard allows a field to be laid out on a background with multiple cards sharing that background, each with its own information. A HyperCard structure diagram is shown in Figure 4.

Hypertalk is an integrated script language meant to define the behavior of the user interface. The language is structured around the sending of events. Scripts are attached to buttons, fields, cards, backgrounds, stacks and HyperCard itself. Each script specifies what to do when a particular event occurs. Anything that can be done interactively through the HyperCard interface will have a corresponding Hypertalk event that can perform the same function in a similar manner. Events can be pre-defined, such as "mouseUp", or user-defined.

An interesting feature of HyperCard is that some degree of inheritance is provided. On the graphics side, each card can inherit the pictures, buttons and fields of its background. On the behavior side, handling of an event is controlled by an inheritance path shown in Figure 5. An event is forwarded on up the chain, searching for a script that can handle the event. For example, when a field script does not know how to handle a given event, the event is forwarded to the current

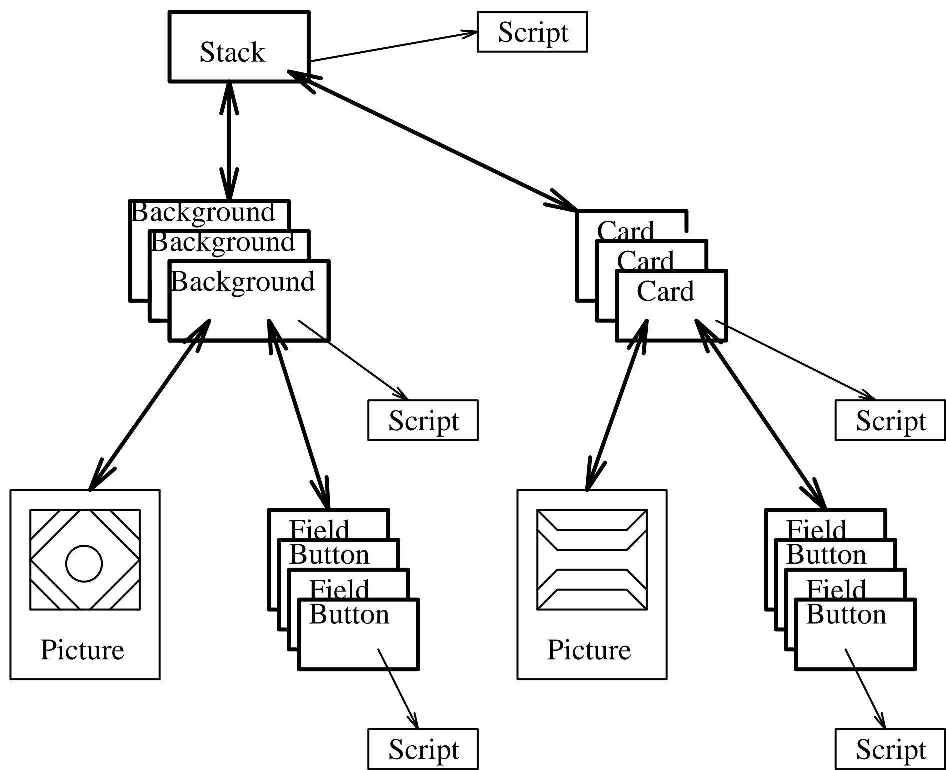


Figure 4: The structure diagram of HyperCard.

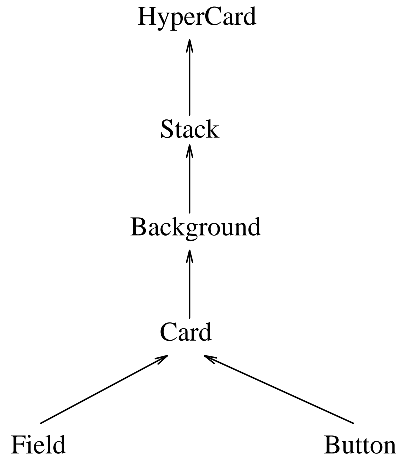


Figure 5: An illustration of the HyperCard inheritance chain.

card and so on. This inheritance structure allows the sharing of event handlers across various HyperCard pieces. However, this inheritance chain is fixed. It is not programmable as inheritance hierarchies in normal object-oriented programs.

HyperCard is essentially similar to interface builders in that pictures are directly manipulated and behavior is defined by attached code, except that the textual language is of higher level. In addition, HyperCard provides inheritance facilities to enable the sharing of both graphics and event handling procedures, which offers significant convenience and power to the developer. Nevertheless, the inherent problems with interface builders - the sharp gap between the graphical part and the functional part of user interfaces, is not addressed.

2.3.3 GVL: Graphical View Language

In order to achieve the goal of completely separating the user interface from the application program, the Conceptual View Model [6] was introduced. In the

Conceptual View Model, the application program deals only with the data structure and the algorithms to manipulate the data structure. It cares nothing about the output. The output views are defined using a special-purpose specification language *GVL* [2] to map the state of the data structure to conceptual output views. The mapping from application program data structure to displays is performed in two parts. First, a mapping specification that can be drawn in GVL translates the current state of the data structure into a set of basic display primitives. Then, a set of layout rules are applied to determine the sizes and positions of these primitives in absolute screen coordinates. GVL is a functional graphical language. It uses a limited number of primitives to achieve the power to specify complex display views. It is shown that several inherent problems associated with imperative graphical languages can be avoided by using the functional paradigm.

The work in this thesis is strongly stimulated by some of the ideas initiated in GVL.

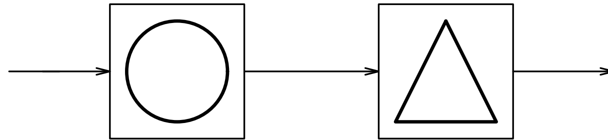
2.3.4 Pic++

Pic++ [12] is a programming language extension designed to add pictures as a base data-type in a textual language. With Pic++, pictures can be embedded within the text of C++ source code. A simple functional sub-language for directly manipulating pictures in textual code is designed and partially implemented on the Macintosh. Pic++ is a research project that actually incorporates text and graphics in program code. It advances one step further towards a harmonic and fluid mixture of text and graphics.

Realizing that adding strings as a base data-type in programming languages (e.g. Turing) has proved to make string manipulation more intuitive and easier, the author of Pic++ argues that picture handling could be as easy and intuitive as string handling if pictures are incorporated into a programming language also as a base data-type. Traditionally, graphics have been added at the geometric level which means pictures are built from pixels, lines, circles and other geometric objects. The designer of Pic++ believes that this is not the correct abstraction level for picture drawing. Loading completely predrawn pictures using a painting program is easier than drawing pictures with code. The Pic++ approach is to copy complete pictures from a painting editor and manipulate these pictures in the program as picture constants. A simple functional style sub-language is designed to handle picture manipulations. Borrowed from *TEX*, reference points are used to arrange spatial relations of pictures and bounding boxes are used to specify the sizing constraints. A set of functions are provided to support picture operation. In Pic++, it is expected that most pictures will be drawn before program execution using a paint editor. However, it does offer some functions to draw basic geometric objects such as lines and circles. The "plus" operator is overloaded to create an ordered list of pictures. The "Group" function is designed to achieve picture hierarchies. With this function, a set of component pictures can be grouped to form a bigger picture for further manipulation.

Pic++ supports the embedding of pictures into a textual program. The sharp division between text and graphics is somewhat blurred. However, Pic++ treats pictures as simple constants, though limited picture manipulation using textual code is possible. The degree of the mixture of pictures and text is not satisfactory.

A = Circle + Triangle



Group(A) + Pentagon

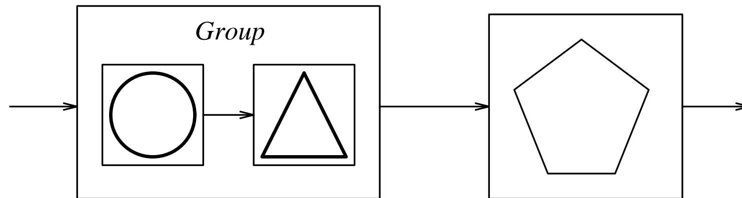


Figure 6: The overloaded plus operator and the Group function in Pic++.

2.3.5 VisualWorks

VisualWorks [11] is a fully object-oriented system for constructing graphical user interfaces, using Smalltalk as the scripting language. VisualWorks is based on the model-view-controller[13] model. It provides a visual means of constructing views from a standard set of components. Typical controller behavior is provided as well.

With VisualWorks, you begin building an application by using the Painter tool to place visual components on a canvas. A variety of tools are provided for sizing, positioning and formatting each component. Then, component properties may be set, such as labeling, bordering and naming. Every component has

an *ID* property that enables the programmer to name the component for reference within the application code. A visualWorks application has two parts: a *user interface* and a *model*. The model can be subdivided into a *domain model*, which stores and processes data, and an *application model*, which coordinates the behavior of the user interface. The model is usually a subclass of the *ApplicationModel* class so that it inherits behavior for communicating with interface components. Views are specified as methods of the application model class. You create a model by giving the application class a name. Each canvas of the application is stored in the form of a method that contains the specifications for the canvas and its components. The system can also generate skeleton code to support the functionality of each component.

VisualWorks is essentially another kind of interface builder. It distinguishes itself from common interface builders in that it stores view specifications as methods of the application class. In this way, view specifications are treated as a more integral part of the application class. The communication overhead between the functional and the presentation part of user interfaces is reduced. However, in VisualWorks, graphics and text are still separated at the individual component level and the maintenance of the constraints between the two parts still relies on explicit programming.

2.4 Conclusion

In understanding and analyzing various user interface development systems based on either purely textual, purely visual or hybrid approaches, the following points become clear:

- The graphical part of user interfaces is easier to specify using graphical constructs than using textual code. A textual language is more appropriate for specifying the functional part of the interface.

- Separating the graphical part and the functional part is vital in user interface development. However, maintaining a seamless connection between these two parts is necessary to ensure effective communication. Such a connection should depend on system support rather than low-level explicit programming.

- As graphical primitives and textual code are treated as essentially different things in some traditional hybrid approaches, a fluid mixture of the two is almost impossible. To address this problem, a framework is necessary in which textual and graphical primitives can be treated equally.

Based on realization of the problems identified from the above systems, this thesis argues that mixing textual primitives and graphical primitives in a functional programming framework provides a potential solution.

3 Clock System and the Textual View Language

In order to experiment with our ideas in mixed-from visual/textual programming, we started with the existing Clock language [7], a purely textual language for the development of graphical user interfaces. This chapter reviews the Clock language; Chapter 4 will then show how Clock was extended to support mixed-form visual/textual programming.

Clock is a system designed to support the development of highly interactive, graphical user interfaces. The major design goals of Clock are to support rapid prototyping, clean structuring, incremental development and component reuse, all of which have been identified as desired features in tools for user interface development [7]. The Clock language aims to realize these features by combining the advantages of constraint programming, functional programming and structural model of such systems such as MVC[13] and PAC[3].

The Clock language is a purely declarative language which can be divided into two levels. The first level is a visual architecture language used to specify the organization of an interactive system as a tree of components. The second level is a purely functional language used to specify the operations of the components themselves. To create an interactive application, the developer first decomposes the application into components, specifies their interfaces, and then organizes them as a tree structure using the Clock architecture language. The view and behavior of each individual component are then defined using the functional language.

In general, a user interface can be decomposed into compositional view com-

ponents. Each individual component has a view function which is responsible for displaying the component view it represents. In the current Clock, views are specified using a textual view language. The textual view language consists of positioning and sizing primitives, a set of display primitives and some abstraction primitives. The Clock textual view language is simple yet powerful in specifying display views. It allows the programmer to scale up simple display primitives and component views to achieve complex graphics.

The current Clock view language has only textual primitives. Thus, it has the inherent problems of using textual languages to define pictures. For example, the programmer may have to develop a picture in terms of concrete coordinates which can be unintuitive and tends to be error prone. It would be preferable for the programmer to simply draw the picture using graphical primitives. On the other hand, textual primitives are more suitable than graphical primitives for specifying certain aspects of the program. For instance, it is difficult, if not impossible, to denote numerical parameters graphically. Hence, seamlessly integrating textual and graphical primitives into a single language is a promising approach to overcome these problems.

In this chapter, a description of Clock system is given first. An interactive calculator program will be used as an example to illustrate development of user interfaces in Clock. A later section will be devoted to a description of the Clock textual view language to set the context for the following chapters.

3.1 The Clock Hierarchical Architecture

The Clock model describes how programs are broken down into components and how these components are organized in the Clock system. Clock architectures are organized as trees of components. The tree architectures do not enforce a strict separation of application and user interface. In Clock, the root of the architecture tree corresponds to the pure application program and the leaves of the tree correspond to the most basic components of the user interface. The nodes inbetween form a smooth progression from the application to the user interface. For example, Figure 7 shows the interactive calculator application and its Clock architecture tree. The root node of the tree "Calculator" represents the application program. The inbetween nodes "NumKey" and "OperKey" map to the group of numerical keys and the group of operation keys. The leaf nodes "Display", "EqualKey" and "CaButton" respectively correspond to the number display screen, the equal sign button and the "clear all" button, which are concrete compositional components of the interface.

A Clock architecture tree is composed of two kinds of components, *event handlers* and *request handlers*. An event handler defines a functional component of the user interface in terms of its response to events, its appearance on the screen, its data consistency constraints and its initial state. In other words, event handlers are responsible for dealing with user inputs, for presenting display views, for maintaining consistent internal states, and for performing initialization. Event handlers are stateless. They depend on request handlers to retrieve data. Request handlers are used to hold data structures of the architecture. Each request handler implements an abstract data type which can be used by event handlers.

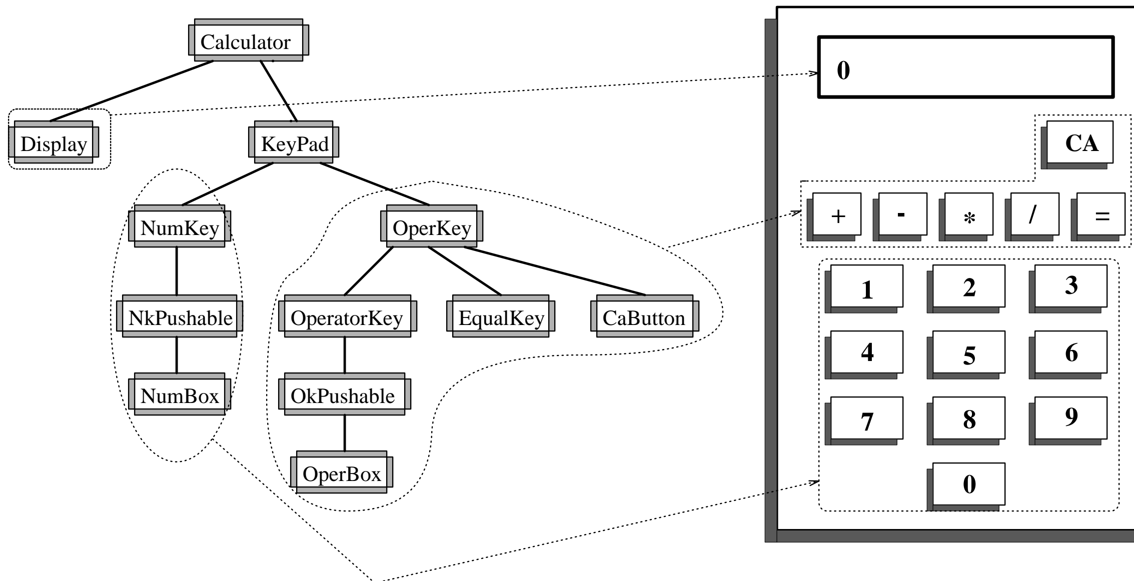


Figure 7: A calculator application and its architecture tree. The calculator is composed of a display screen and a key pad. The key pad is made up of numerical keys, operator and functional keys.

As an example, an event handler and an attached request handler to implement a push button is shown in Figure 8.

Request handlers may respond to a set of explicitly defined *requests* for data, or to *updates* to refresh their own data. Requests and updates correspond to *accessors* and *mutators* in the Object-Oriented terminology.

A Clock program is organized as a hierarchical tree of components. Each tree node is made up of an event handler and a set of request handlers attached to it. A detailed architecture tree diagram for the interactive calculator program is shown in Figure 9.

v The calculator's main screen is composed of two components: "Display" and "KeyPad", which are subviews of the "Calculator" event handler at the root

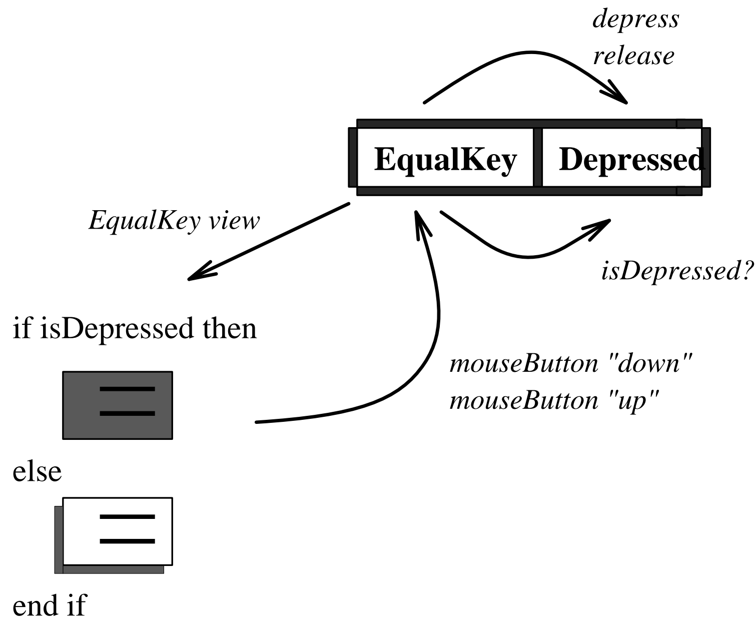


Figure 8: Event handler "EqualKey" and request handler "Depressed".

of the architecture tree. The "Display" event handler is used to display the operands and the results. The "KeyPad" event handler is used to handle user's inputs.

The "KeyPad" is made up of a number of numerical keys ("NumKey") and operator keys ("OperKey"). The "NumKey" allows the user to enter a digit by pressing the corresponding button. The "NumKey" generates one "NkPushable" instance for each digit from "0" to "9". The "NumKey" event handler receives input from its ten children whenever one of them is pressed. Then, it issues an event to indicate which of its children is pressed so that the relevant request handler can record the user's input. The "OperKey" is made up of a set of operator keys and two functional keys "EqualKey" and "CaButton". The "OperatorKey" event handler is similar to the "NumKey" in terms of their children and their

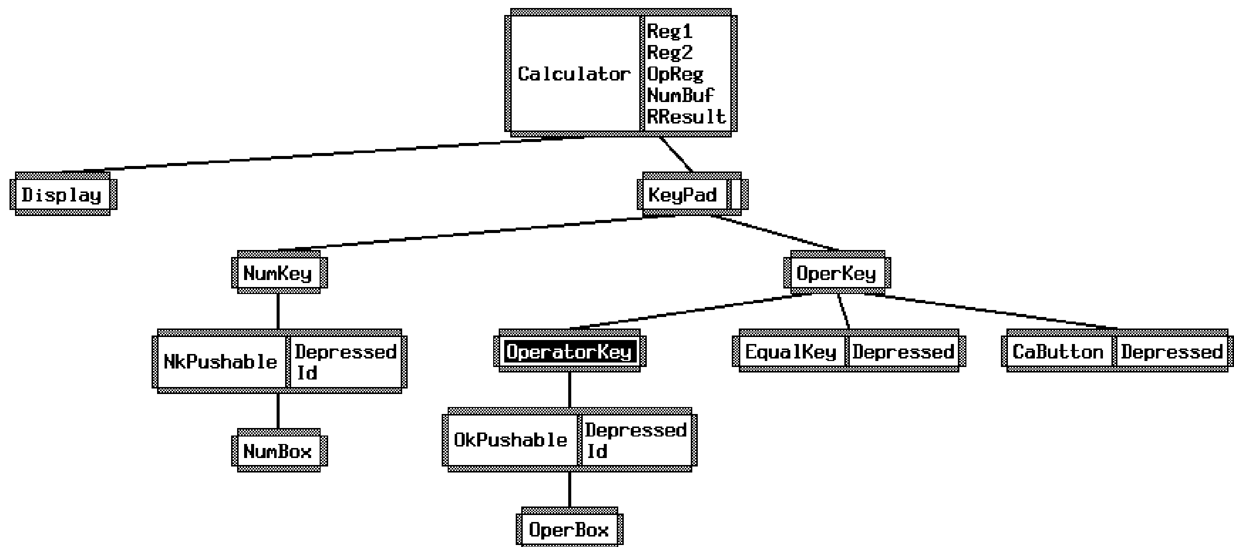


Figure 9: An architecture tree for the calculator program showing request handlers. For each tree node, event handler is drawn at the left and request handlers at the right.

operation.

Architecture components interact via the mechanism of passing events. There are three kinds of events for component communication in Clock. Event handlers issue *requests* to retrieve data from some request handlers and issue *updates* to modify the data in certain request handlers. *Inputs* may come from mouse or keyboard input or from updates generated by event handlers at lower levels in the architecture tree. A request issued by an event handler is passed up the tree architecture to the first request handler who takes that kind of request. Similarly, an update generated by an event handler is passed up the tree to the first request handler that is able to accept this kind of update, or to the first event handler who takes this kind of update as input and may in turn trigger other updates. In general, event handlers actively send requests and updates. Requests are used to access data represented by request handlers at the same level or higher in the architecture tree. Updates are used to inform some events taking place or some required changes of state. Request handlers passively answer requests or receive updates to change the data they represent. A graphical representation of the architectural detail [17] for the "NkPushable" component is shown in Figure 10.

This request and update mechanism facilitates easy modification. A component in the architecture tree basically knows what is above it in the tree, but needs not to know what is below it. Thus, any subtree of an architecture can be replaced without having to modify other parts of the architecture above it. This mechanism reduces the amount of dependencies among components and organizes them in a one-way style (from bottom to top).

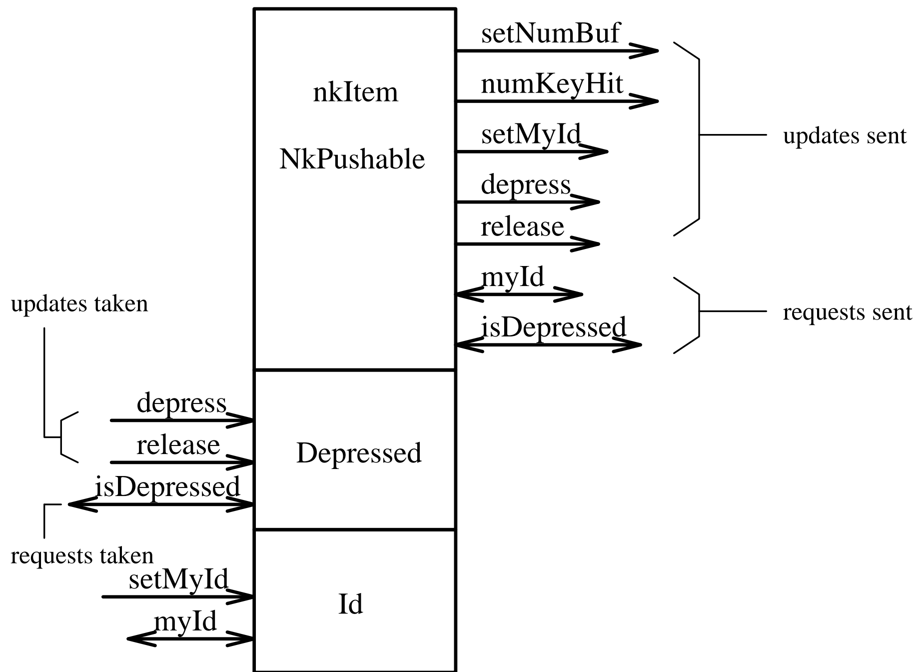


Figure 10: Architectural details of the "NkPushable" component. In this graphical representation, a line with one arrow denotes a update or input. A line with two arrows denotes a request. Single arrow lines on the right side of a component represent updates issued by the component and single arrowed lines on the left hand represent updates or inputs taken by the component. Double arrowed lines on the right side represent requests issued by the component and double arrowed lines on the left side represent requests handled by the component.

As an example, Figure 8 shows an event handler "EqualKey" and a request handler "Depressed" associated with it. "EqualKey" is responsible for displaying a picture of an equal sign button on the screen, and for taking mouse inputs to the button. Any user action performed within this part of the display is reported directly to this event handler. For instance, if the user depresses the mouse button on any part of the equal sign view, the input event **mouseButton "down"** is reported to the "EqualKey" event handler. If this input event affects the display of the button such as turning it into reverse video, the event handler is given the opportunity to update the display. In this example, if the user presses the mouse on the equal sign button, the button turns into reverse video (white-on-black). When the mouse button is released, the equal sign button returns to its normal display. The request handler "Depressed" is used to record whether the "EqualKey" is pressed or not. The request *isDepressed* is sent to the request handler "Depressed" in computing the view of "EqualKey" to determine whether it should be displayed in reverse or normal video. The updates *depress* and *release* are sent to "Depressed" by event handler "EqualKey" when the user clicks mouse button on the "EqualKey" picture.

Figure 11 shows the interactive calculator program running under the Clock system.

3.2 Component Specification

In Clock, the components of an architecture tree are instances of some *component classes*. This approach allows predefined classes to be reused in different architectures or in multiple locations of the same architecture. Thus, an architecture

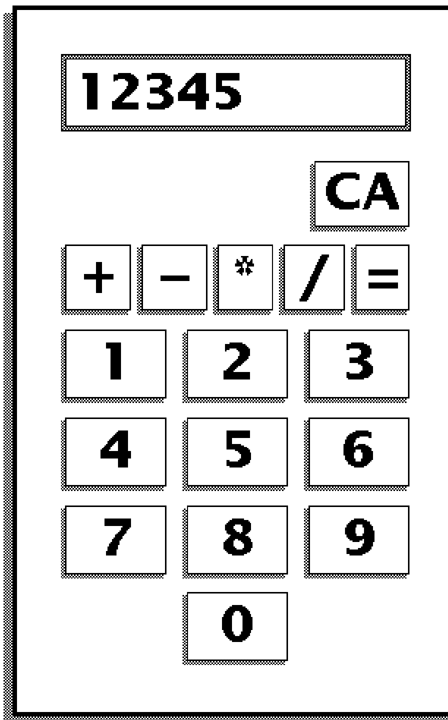


Figure 11: The interactive calculator program running under Clock.

specification consists of definitions of component classes, as well as descriptions of how components are instantiated from these classes to form an architecture.

Classes are specified roughly in two parts: their interface and their behavior. The interface part describes what updates and requests they can take (for request handlers) or generate (for event handlers). The behavior part describes how they react to their inputs. The interface of a component class exclusively defines how instances of that class communicate with other components. Implementation of the class behavior is encapsulated within the class.

3.2.1 Event Handlers

The interface of an event handler consists of four parts: the *subviews* the event handler uses, the *inputs* it can handle, and the *requests* and *updates* it may generate. As shown in Figure 8, event handler "EqualKey" issues *depress* and *release* updates. It also sends *isDepressed* request. The only input it takes is *mouseButton*.

An event handler may construct its views in terms of the views of its children. Views of children components are referred as *subviews* of an event handler. In Figure 12, a subtree of the calculator architecture and the views corresponding to each node are shown.

In Figure 12, event handler "NumBox" simply displays a digit surrounded by a box. The "NkPushable" takes this as its subview and adds a shadow to it. The event handler "NumKey" virtually makes ten instances of its subview "NkPushable" to compose its own view. The request handler "Id" is used to

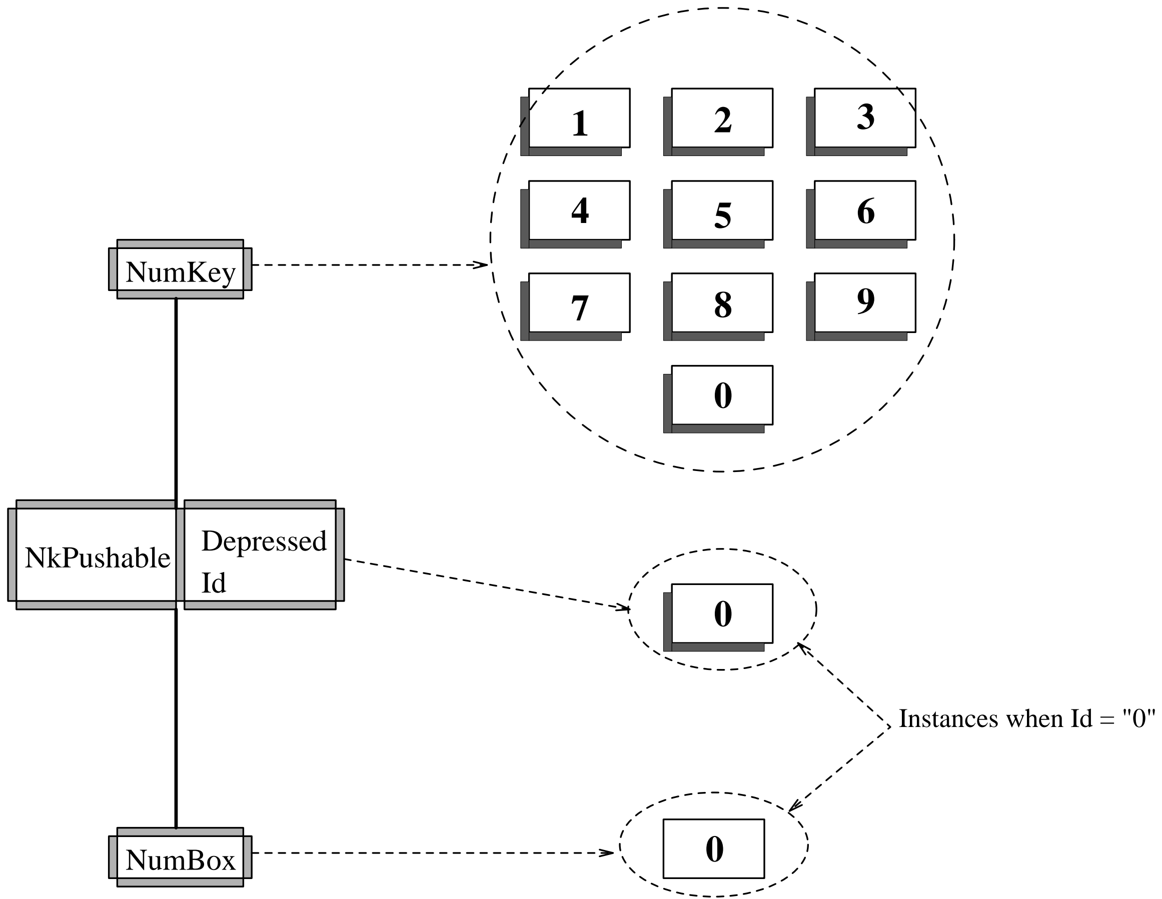


Figure 12: "NumKey" subtree and pictures corresponding to each node. The views for "NkPushable" and "NumBox" are specific for instance when "Id" is "0". However, they are applicable to other Ids.

distinguish among these ten instances.

This view and subview approach allows to scale up views of the most basic user interface components to composite views, and eventually to the whole interface.

Inputs may come from the user's action on the view of the current event handler, or may be passed up from a child event handler. For the sake of reuse, a set of predefined updates are provided in Clock, such as *mouseButton*.

Requests are the requests that may be generated by an event handler in the course of its execution. Requests are used to keep event handlers consistent to data structures represented in some request handlers.

Updates are the updates that may be generated by an event handler in the course of its execution. Through updates, internal data structures can be refreshed to satisfy some constraints.

The description of an event handler's behavior consists of four parts. The *view* function specifies the picture rendered by this event handler on the display. The *event* function describes how the event handler responds to its inputs. The *invariant* function describes how the event handler keeps itself up to date in the presence of changes to other parts of the system. An invariant function issues updates to local request handlers when they become inconsistent with other request handlers. The Clock system automatically determines when invariant functions should be applied. The programmer needs only to specify what consistency constraints should be maintained. The *initially* function describes how the event

handler should be initialized.

The complete class behavior definition of the “NkPushable” class is shown below:

```
mouseButtonUpdt m =
   if streq m "Down" then
            depress
   elsif streq m "Up" then
            all [release, setNumBuf myId, numKeyHit]
   else
            all []
   end if.

invariant = noUpdate.

initially a = setMyId a.

pItem =
   if isDepressed then
            invert (nbButton "")
   else
            nbButton ""
   end if.

view =
   if isDepressed then
            whiteUpperShadow pItem
   else
            greyShadow pItem
   end if.
```

3.2.2 Request Handlers

The interface of a request handler consists of the requests and updates that request handler is able to take. The behavior of a request handler is defined in three parts: the *request* functions describe how requests are answered, the *update* functions state how updates should affect the state of the request handler, and an *initially* function specifies the initial state of the request handler.

For example, the definition for request handler *Depressed* taken from the Clock library is shown below:

```
data State = Depressed | Released.

depressUpdt _ = Depressed.
releaseUpdt _ = Released.

isDepressedReq Depressed = True.
isDepressedReq Released = False.

initially = Released.
```

The data type *State* is a user defined type that has two possible values: *Depressed* and *Released*. An update function is denoted as the name of the update it handles plus "Updt". Therefore, the update function "depressUpdt" handles update *depress*. In this definition, the "depressUpdt" function takes the request handler's state as a parameter and returns a new state. The equation "depressUpdt _ = Depressed." specifies that if the update *depress* is received, the state of the request handler will be set to be *Depressed*, replacing its original state.

Similarly, the equation "releaseUpdt _ = Released." specifies that if the update *release* is received, the state of the request handler will be set to be *Released.* A request function is denoted as the name of the request it handles plus "Req". The "isDepressedReq" request function specifies that if request "isDepressed" is received and the request handler's state is *Depressed*, then return *True*; if the request handler's state is *Released*, return *False*.

3.3 The Clock Textual View Language

The Clock textual view language is designed to specify views displayed by event handlers. The picture rendered by an event handler is defined in its view function. For example, the view function "view = Style Dashed (Box(Text "Hello"))." displays the text "Hello" surrounded by a dashed box.

In Clock programs, a view function definition is made up of a series of simple primitives, such as boxes, lines and text. The location and size of a primitive are specified by the positioning and sizing constraints in the coordinate space that surrounds it (the line primitive is an exception as its location and size are determined by the associated pair of coordinates). In the Clock language, the abstraction mechanism is realized by function definition and evaluation. Component views can be first defined in some functions, and are then evaluated in a bigger context to compose a bigger picture.

The Clock textual view language is simple, yet powerful and flexible enough for programmers to create complex pictures. However, the programmer still has to specify graphics using textual primitives. The inherent problems in specifying

graphics textually are therefore not solved.

In this section, we first discuss the underlying model of view functions in Clock. A detailed description of the Clock textual view language is then given. For a general description of the Clock language, readers are directed to [7].

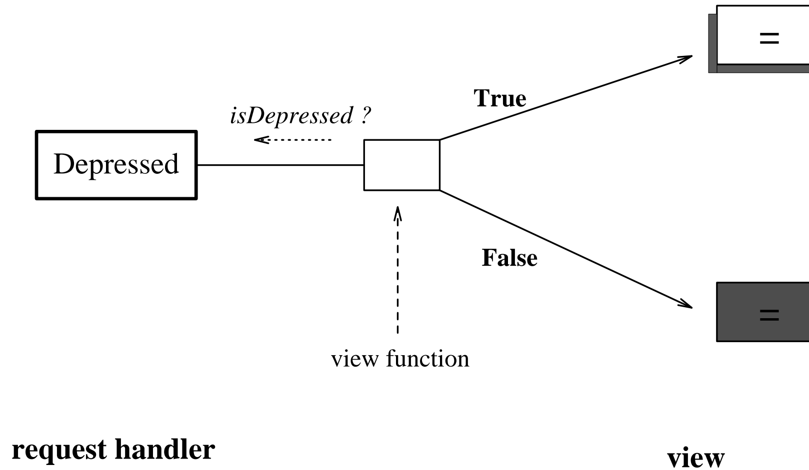
3.3.1 The Underlying View Model

The underlying view model in Clock is called the *Relational View Model* [21]. In Clock, a picture is defined by the view function in an event handler using the textual view language. View functions act as one-way constraints to map system states to pictures. As data are represented in request handlers, display views are considered as functions of the data states of request handlers. A view function simply specifies how to map the data states represented by some request handlers into views. Whenever the states of those request handlers are changed, the corresponding views are automatically updated. The programmer has only to specify how the data state is mapped to a view. When the view is updated is left to the system.

As an example, the view function definition of the event handler "EqualKey" is shown below:

```
view =
   let
            pItem = Font hugeBoldText (
                           invertIfNecessary (Box( padpText 6 2 "="))
                         )
   in
            if isDepressed then
               whiteUpperShadow pItem
            else
               greyShadow pItem
            end if
   end let.
```

This view definition specifies that *pItem* displays a "=" sign surrounded by a box. If *isDepressed* is true, display *pItem* in reverse video. If *isDepressed* is false, display *pItem* in normal video with a shadow.



3.3.2 Basic Concepts of Coordinate Spaces

The Clock view language uses coordinate spaces to achieve geometric compositions of display primitives and component views. The concept of coordinate spaces is inherited from the Graphical View Language [6]. Before we start describing individual display primitives, it is necessary to introduce the basic idea of coordinate spaces.

A coordinate space is simply an area of display space. Display primitives are located by positioning primitives in a world coordinate space. This coordinate space has a (0,0) origin, and extends indefinitely along the positive arms of the (x) and (y) axes. The world coordinate space is mapped to the physical display screen by mapping the world space origin to the lower-left corner of the avail-

able physical window, and the maximum (x,y)-extent of the world space to the upper-right corner of the physical window. Most display primitives rely on a positioning primitive taking two coordinates to position them in a coordinate space. In Clock, the coordinate space also serves as a mechanism to group primitives and component views of user interfaces. The attached positioning primitives specify the spatial relation constraints and the sizing constraints of the display primitives and component views in a coordinate space. Thus, together with the positioning primitives, coordinate spaces are used to scale up simple primitives and display views to achieve randomly complex graphics for user interfaces. For example, in the following view specification,

```
nameBoxes = Views [
                At (x 10, y 10) stretching ( Box(Text "Mark")),
                At (x 100, y 10) stretching ( Box(Text "Larry"))
            ].
```

the **Views** function introduces a coordinate space. It groups two component views, two pieces of text "Mark" and "Larry" each surrounded by a box, to a single display view *nameBoxes*. The relative positions of the two component views within the *nameBoxes* are specified by the positioning primitive **At**. The picture generated by this specification is shown in Figure 13.

3.3.3 Display Primitives

The display primitives are those which are used to render the actual pictures. With the exception of the **Line** primitive, the positions of all display primitives

Figure 13: Name boxes.

are determined by the attached positioning constraints. The Clock textual view language contains a limited set of display primitives, namely, **Box, Line** and **Text**.

The **Box** primitive displays a box surrounding some content display views in the current default style. For example, the following view function

```
view = At ( x 0, y 0) ( x 100, y 100) Box (
            At ( x 10, y 10) stretching ( Text "Yonge Street" )
       ) .
```

displays a surrounding box of width 90 and height 90. Within the box, the text "Yonge Street" is displayed in a default font. The surrounding box locates at the lower-left corner of the display. The text "Yonge Street" is positioned at point (10, 10) with respect to the lower-left vertex of the surrounding box.

The picture generated by this view function is shown in Figure 14.

Box is an aggregate display primitive. A box primitive brings up a new coordinate space for the display views it encapsulates. Thus, it is impossible to refer

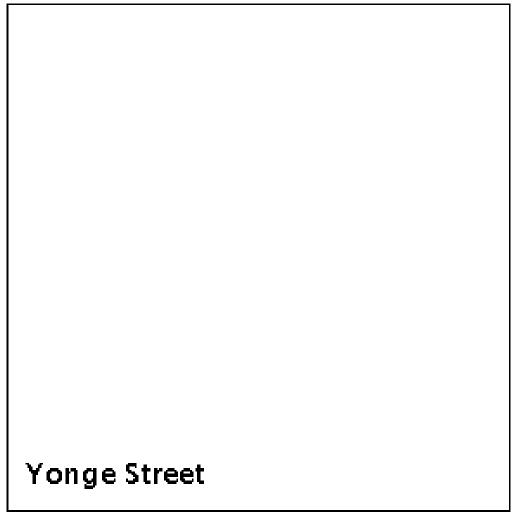


Figure 14: A picture generated using the "Box" primitive.

to unknown ordinates inside a box from the outside and vice versa.

If a box has unknown coordinates which will be determined by the layout rules, it is sized to exactly surround its contents. For example, in the following view function, **stretching** denotes an unconstrained coordinate which means that the upper-right vertex is permitted to stretch to fit the content.

```
view = At ( x 5, y 5) stretching (Box (
        At ( x 10, y 10) stretching ( Text "Yonge Street" )
    )) .
```

Thus, this view function draws the text "Yonge Street" surrounded by a box that is exactly sized to the right height and width to accommodate the text which is positioned at (10, 10).

Figure 15: A box with stretching top-right vertex.

The picture displayed is shown in Figure 15.

The location and size of a box may depend on the positioning and sizing constraints associated with it. For example, the view function

```
view = At origin ( x 120, y 120) Box (
           At ( x 40, y 20) ( x 80, y 40) Box ( Views [ ] )
       ) .
```

draws an outer box of size 120x120 and an inner box within the outer box. The lower-left corner of the inner box locates at coordinate (40,20) and its top-right corner locates at coordinate (80, 40). The height of the inner box is 20 and the width is 40. *origin* is a substitution for (x 0, y 0).

Visually, this is displayed as in Figure 16.

Sometimes, it is desirable to allow the size of a display view to be adjusted by the layout rules rather than specified absolutely. We can achieve this by using non-absolutely specified coordinates. For instance, the view function

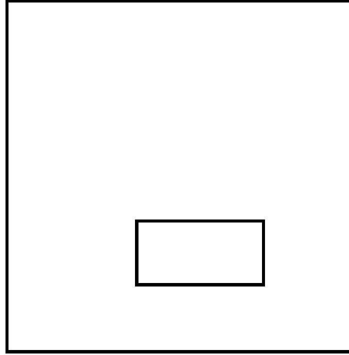


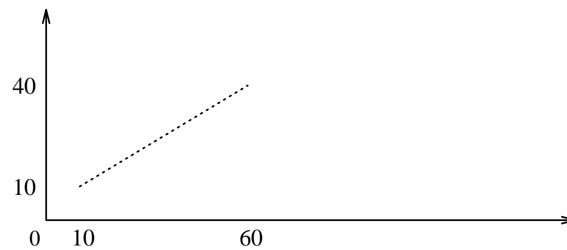
Figure 16: Two boxes with fixed position and size.

```
view = At (x 5, y 5) (LeftStretching 100, BottomStretching 100) Box(
  At (x 10, y 10) (LeftStretching 60, BottomStretching 40)  ( otherView )
)
```

displays a box whose lower-left corner is fixed in a coordinate space and a display view *otherView* whose lower-left corner is fixed in the box. But the top-right corners of both the box and *otherView* are not absolutely fixed though two coordinates are provided to specify the default cases. That means the *otherView*, whatever it is, will be displayed in the area from (10,10) to (40, 60) within the outer box if it can be fit there. In case this area can not hold the display view, the view may stretch along the top-right direction. **LeftStretching 100** denotes that the default ordinate is offset 100 to the left edge of the coordinate space. However, it may stretch to the right if necessary. Similarly, if the outer box can hold the inner display view, it is displayed as exactly of the size described in the specification. If it cannot hold the inner display view, it may stretch to the top and to the right in order to accommodate the content.

The **Line** primitive displays a line from the first coordinate to the second coordinate associated with it. The line style is determined by the current style context. For example, the following specification draws a dotted line from (10, 10) to (60, 40).

```
view = Style  Dotted ( Line (x 10, y 10) (x 60, y 40) ).
```

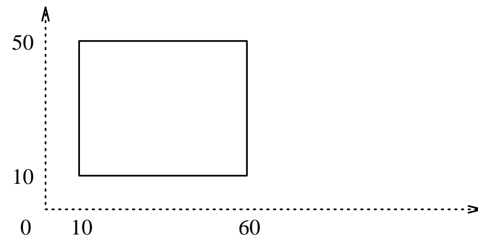


The **Text** primitive displays the string which follows it on the screen. The font of the displayed string is determined by the current font style.

3.3.4 The Views Primitive

The **Views** primitive introduces a new coordinate space. It is mainly used to group a set of individual display views to form a bigger one. The spatial arrangement of component views can be achieved using constraint primitives. A special specification **Views**[] returns a null display view. It is usually used with a **Box** primitive to create an empty box. For example, the following view function generates an empty box of size 50 by 40.

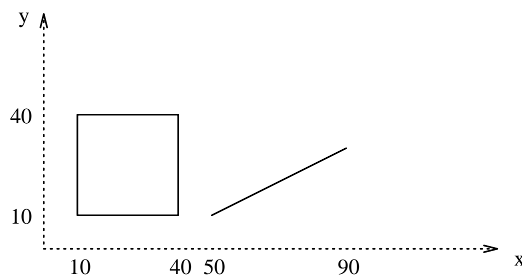
```
view = At (x 10, y 10) (x 60, y 50) (Box(Views[])).
```



3.3.5 Coordinates and Coordinate Spaces

All the Clock language primitives except the **Line** primitive can be located by the positioning constraints specified in the coordinate space that surrounds them. Their sizes are implicitly specified through the positions of their lower-left and top-right corners. The positions of the starting and ending points of a line are specified by the pair of coordinates associated with the **Line** primitive. For example, the following view specification displays a fixed-size box and a line.

```
view = Views [
            At (x 10, y 10) (x 40, y 40) (Box(noView)),
            Line (x 50, y 10) ( x 90, y 30)
        ].
```



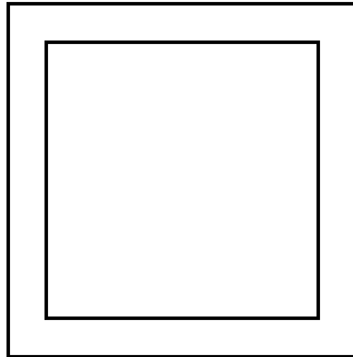


Figure 17: A picture that demonstrates the specification of relative size and position.

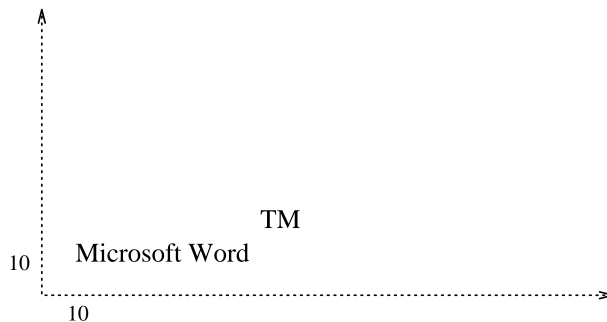
In a view specification, the value of an x or y ordinate can be specified absolutely, or can be left unspecified, to be determined by the layout rules. An absolutely specified ordinate consists of a reference edge and an offset. Reference edges refer to the **Left** edge, the **Right** edge, the **Top** and the **Bottom** edge of the coordinate space in which the primitives are specified. The offset is simply an integer value. So, "Left 10" means 10 units to the right of the left edge of the surrounding coordinate space and "Top (-20)" means 20 units below the top edge the surrounding coordinate space. For example, the view specification

```
view = At origin (x 100, y 100) (
           Box (  At (Left 10, Bottom 10) (Right (-10), Top (-10))  (
                Box(noView)
           ))
       ).
```

displays a box within a bigger box. The picture is shown in Figure 17. The lower-left corner of the inner box is located at (10, 10) to the lower-left corner

of the outer box. The upper-right corner of the inner box is located at (-10, -10) relative to the upper-right corner of the outer box. It does not make sense to associate a negative offset with the **Left** or the **Bottom** edge, or to associate a positive offset with the **Right** or the **Top** edge of the coordinate space. An unspecified ordinate to be determined by the layout rules can be denoted as **XSomewhere**, **YSomewhere**. A non-absolutely specified ordinate starts with the key word **XBaseOffset** or **YBaseOffset** followed by an ordinate label and an offset. The ordinate label is a natural number used to denote a reference point either along the x axis or the y axis. For example, the following view function specifies: display a piece of text "Microsoft Word" and a superscript. The text locates at (x 10, y 10). Since we don't know the exact size of the text, its top-right corner is unconstrained. It is denoted as "(XBaseOffset 1 0, YBaseOffset 1 0)". The superscript "TM" is put at 5 units right and 5 unit above to the top-right point of the text.

```
view = Views [
                    At (x 10, y 10) (XBaseOffset 1 0, YBaseOffset 1 0) (Text "Microsoft Word"),
                    At (XBaseOffset 1 5, YBaseOffset 1 5) stretching (Text "TM")
                  ].
```



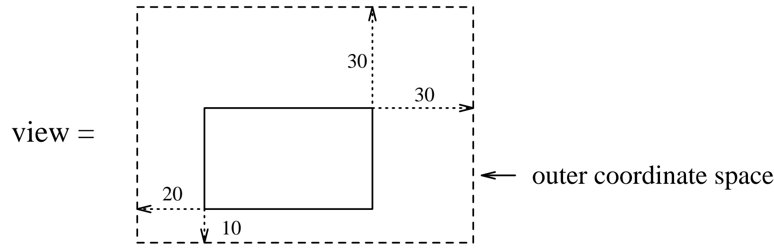
3.3.6 The Positioning Primitive

In the Clock language, the positioning primitive **At** is used to describe spatial constraints of display primitives or display views in a coordinate space.

An **At** is followed by two coordinates, which consist of either absolutely specified, non-absolutely specified or non-specified ordinates. An absolutely specified ordinate is denoted by a reference edge and an offset. For example, the view specification

```
view = At  (Left 20, Bottom 10)  (Right (-30), Top (-30))  ( Box (Views[]))
```

displays an empty box in a coordinate space. The position of the empty box relative to the coordinate space is shown in the following:



That means the lower-left corner of the box is kept 20 units right to the left edge of the coordinate space and 10 units up to the bottom edge of the coordinate space. Similarly, its top-right corner is kept 30 units below the top edge of the coordinate space and 30 units left to the right edge. Thus, the position of the box relative to the outer coordinate space is kept fixed. In case that the outer coordinate space stretches, the box will stretch accordingly to comply the constraint.

3.3.7 Function Evaluation

Function definition and evaluation introduce the abstraction mechanism in Clock. In the graphics context, it implies that we can define basic individual interface components as functions that are composed of display primitives, then incrementally scale them up to bigger components and eventually obtain the desired complex graphics of the interface. This is the same function definition and evaluation mechanism found elsewhere in the functional language. It is not special for the view language.

For example, in the following specification,

```
textBox1 = Style Thick ( Box(Text "Kingston")).
textBox2 = Style Thick ( Box(Text "Watertown")).

view = Views[
            At (x 10, y 10) stretching (textBox1),
            At (x 140, y 10) stretching (textBox2)
        ].
```

the *textBox1* and the *textBox2* functions are previously defined. Then, they are evaluated in an outer coordinate space. The results of evaluating these two functions are views of the *textBox1* and the *textBox2*. Through function evaluation, these two subviews are plugged into a coordinate space to compose a bigger view.

The picture specified by this function appears as:

3.3.8 Conclusion

The Clock textual view language is a purely declarative language with no side-effects. The view function specification acts as a set of one-way constraints to map the system states to pictures on the display. The programmer needs only to specify what the picture is supposed to look like. How and when the picture is generated is left to the system and is transparent to the programmer. In this way, the programmer is freed from thinking about pictures at very low level. Rapid prototyping is thus supported.

In Clock, display views are considered as first class values. Thus, among other

operations, they can be assigned to variables, passed as parameters and used to compose bigger views. Making views first class values offers extra power and flexibility in expressing pictures. The programmer can easily scale up the most basic drawing primitives and component views to arbitrarily complex pictures. In the interactive calculator example, to generate all the operator keys, the following view function specification is used:

```
operatorSigns = ["+", "-", "*", "/"].

opbuttons = map okItem operatorSigns.

view = beside ( insertBetween opbuttons (space 4)).
```

The *okItem* function takes a character parameter and returns a picture of a button labeled by that character. The four operator buttons are created by applying the *okItem* function to operatorSigns: a list of four characters. Then, the function *insertBetween* is applied to the *opButtons*, a list of four buttons. The effect is: a space of 4 pixels is inserted between adjacent buttons. The view is finally achieved by applying the *beside* function to the four buttons with fixed space among them. The result is: the four buttons are displayed in a horizontal line one by one.

However, using the textual view language, the programmers still have to conceive a picture in terms of textual primitives. The inherent problems of using textual language to specify display views are not solved. The programmer is obligated to go though a significant conversion process to specify graphics in textual representations. To achieve a desirable view, the programmer may have to first draw the picture on a graph paper, obtain the coordinates and then translate

the picture into a textual specification. Furthermore, textual representations are difficult to understand. Some information (such as spatial relationship among component views) that is explicit in a graphical specification is implicitly represented in the corresponding textual specification. This implies that more effort and refinement iterations are needed to achieve desirable views if the programmer is forced to use textual specifications.

4 Mixed-form Visual-textual Programming

This chapter first presents motivations for mixed-form visual-textual programming. The advantages and shortcomings of employing graphical and textual primitives in a language for user interface development are discussed. We make the argument that a mixed-form language with a set of carefully designed graphical primitives fluidly mixed with a set of textual primitives may best meet the programmer's needs in specifying user interfaces. Several motivating examples will be presented to justify this argument. In the second section of this chapter, we will introduce, in an informal way, the mixed-form programming in the Clock environment. A formal description follows in Chapter 5. A complete, simple application is developed using mixed-form programming to demonstrate the power. In the last section, several design concerns and trade-offs of graphical primitives are discussed.

4.1 Motivation

It is generally accepted that many of the difficulties encountered in developing user interfaces result from insufficient support in the underlying programming language [9]. For example, most currently available languages have only text-based I/O primitives: read or write a string, and no graphical I/O primitives are supported. To address this problem, a number of toolkits have been developed, such as X [22] and Interviews [14]. The idea behind toolkits is to use a large external library to handle the graphical layout of the interfaces. However, even with such external graphical routine libraries, the programmer still experiences difficulties in achieving the desired graphical views. This is primarily because

the programmer is forced to specify pictures in terms of textual primitives and concrete coordinates. In this process, the programmer has to go through conversions from pictures to text, and from text to pictures. These conversions are unintuitive and can be error prone. The difficulties people experience in using textual-form graphical routines to specify graphical views reveal that specifying graphics using text imposes a heavy burden on programmers. Furthermore, a textual representation of graphical views is difficult for people to comprehend.

A number of visual languages have emerged to allow people to program by drawing pictures and by directly manipulating icons. In this approach, the programmer can specify a picture in a much more intuitive way by simply drawing it. The erroneous conversion in specifying pictures using textual representations is eliminated. However, in visual languages, all primitives are purely graphical. Graphical primitives are less expressive and cumbersome to use in specifying certain aspects of the program, such as control structure and abstraction.

Based on the realization of the advantages offered by purely textual and graphical primitives and their inherent problems, we propose that a language mixing both textual primitives and graphical primitives may best meet the needs in programming user interfaces. In such a mixed-form language, the textual and graphical primitives may draw on each other's merits to offset their shortcomings, and offer the most convenience and expressive power to the programmer. Our goal is to combine the advantages of textual and visual programming. The functional paradigm provides an excellent framework to enable fluid mixture of textual and graphical primitives.

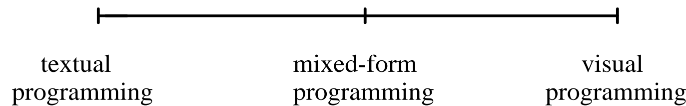


Figure 18: Mixed-form programming is somewhere at the middle point between purely textual and purely graphical programming.

4.1.1 Norman's Gulfs of Execution and Evaluation

The difficulties people encounter in using textual languages to develop graphical user interfaces have been well observed. First of all, it is difficult to attain textual representation of pictures. Secondly, such textual representations are hard to understand. A number of iterations are usually necessary to achieve the desired picture.

In his interesting book "The Design of Everyday Things" [20], Donald Norman presents a psychological analysis of human actions. According to his theory, an action involves two major steps: doing something and checking the result, which are referred to as *"Execution"* and *"Evaluation"*. Execution involves doing something. To get something done, people start by forming a *goal*, the specification of what is to be achieved. This goal is somehow translated into an action sequence. Execution of this sequence of actions is supposed to achieve the goal. Evaluation is the process of comparing what happened and what people want to see happen. In other words, it is the comparison of the result of the action sequence with the initial goal. Evaluation starts with perceiving what happened in the world as the result of the action sequence. The result is then compared with respect to what was wanted initially. The stages of execution and evaluation are illustrated in the Figure 19.

Norman identified that the difficulties people encounter in doing everyday

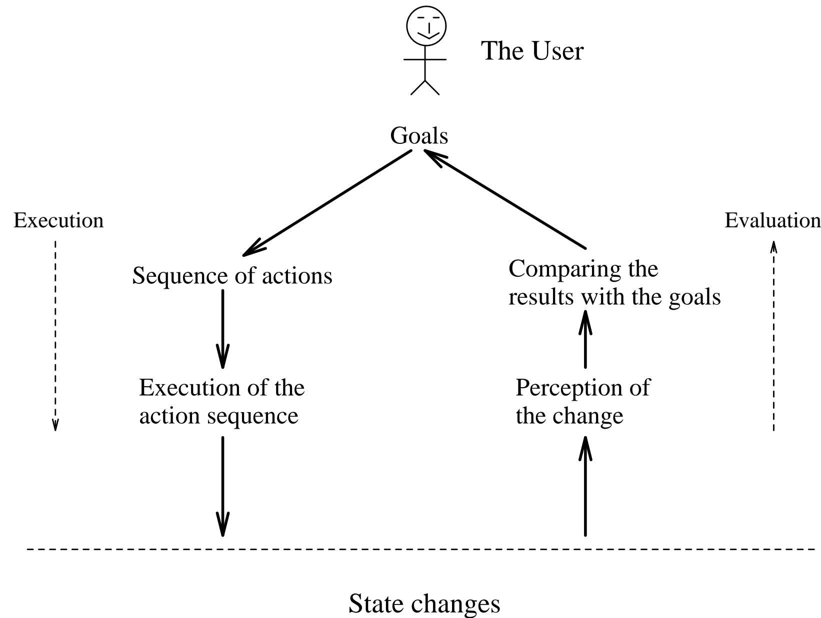


Figure 19: Illustration of the stages of execution and evaluation.

tasks reside entirely in deriving the relationships between mental intentions and physical actions to realize the intentions. The distances between human mental representation and physical actions are called *gulfs*. Such gulfs are the major sources of difficulties for the users.

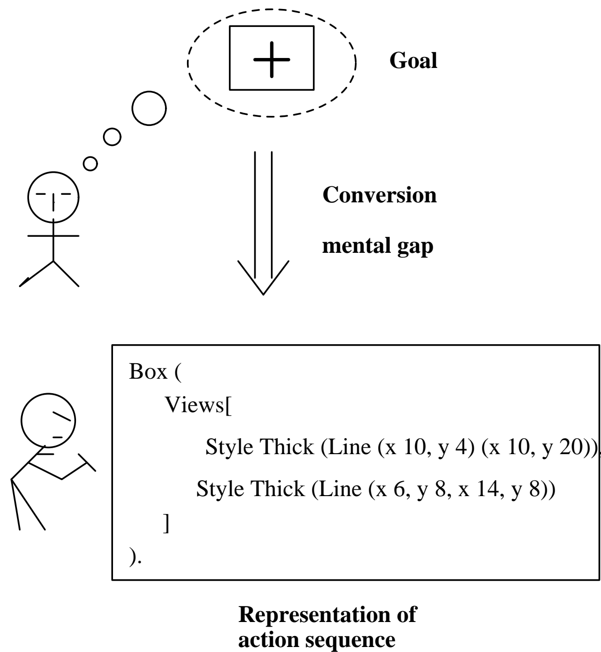
The difference between human intentions and the sequence of actions executed by the system to achieve such intentions is the *Gulf of Execution*. This gulf reflects the difficulty people encounter in mapping their goals to sequences of actions. The more the system allows people to do the intended actions directly without extra effort, the smaller the Gulf of Execution. The *Gulf of Evaluation* measures the difficulty people encounter in obtaining the updated states of the system, in interpreting them and in comparing them to the initial goals. The Gulf of Evaluation is small if the representation of the new system states of the system can be easily obtained, directly interpreted in terms of people's intentions.

Thus, the system design goals would be, to minimize the Gulf of Execution and Evaluation to allow the user to easily map their intentions to allowable operations and to easily obtain comprehensible feedback information.

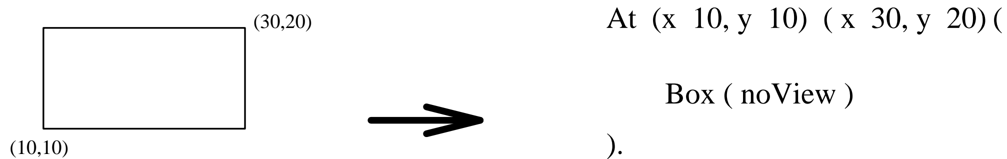
It is not difficult to apply the concepts of Gulf of Execution and Evaluation to the problem of using textual languages to develop graphical user interfaces. The amount of effort the programmer exerts to translate the desirable picture - the goal into its textual representation forms the Gulf of Execution. There is usually no difficulty involved in interpreting and comprehending the system feedback - the picture generated by the system, as the way the feedback is presented exactly matches the way people think of their goals. However, the programmer has to spend time on re-compiling the program and re-executing it before he can get the feedback. Thus, the Gulf of Evaluation resides in obtaining the feedback from the system.

4.1.2 Gap between Pictures and Their Textual Representations

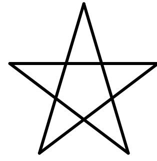
Specifying pictures using textual languages is not a trivial task. There exists a significant gap between the programmer's intention and the representation used to carry out the intention. The goal is pictures. But the sequence of actions to achieve the goal are specified in terms of textual strings. To bridge them, a conversion which demands significant amount of effort is needed.



Graphical views are inherently two-dimensional entities while text is one-dimensional. In the conversion from two-dimensional entities to one-dimensional notions, concrete coordinates and numerical constraints are inevitable. For example, a box with two vertices at (10,10) and (40,20) and its specification in Clock textual view language is shown below:



(x 10, y 10) and (x 30, y 20) are used to denote the two vertices of the box. In this way, the two-dimensional geometric constraints, in this case, the size and position of the box, are encoded into one-dimensional notions.



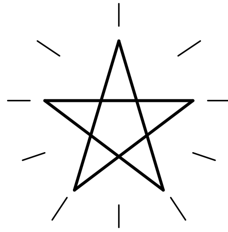


Figure 20: A five-point star example.

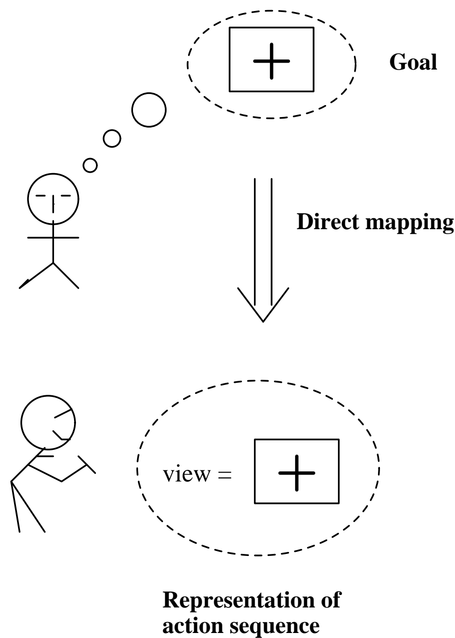
Two major factors contribute to this gap. First, in the picture world, absolute values are rarely used. Most geometric constraints are specified by relative positions and relative sizes. In contrast, to describe a picture in textual primitives, the only way to encode geometric constraints is to use concrete coordinates or ratio values. No wonder sometimes people might have to draw a picture on a piece of graph paper, and then derive the concrete coordinates and numbers. Secondly, some constraints explicitly encoded in a picture have to be specified implicitly using texts. In the previous example, the size of the box is implicitly represented in the textual specification.

Figure 20 shows a simple program which displays a five-pointed star. The star has two possible states: dim or shining. The user may toggle the state by clicking mouse button on the picture of the star. The textual specification of the picture in Clock view language is shown in the following:

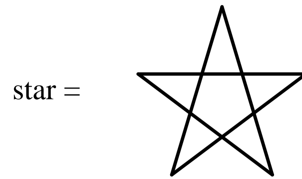
```
star = Style Thick (
          polyLine [(30,100), (70,100), (80,140), (90,100), (130,100),
                   (100,75), (110,40), (80,60), (50,40), (60,75), (30,100)]
      ).

shiningStar = Views [
               Style Thick (
                  polyLine [(30,100), (70,100), (80,140), (90,100), (130,100),
                           (100,75), (110,40), (80,60), (50,40), (60,75), (30,100)]
               ),
                  Line (x 5, y 100)   (x 20, y 100),
                  Line (x 25, y 140)  (x 40, y 130),
                  Line (x 80, y 165)  (x 80, y 150),
                  Line (x 120, y 130) (x 135, y 140),
                  Line (x 140, y 100) (x 155, y 100),
                  Line (x 130, y 65)  (x 145, y 60),
                  Line (x 100, y 75)  (x 110, y 40),
                  Line (x 80, y 15)   (x 80, y 30),
                  Line (x 35, y 20)   (x 45, y 35),
                  Line (x 15, y 60)   (x 30, y 65),
                  Line (x 115, y 35)  (x 125, y 20)

             ].



view = if powerOn
         shiningStar
      else
         star
      end if.
```

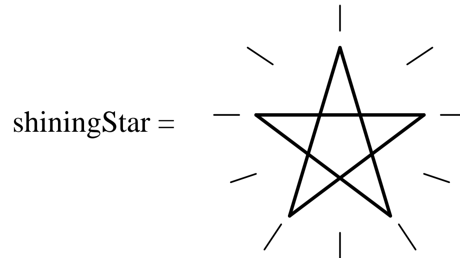
This example clearly demonstrates how difficult it could be to determine all the coordinates in specifying pictures with a textual language.

It is much easier and more intuitive to use graphical notions to specify pictures. As we can directly map our goal, the desired picture, into the representation of the actions to achieve the goal, the Gulf of Execution is basically eliminated.



As an example, the Clock graphical specification of the star picture is shown below:






```
view =  if  powerOn then
              shiningStar
          else
              star
        end if.
```

In this specification, we simply draw the five-pointed star and several identical segments of lines and arrange their relative positions by moving them around to the appropriate sites.

4.1.3 Poor Comprehensibility of Textual Representations

In textual representations, pictures are described in terms of concrete coordinates, numerical constraints and textual primitives. Some geometric constraints that are explicit in a graphical specification may be hidden or indirectly represented in the corresponding textual specification. The programmer needs to

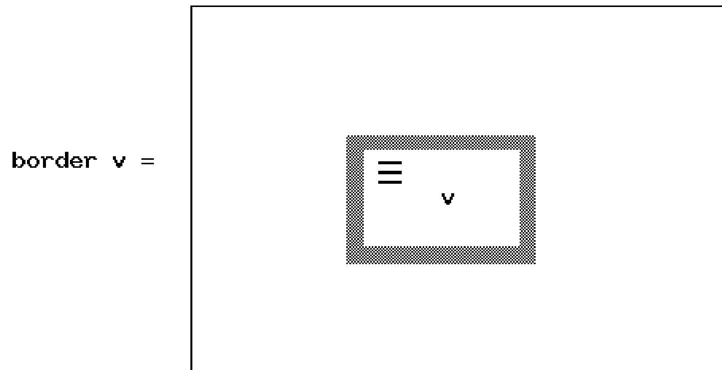


Figure 21: A view function in graphical representation. v is a predefined display view, and *border v* adds a gray border to it.

perform a reverse conversion to see what the picture really looks like. For example, the following is a view function specification in the Clock textual view language:

```
border v = Style Solid (
            Views[
                Style NoBorderGrey ( At (Left 0, Bottom 0) (Right 0, Bottom 7) (Box(noView))),
                Style NoBorderGrey ( At (Left 0, Top (-7)) (Right 0, Top 0) (Box(noView))),
                Style NoBorderGrey ( At (Left 0, Bottom 7) (Left 7, Top (-7)) (Box(noView))),
                Style NoBorderGrey ( At (Right (-7), Bottom 7) (Right 0, Top (-7)) (Box(noView))),
              At (Left 7, Bottom 7) (Right (-7), Top (-7)) (v)
           ]
         ).
```

It might take minutes for people to convert this piece of code to the picture it really describes. In comparison, the graphical specification of the same pic-

ture is shown in Figure 21. Almost at the first glance, people can identify that this picture is meant to add a border to a predefined display view. Since people working on textual specifications have little idea what the picture looks like, a minor modification of the textual representation may require re-compilation and re-execution of the program to see the actual effect. This re-compilation and re-execution procedure helps people cross the Gulf of Evaluation. However, the inefficiency and inconvenience of this bridge are unacceptable.

A graphical representation directly and naturally reflects the properties of the resulting picture. It not only enables a direct and natural mapping from the goal to the action sequence, but also shortcuts the evaluation process.

4.1.4 Shortcomings of Graphical Primitives

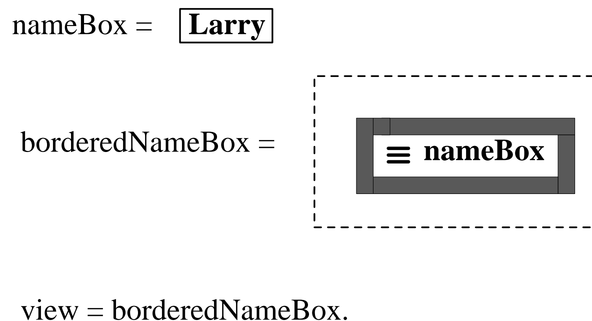
Graphical primitives offer us convenience in specifying graphical components and in arranging their geometric constraints. However, they are less expressive in specifying certain aspects of the program such as control structures, identifiers and parameters.

It is easy to end up with a messy graph if the programmer uses graphical icons to specify the program control structure. This mainly accounts for the limitation of the sizes of visual programs.

It seems that conventional interface builders do not need identifiers and parameters. All interactive objects are graphically presented. However, the fact is user interface builders use identifiers and parameters at a lower level. For example, in Microsoft Visual Basic [16], each graphical object is associated with

a unique ID, through which the object can be referred to in the textual program. Each interface object has a predefined set of "Properties" like "Height", "Width", "ForeColor" and "BackgroundColor", which determine the object's appearance and behavior. These properties are in fact parameters. They can either be manipulated interactively or accessed and updated within the program through their names. Interface builders try to combine advantages of visual and textual programming in making interface objects graphically manipulatable and programmatically accessible through their IDs. They also provide some system support to bridge the interface part and the program logic part. However, interface builders can only handle interfaces with static appearance. The number of interactive objects in such interfaces has to be fixed. Interface builders are incapable of specifying interfaces with dynamic nature, such as the equalizer example mentioned later in this chapter.

It is difficult to use graphical icons to represent identifiers. In Clock, identifiers serve as part of the abstraction mechanism. Functions and subviews may be first defined and later on be referred through their identifiers in a bigger context. For example, in the following view function specification:



A *nameBox* function is first defined. Another function *borderedNameBox* adds

a border to it. In *borderedNameBox*, *nameBox* is used as a component view and is referred by the function name.

If the programmer is restricted to use only graphical primitives like in some purely visual languages [4], icons have to be used as identifiers. However, graphical icons seem to be handicapped in identifying entities. For example, in the *Pict/D* visual language, four colors are used as identifiers. Thus, the number of variables allowed in *Pict/D* is only four, which is far from sufficient in real applications. In general, text strings are the best candidates for identifiers.

Furthermore, in some cases, it is impossible to represent parameters graphically. In the previous border example, it is easy to denote the parameter *width* in the textual specification. Thus, function "border v width" means: add a border of width *width* to view *v*. Using graphical primitives, there is no way for us to specify the *width* parameter.

4.2 Mixed-form Programming in Clock

The mixed-form Clock language consists of both textual primitives and graphical primitives. The mixture of textual and graphical primitives in a single language is based on the realization that textual and graphical primitives have their own advantages and disadvantages. By properly mixing them together, we allow the programmer to choose either of the two forms of primitives depending on which is more appropriate.

This section first uses a simple graphical equalizer application to demon-

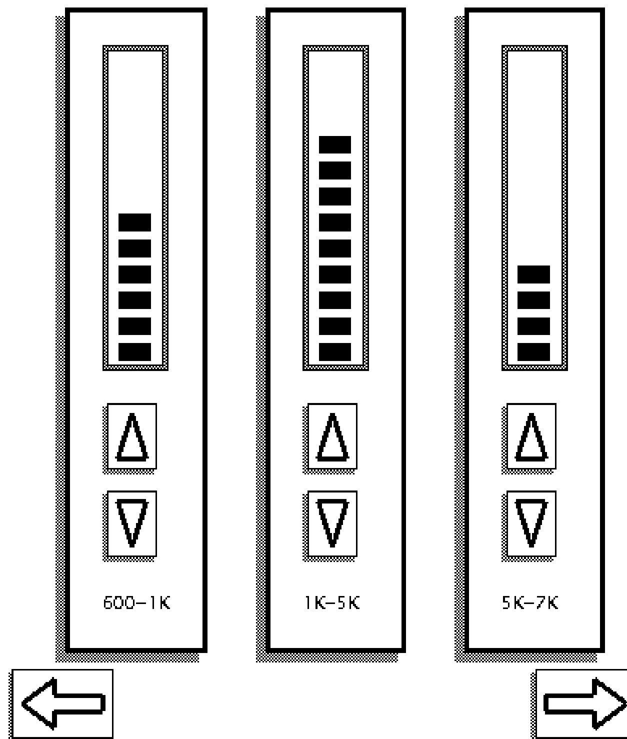


Figure 22: A dynamic graphic equalizer.

strate the usefulness of mixed-form programming. The advantages provided by the mixed-form programming in Clock are then discussed. An informal description of the mixed-form Clock view language and the prototype graphical editor are presented in the final part.

4.2.1 An Equalizer Application

As a small, complete example to demonstrate the power of mixed-form programming, a dynamic graphics equalizer application is developed. The interface of the application is shown in Figure 22.

The interface is composed of a row of equalizer controllers, a left arrow button and a right arrow button. Each controller consists of a graphical display, an increase button and a decrease button. To increase or decrease the db for sound of a given frequency range, the user simply presses the increase or decrease button of the corresponding controller. The number of small filled rectangles will increase or decrease accordingly to show the db level. However, the number of available controllers can be varied. To adjust the db for sound of a certain frequency range, the user may need to pop up the controller for that frequency range by pressing the right arrow button. As the number of controllers in the row increases, the right arrow button will move accordingly to the right to make the the interface geometrically balanced.

With some conventional interface builders, it is not difficult to develop some static interfaces in which the number of objects is fixed and their positions are also fixed. However, traditional interface builders are handicapped in composing dynamic interfaces like this graphical equalizer. In the equalizer application, the db level display, the number of controllers are dynamic, and the right arrow button may float according the number of controllers available. Obviously, it is not a trivial task to specify such dynamic features using the conventional interface builders, if not totally impossible.

In the following, we demonstrate how we can develop this equalizer application using mixed-form programming in Clock.

The view specification for the equalizer display is shown in Figure 23. In the specification, the function *bars* with parameter *amount* yields a picture of

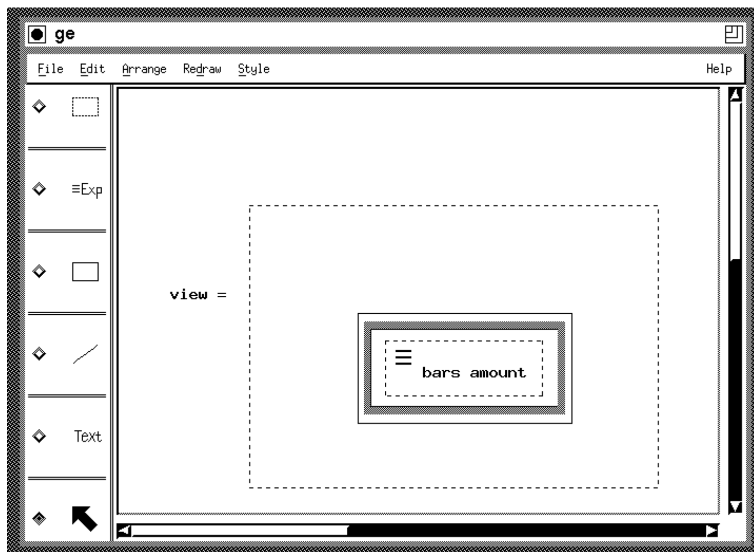


Figure 23: Mixed-form specification for the equalizer display.

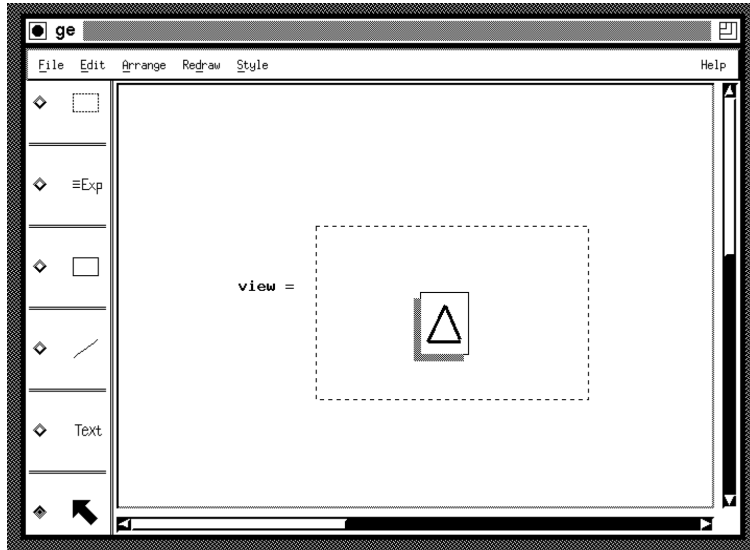


Figure 24: View specification for the increase button when it is not pressed.

a column of small filled rectangles. Surrounding this picture, there is a box, a gray border added to the box and another box surrounding the border. The *bars* function is defined textually as:

```
bars [] = noView.
bars [c] = Views [bar].
bars [c|cs] = nabove [ bars cs, space 5, bar].
```

Here, we use graphical primitives to handle the static screen layout and use textual primitives to handle dynamic pictures.

The visual specification for the increase button and the right button are shown in Figure 24 and 25 respectively.

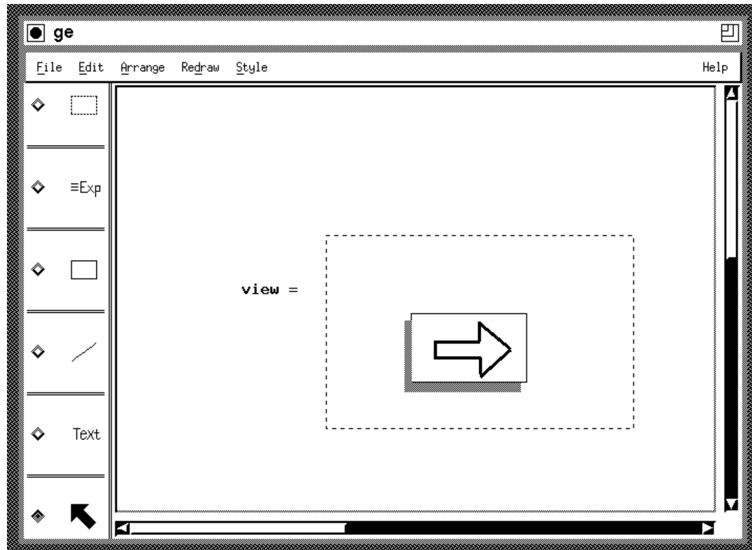


Figure 25: Visual specification for the left arrow button when it is not pressed.

The view of the controller row is defined textually as:

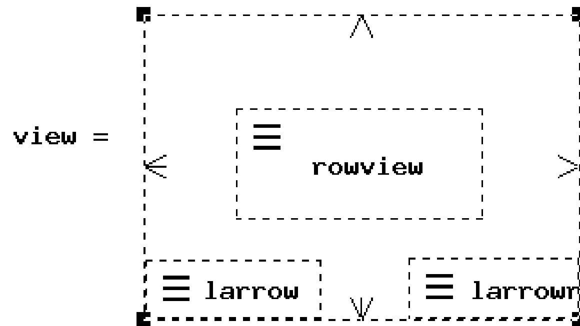


Figure 26: Mixed-form specification for the equalizer application.

```
kth 0 _ = [].
kth 1 s = let [a | _] = s in
              [a]
          end let.

kth k [a | as] = [a | kth (k-1) as].

rangelist = kth getBCounter range.

rangess = map oneRange rangelist.

rowview = beside (rangess).
```

The view specification for the whole application is shown in Figure 26.

In this specification, three objects *rowview, larrow* and *rarrow* have been grouped together. Thus, their relative positions are fixed. If the *rowview* stretches to the right as the number of controllers increases, the right arrow button will

move to the right accordingly.

4.2.2 Advantages Offered by Mixed-form Programming

Clock is a purely functional language. The functional style is in harmony with a graphical representation. Based on the functional paradigm, the Clock language has no side-effects. The programmer can freely combine textual primitives, graphical primitives and any predefined display views to generate the desired complex graphics of the user interface. There is no danger that any of the display views will have a side effect that could result in interference to another display view.

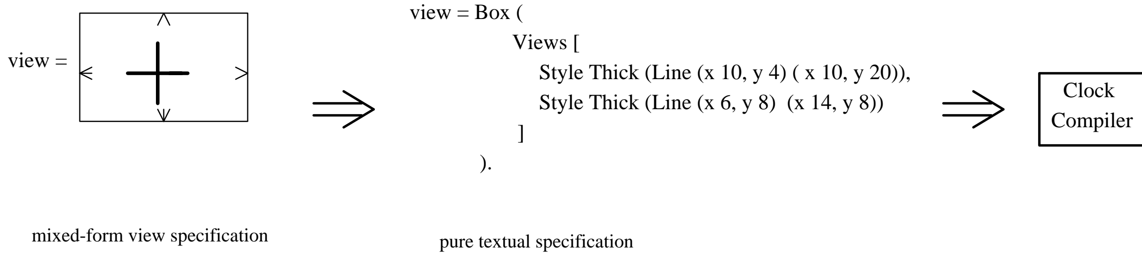
Based on the declarative style of Clock language, display views are described directly rather than generated by imperative commands. For example, **Box(Text "Hello")** means "put a box around the text Hello and make the size of the box exactly able to hold the text". Thus, the programmer needs only to describe the result he/she wants, and how to achieve the effect is left to the underlying layout algorithm.

In Clock, graphical views are first class values. Thus, we can insert a pictorial specification of a view wherever an expression is allowed. Similarly, an expression can be inserted into pictures as long as evaluation of that expression yields a display view. In this way, fluid mixture of textual and graphical forms can be achieved. The programmer can freely choose the appropriate form of primitives in specifying user interfaces.

In Clock, function definition and evaluation serve as the primary abstraction mechanism. In user interfaces, we can define basic individual interface components as functions using a small set of basic primitives and then scale them up to bigger components and eventually obtain the desired complex pictures of the interface.

4.2.3 Mixed-form Programming in Clock

The original Clock view language has only textual primitives. In the mixed-form view language, a set of graphical primitives are introduced to take places of corresponding textual equivalents. In general, in order to make a program with heterogeneous primitives to be compilable, an intermediate language with uniform primitives has to be used. The mixed-form program is translated into the intermediate language which can be fed into the compiler to produce the executable code. In Clock, the original textual view language serves as this intermediate language. With mixed-form programming, a view description can be specified using both graphical primitives and textual primitives. This mixed-form view specification is then translated into the original Clock textual view language to produce the actual picture. This is possible because each graphical primitive introduced can be directly mapped to one or a set of textual primitives in the original view language.



In the mixed-form view language, all the display primitives in the textual view language such as **Line** and **Box** have been visualized to be graphical primitives. In addition, some positioning primitives are also visualized. The graphical positioning primitives are designed so as to reflect the nature of the screen layout algorithm underlying the textual view language. A prototype graphical editor has been developed to allow the programmer to edit mixed-form programs. With the graphical editor, the programmer can simply draw graphical primitives or type in textual primitives to compose mixed-form programs. The editor will then translate the mixed-form view specification into the corresponding textual specification.

4.2.4 The Graphical Editor

The Clock graphical editor serves as both a mixed-form program editor and a preprocessor. It enables the programmer to compose view specifications using both graphical and textual primitives. The graphical editor also preprocesses a mixed-form program by translating the graphical primitives within it into the corresponding textual equivalents. The derived textual view specification can then plugged into an application and be compiled to yield the actual display

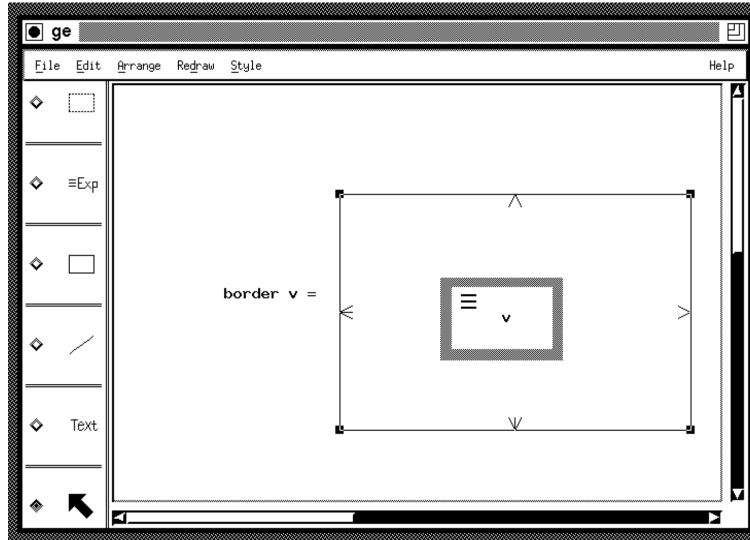


Figure 27: The Clock graphical editor.

view.

A snapshot of the graphical editor is shown in Figure 27.

The graphical editor consists of a drawing area, a palette of graphical primitives and tools and a pulldown menu bar. On the left side of the editor, there is a palette of graphical primitives and tools including buttons for *canvas, function evaluation sign, box, line, text* primitives and a pointing tool. This palette is similar to the tool palette in MacDraw [15]. The programmer can choose to draw a graphical primitive by selecting it in the palette. Each graphical primitive drawn on the screen is associated with two or four small handles. The programmer can

resize the object by dragging the associated handles. The handles are also used to indicate the currently selected object. The pointing tool is used to select an object for operation, to move an object or to resize an object by dragging the handles.

The positioning and sizing primitives are visualized as four small magnets associated with a canvas or a box. The magnets are used to specify the relative positions and sizes of objects within a surrounding box or canvas. Each magnet has two possible states: *On* or *Off*. If a magnet is on, it intuitively indicates that the magnet attracts to the inner object to make it fixed to the edge of the surrounding object to which the magnet is associated. Thus, if the edge of the outer object moves, the corresponding edge of the inner object will move accordingly. In this way, the underlying layout algorithm of the view language is visualized. The programmer can actually see the effect of the positioning and sizing constraints by resizing the outer object.

The programmer can use the **File** menu to retrieve a mixed-form view specification from a file with extension *.clk* or store the current specification into a file. The **Style** menu is used to manipulate certain attributes of the selected object such as fill pattern, line style and line width. The **Arrange** menu is used to group or to align objects. The **Translate** button in the **File** menu is used to translate the current mixed-form specification into the corresponding textual specification and store it in a given file.

The Clock graphical editor is a prototype tool developed to demonstrate the idea proposed in this thesis. Some features are not fully implemented. For ex-

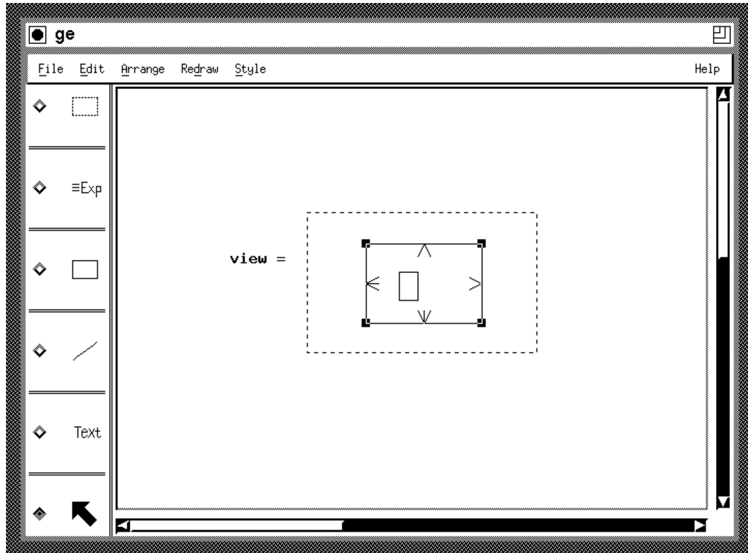


Figure 28: Handles and magnets

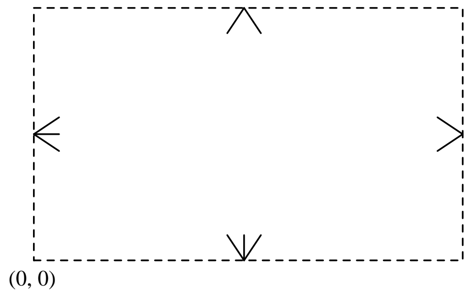
ample, the editor supports only simple text editing.

4.3 Graphical Primitives in the Mixed-form Language

In this section, the graphical primitives incorporated into the Clock mixed-form view language are introduced in an informal way. A formal description will be presented in chapter 5. The graphical primitives include the canvas and positioning primitives, a set of display primitives, the function evaluation and the grouping primitives.

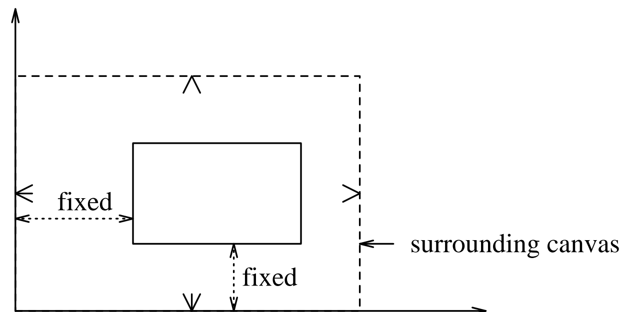
4.3.1 Canvas and Positioning Primitives

The graphical canvas primitive appears as a dashed box. A canvas is used to reserve space to hold some display views and introduce a geometric context for the views within it. A canvas has a lower-left vertex of coordinate (0, 0) and may stretch to the upper-right indefinitely. The relative positions of component views can be specified by a canvas and the associated positioning primitives. The graphical box primitive and the function evaluation primitive introduce an invisible canvas associated with them. The graphical canvas corresponds to the coordinate space primitive in the textual view language. A canvas is always associated with four magnets.



canvas and magnets

The four magnets are graphical primitives to specify spatial arrangement of component views in a canvas or a box. Each magnet has two possible states: *on* or *off*. If a magnet is on, it makes the distance between the edge of the canvas it attaches to the corresponding edge of the component view fixed. If it is off, this distance may be varied. Intuitively, magnets drag pictures with them as they move. In this way, some positioning constraints can be maintained. This magnet mechanism reflects the underlying layout algorithm. The screen layout constraint solver is visualized by the magnet. Using the Clock graphical editor, the programmer may resize a canvas by dragging one of its handles and actually see how the component views stretch or move to maintain some specified positioning constraints.

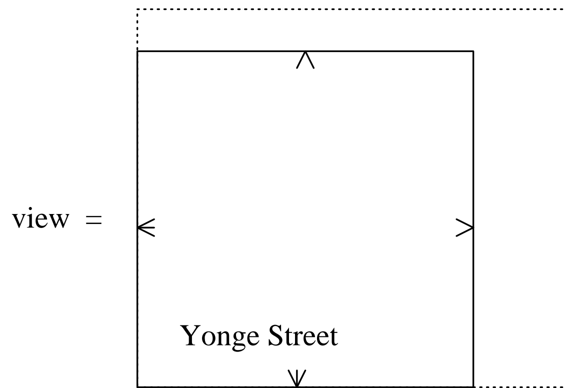


4.3.2 Display Primitives

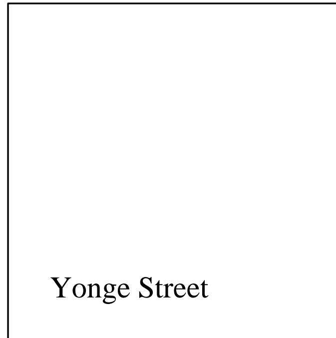
Graphical display primitives include box, line and text.

The graphical primitive for *Box* is just a box drawn on the screen using the Clock graphical editor. Certain attributes of a box such as fill pattern, line style and line width can be modified using the **Style** menu in the editor. Box primitives always introduce an invisible canvas with them. Thus, a box is always associated with four magnets to specify the spatial relationship of its component views.

In the following view specification, a piece of text is displayed in a box of a given size. The position of the text is fixed relative to the lower-left corner of the box.



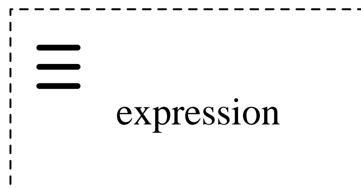
The picture generated by this view function is shown as follows:



The graphical primitive for *Line* is just a line drawn on the screen using the graphical editor. The line width and line style may be changed using the **Style** menu. A line can be modified by dragging the small handles associated with its two vertices.

4.3.3 Function Evaluation Primitive

The graphical function evaluation sign is shown below:

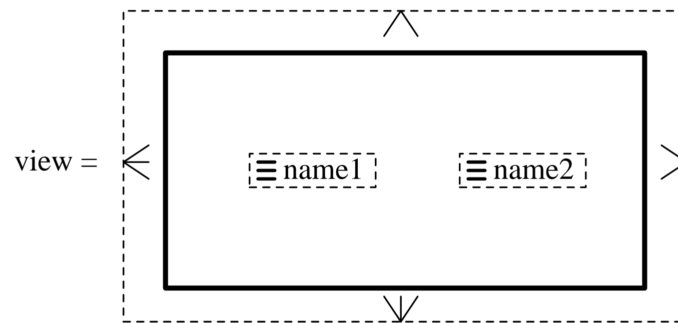


The evaluation sign introduces a new canvas. The expression within evaluation sign will be evaluated to yield a display view and put it in the new canvas. Through this approach, predefined display functions can be slotted to compose a bigger display view.

For example, in the following specification,

name1 = **Larry**

name2 = **Moe**

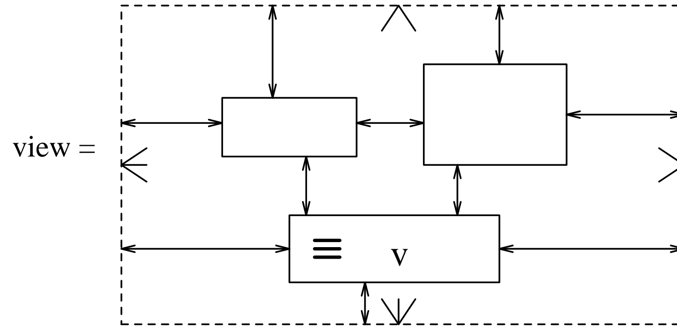


the *name1* and *name2* functions are previously defined. Then, they are evaluated in an outer canvas. The results of evaluating these two functions are views of the *name1* and the *name2*. Through function evaluation, these two subviews are substituted into a canvas to compose a bigger view.

4.3.4 The Grouping Tool

Using the *Group* option in the **Arrange** menu, the programmer can group component views in a canvas or box. The grouping tool is used to make the relative positions of component views in a canvas or box fixed.

As an example, in the following view specification, if a display view v and two boxes are grouped together, the distances denoted by arrow lines are kept fixed.



4.4 Design Tradeoffs

There are a number of tradeoffs in the design of the mixed-form Clock language. First of all, we have to compromise between full expressiveness of the graphical primitives and the ease of using them in specifications. It would be possible to come up with graphical primitives that achieve full expressive power of their corresponding textual primitives. However, the sacrifice would be that the graphical primitives are difficult to use. In such cases, we choose to limit the expressive power of the primitives to make them easier to use. For example, in the Clock textual language, it is possible to precisely denote a reference point and then make the positions of some other objects fixed relative to this point. We might design some graphical primitives to achieve this. However, such graphical primitives will be too complex to really apply them in real specifications. For the sake of ease of use, we choose a moderate alternative. In our approach, it is impossible to explicitly denote a reference point. Instead, to make the relative positions of a group of objects fixed, we use the grouping tool. In this approach, the inexplicit reference points are the lower-left and top-right vertex of all the grouped objects. This grouping tool has less expressive power than the textual "reference point" primitive in precisely specifying relative positions, but it serves as a practically

useful and much easier way to achieve similar effects.

5 Syntax and Semantics of the Mixed-form View Language

The mixed-form view language is a sub-language of Clock designed to specify the graphics part of user interfaces. It consists of both textual and graphical primitives. Graphical primitives are used to specify concrete graphical objects, whereas textual primitives are used to specify parameters, some control structures, and to denote variables. By mixing textual primitives and graphical primitives fluidly in a framework based on the functional paradigm, we aim to combine advantages of both textual programming and visual programming.

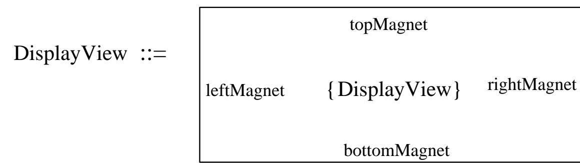
With the mixed-form view language, the Clock programmer can construct views of the interfaces in a direct manipulation manner. He/she may choose to draw graphical primitives to directly specify the views or may type in textual primitives to specify those aspects of the interfaces where graphical primitives are unsuitable, such as specifying numerical parameters. A graphical editor has been implemented to allow the programmer to compose view definitions using both textual and graphical primitives. After specification, the graphical primitives in a mixed-form program are translated into their corresponding textual primitives by the editor. The graphical editor acts as a mixed-form program editor as well as a preprocessor. It converts the mixed-form programs into textual code that can be accepted by the Clock compiler to produce executable code. Basically, each graphical primitive maps directly to one or a set of equivalent textual primitives. On the other hand, the textual language is more powerful. Thus, not every textual primitive has a graphical equivalent.

The following sections present the syntax and semantics of the mixed-form view language in Clock. The language is based on the functional paradigm and involves both textual primitives and graphical primitives.

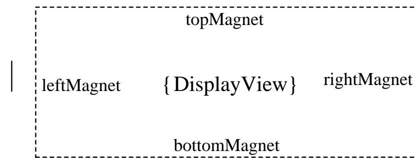
5.1 Syntax of the Mixed-form Language

The mixed-form view language is part of the Clock language designed for display view specification. It is a purely functional language involving both textual and graphical primitives. The syntax of the mixed-form view language is defined by the following grammar. For a complete specification of the syntax of the Clock language, please refer to [7].

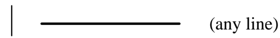
DisplayView ::=



|

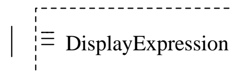


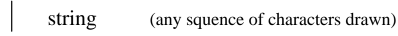
|



| string (any squence of characters drawn)

|





DisplayExpression ::= **Views** '[' [DisplayExpression] { , DisplayExpression } ']'

| **Style** style '(' DisplayExpression ')'

| **Font** font '(' DisplayExpression ')'

| **At** coord coord '(' DisplayExpression ')'

| **Box** '(' DisplayExpression ')'

| **Line** coord coord

| **Text** stringLiteral

coord ::= '(' ordinate, ordinate ')'

ordinate ::= **Left** offset | **Right** offset | **Top** offset | **Bottom** offset

| **XBaseOffset** ordinateLabel offset | **YBaseOffset** ordinateLabel offset

| **XSomewhere** | **YSomewhere**

style ::= **Invisible** | **Solid** | **Filled** | **Dashed** | **Dotted** | **Bold** | **Grey** | **White** | **Elipse** | **BoldElipse** |
Thick | **Inverted** | **LightGrey** | **NoBorderGrey** | **NoBorderLightGrey**

font ::= String

ordinateLabel ::= naturalNumber

offset ::= integer

topMagnet ::= ∧ | ⩚

bottomMagnet ::= ∨ | ⩛

leftMagnet ::= < | ←

rightMagnet ::= > | →

As can be seen from the above figure, the grammar contains nonterminals in both textual from and graphical form. The graphical nonterminals include a free-hand box, a canvas sign, a free-hand line, eight magnet signs and a function evaluation sign.

5.2 Semantics of the Mixed-form Language

In Clock, a view function definition describes the picture presented by an individual component of the user interface. Such component views are used to compose the view of the whole interface in a way specified in the architecture definition. A view function describes a mapping from some internal data structures to a picture. The Clock mixed-form view language allows the programmer to use graphical primitives as well as textual primitives in the specification of this map-

ping. During the execution of a Clock program, a view function is continuously applied to the current data state, providing a continuously updated display view. The mixed-form view language is purely functional, without side-effects.

In this section, we describe the semantics for the view language in Clock. For a description of the semantics of the full Clock language, readers are directed to [7].

Technically, a program that is composed of heterogeneous primitives has to be translated into some uniform representation before it can be compiled to generate executable code. This can be done either by converting the graphical primitives into corresponding textual primitives or by translating both forms of primitives into an intermediate, unified representation. We employ the first approach in translating Clock mixed-form programs. A view function definition is first preprocessed by the Clock graphical editor. All the graphical primitives appearing in the view function are translated into their textual equivalents. Consequently, the textual language forms the semantic basis of the graphical language. In this section, we explain the semantics of graphical primitives in terms of textual primitives. In our approach, the syntactic domain is made up of graphical primitives and the semantic domain is made up of textual primitives. A semantic function applied to one or a group of graphical primitives yields their corresponding textual equivalents.

5.2.1 Mapping from Graphical Primitives to Textual Equivalents

In the Clock mixed-form view language, graphical primitives serve merely as an alternative syntax to the existing textual forms. The semantics of the mixed-form

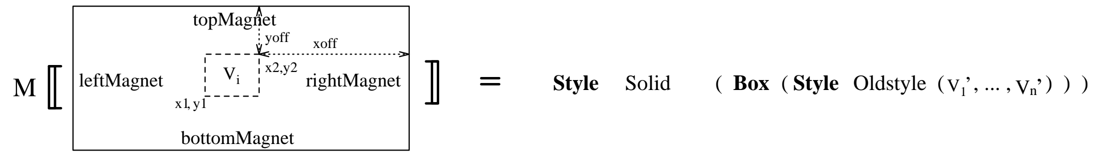
language is virtually based on the semantics of the textual language. Thus, the presence of a clear mapping from the graphical primitives to the corresponding textual primitives aids in the comprehension of the semantics. These mappings serve as meaning functions from the syntactic domain to the semantic domain. Each meaning function takes one or a group of graphical primitives and yields their meaning definition in terms of textual primitives.

$$M[\![\ldots]\!] : \text{syntex} \longrightarrow \text{semantics}$$

$$M[\![\ldots]\!] : \text{mixed-form language} \longrightarrow \text{textual language}$$

A correspondence mapping from graphical primitives to textual primitives is shown in the following:

Graphical Primitives	**Textual Primitives**



Annotation: The picture contains n *views named* $V_1, \ldots, V_n$

where

V_i' = **At** (leftOrdinate, bottomOrdinate) (rightOrdinate, topOrdinate) ($M [\![V_i]\!]$)

where

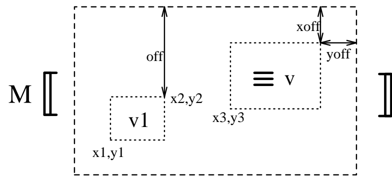
leftOrdinate = { **Left** x1 if leftMagnet = $\leftarrow$; **RightStretching** x1 if leftMagnet = $<$ }

rightOrdinate = { **Right** (-xoff) if rightMagnet = $\rightarrow$; **LeftStretching** x2 if rightMagnet = $>$ }

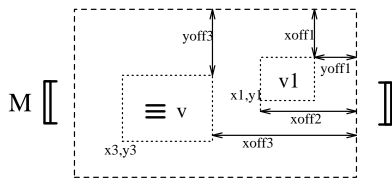
bottomOrdinate = { **Bottom** y1 if bottomMagnet = $\downarrow$; **TopStretching** y1 if bottomMagnet = $\vee$ }

topOrdinate = { **Top** (-yoff) if topMagnet = $\uparrow$; **BottomStretching** y2 if topMagnet = $\wedge$ }

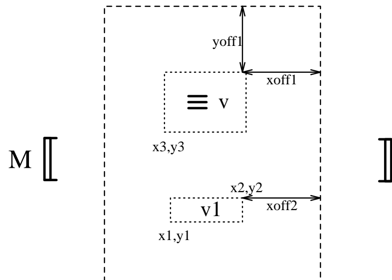
Grouping

M [[]]

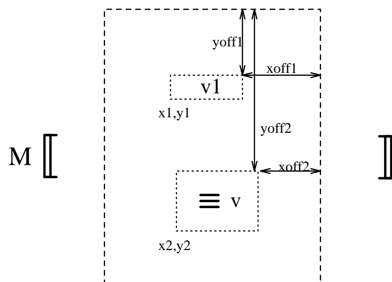
Views[
  At (Left x3, Bottom y3) (Right (-xoff), Top (-yoff)) (M [[v]]),
  At (Left x1, Bottom y1) (Left x2, Top (-off)) (M [[v1]])
].

M [[]]

Views[
  At (Left x3, Bottom y3) (Right (-xoff3), Top (-yoff3)) (M [[v]]),
  At (Right (-xoff2), Bottom y1) (Right (-xoff1), Top (-yoff1)) (M [[v1]])
].

M [[]]

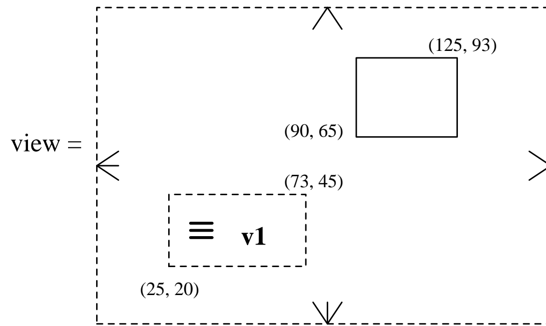
Views[
  At (Left x3, Bottom y3) (Right (-xoff1), Top (-yoff1)) (M [[v]]),
  At (Left x1, Bottom y1) (Right (-xoff2), Bottom x2) (M [[v1]])
].

M [[]]

Views[
  At (Left x2, Bottom y2) (Right (-xoff2), Top (-yoff2)) (M [[v]]),
  At (Left x1, Top (-yoff1)) (Right (-xoff1), Top (-yoff1)) (M [[v1]])
].

An empty canvas with no component views yields a null view, which is textually denoted as `View[]`. An empty box maps to `Box(Views[])` and the `Style`

primitive followed by a given style. The relative positions of views within a canvas or a box are affected by the states of the four magnets attached. If the magnet associated with an edge of the surrounding canvas or box is *on*, then the distance between the corresponding edge of a component view to the surrounding canvas or box is fixed. If the magnet is *off*, the corresponding edge will stay where it is drawn if the component view does not stretch. In case the component view stretches, the corresponding edge will stretch to the opposite direction. In the former case, the positioning constraint is mapped to **Left**, **Right**, **Top** or **Bottom** followed by an offset. In the latter case, the constraint is mapped to **LeftStretching**, **RightStretching**, **TopStretching** and **BottomStretching** followed by an offset. In the following example, we suppose *v1* is predefined view which is evaluated in a canvas:



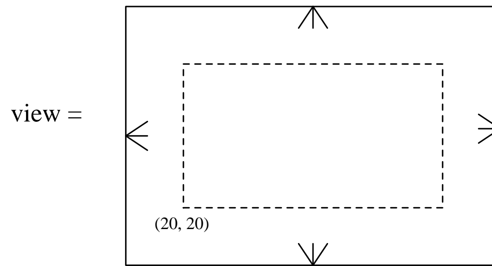
Graphical Specification

```
view =  Views [
          At  ( Left 25,  Bottom 20)  (RightStretching 73, TopStretching 45)  (v1),
          Style Solid (
              At  ( Left 90,  Bottom 65)  (RightStretching 125, TopStretching 93)  (
                  Box (Views[])
              )
          )
        ].
```

Corresponding textual specification

In addition to the view of *v1*, an empty box also appears. As the left and bottom magnets of the canvas are on, the positions of the lower-left corner of *v1* and the box are fixed relative to the lower-left corner of the canvas. Thus, if the lower-left vertex of the canvas moves, the *v1* and the box will move accordingly to maintain the fixed distances. The top and right magnets are off. That means if the *v1* evaluation sign is big enough to hold the actual view, it will appear as it is drawn. If it is not big enough, the view is allowed to stretch only to the top and right directions. The equivalent textual view specification is also given in the above figure.

As another example, a dashed box is drawn within an outer box in the following specification:



Graphical Specification

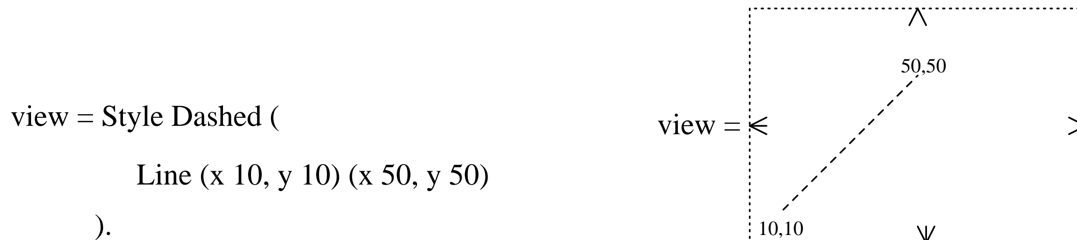
```
view = Style Solid (
            At ( Left 20, Bottom 20) (Right (-20), Top (-20)) (
               Style Dashed (
                        Box ( noView)
               )
            )
         ).
```

Corresponding textual specification

In this example, the four magnets are all on. Thus, the distance between every edge of the surrounding box to the corresponding edge of the inner box is kept fixed. If the outer box stretches, the inner box will stretch in the same way to maintain the constraints. The size of the inner box may change accordingly.

Due to the high-level features of the mixed-form language, it should be noticed that one-to-one mapping from graphical primitives to textual primitives is not always possible. A simple graphical primitive may convey more information

in specifying display views. Certain aspects of the specification are encoded implicitly in the graphical primitives, whereas in textual representation they have to be specified explicitly using additional textual primitives. For instance, the simple graphical **Line** primitive encodes not only specification of the two ending points of a line but also specification of the line style. By comparison, in the textual representation, an additional textual primitive **Style** is needed in addition to the **Line** to achieve the same effect. The following graphical and textual view function specifications display the same picture: a segment of dashed line. However, in the textual specification, an additional primitive is needed to specify the style of the line.



In several cases, one-to-multiple mapping from graphical primitives to textual primitives is necessary.

In general, every graphical primitive can be mapped directly to one or a set of textual equivalents. On the other hand, the textual language, however, offers more expressive power and flexibility. Thus, not all textual primitives necessarily has a graphical equivalent.

6 Mixed-form View Specification Examples

This chapter presents a series of simple user interface examples constructed using mixed-form programming. We use these examples to evaluate the expressiveness of the mixed-form view language in Clock.

The first example mimics a score board which is used to display scores of two teams in a football match. This example demonstrates that the mixed-form language makes it easier to achieve the desirable graphics.

The second example implements a simple arithmetic calculator. It illustrates how graphical primitives can be fluidly mixed with textual primitives in developing a user interface with the mixed-form view language.

The third example implements the interface of a new audio facility BBE on mini audio systems. This example demonstrates, from another point of view, the benefits of combining graphical and textual primitives seamlessly in a language for user interface development.

6.1 A Score Board

This score board example is created to demonstrate how the availability of graphical primitives in the mixed-form language can ease the task of obtaining desired graphics in constructing a user interface. The appearance of this example is shown in Figure 29. The interface contains two huge digit displays showing the scores of two teams in a sports match. There are two buttons labeled “+1” below the score display. If one team scores, the display can be updated by pressing the

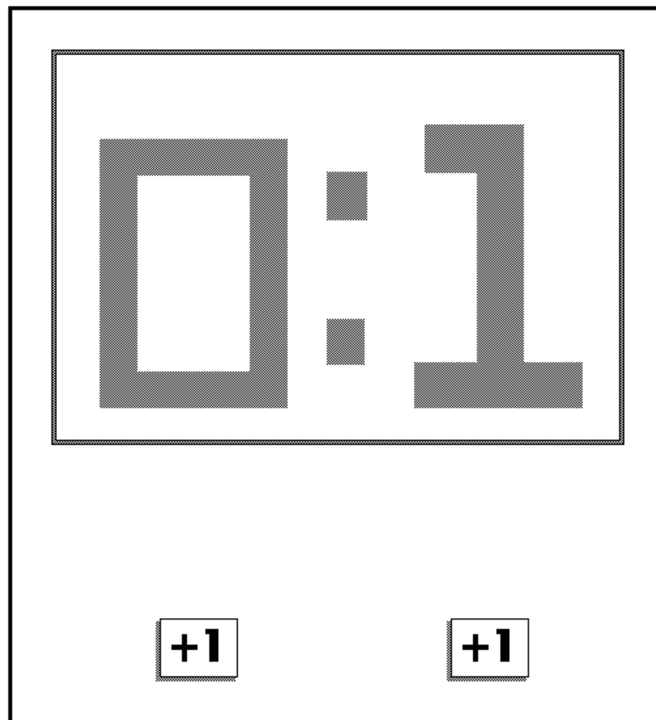


Figure 29: A simple score board example. The interface contains two huge digits showing the scores of two teams in a sports match. Two buttons labeled "+1" are used to update the scores. As no font is big enough to display characters as huge as that, we have to piece together gray boxes to compose the digit displays.

button under the score of that team. The key problem in this application is that no font is big enough to display such huge characters. A simple solution to this problem would be to piece together gray rectangles to compose the huge digit displays.

To specify such views using a purely textual language, the programmer has no choice but to draw the digit pictures on graph papers and carefully calculates the coordinates for each rectangle. Then, he/she has to translate the pictures into sequences of textual commands. It is easy to make mistakes in this process,

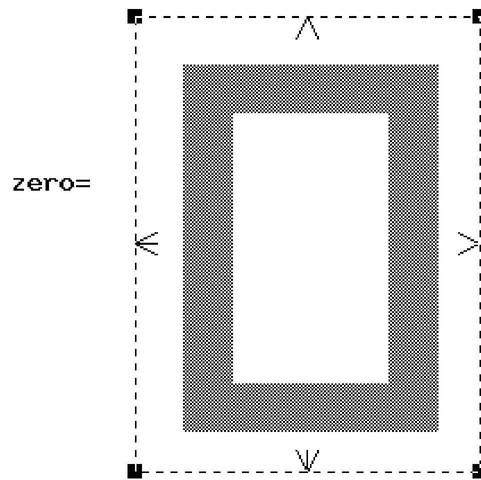


Figure 30: The graphical specification for digit "0".

since it demands lots of attention and patience. A single modification of the picture specification requires re-compilation and execution of the program to see the actual effect. The programmer has to go through a significant number of such iterations to obtain the satisfactory graphics. The Clock mixed-form view language integrates a set of graphical primitives. Using the graphical editor, the programmer can directly specify the picture by drawing graphical primitives.

Specific to this example, the user can draw several grayed boxes and compose them to a digit display. The graphical view function specification directly reflects the actual picture in a WYSIWYG way. Thus, the programmer can constantly resize and move these grayed boxes to achieve the desired graphics without having to compile and execute the specification. As an example, the graphical specification for digit "0" is shown in Figure 30. The equivalent tex-

tual code translated by the Clock graphical editor is shown in the following:

```
zero = Style NoBorderGrey (
   Views[
      At (Left 142, Bottom 22) (LeftStretching 170, BottomStretching 226) (Box(noView)),
      At (Left 27, Bottom 22) (LeftStretching 55, BottomStretching 226) (Box(noView)),
      At (Left 55, Bottom 199) (LeftStretching 142, BottomStretching 226) (Box(noView)),
      At (Left 55, Bottom 22) (LeftStretching 142, BottomStretching 49) (Box(noView)),
        ]
).
```

In this example, all the digit displays are first specified graphically using the graphical editor, and then translated into the equivalent textual code.

This score board example illustrates that it is easier to obtain desirable graphics of a user interface using the mixed-form language than using a purely textual language, since the incorporated graphical primitives bridges the conceptual gap. However, as the graphical primitives for composing pictures are limited to simple lines and rectangles with a limited number of fill patterns, the graphics we are able to generate is very simple. This is partially due to the limitations of the underlying textual view language. Nevertheless, as incorporating more fancy graphical primitives is just a problem of time, such limitations do not hinder the demonstration of our ideas.

6.2 An Interactive Calculator

The next example is an interactive arithmetic calculator. This calculator application allows the user to perform simple arithmetic operations. The user can enter

operands by pressing the ten digit buttons or enter operators by pressing the five available operator keys. A "CA" key is used to reset the calculator. Figure 31 shows the user interface of the calculator. This example is used to demonstrate the advantages of fluidly mixing graphical and textual primitives in constructing user interfaces using the Clock mixed-form view language.

The architecture diagram is shown in Figure 32. As can be seen from this diagram, the interface of the application is decomposed into "dispad" and "keypad". The "keypad" consists of "numkey" and "operkey" subviews. These subviews are furtherly decomposed into smaller subviews.

The "nkey" subview is responsible to display a box surrounding a digit character specified by the "Id" request handler associated with it. The mixed-form view specification for "nkey" is shown in Figure 33. Function "t" is used to obtain the digit character in "hugeBold" font. The "ptext" function adds some space around a display view "v" for whatever "v" is. Thus, "ptext t" yields a huge bold digit character with some space around it. And the view function simply adds a box to the inside display view.

The "nkitem" function adds a shadow and operational behavior to the "nkey". Figure 34 shows the mixed-from specification of function "nkview" which defines the appearance of the button when it is not pressed.

The "numkey" subview contains three numerical key rows and a single "0" key. Each key row is obtained by mapping the "nkitem" subview to a list of three digit characters. Then, a certain amount of space is inserted among each key in the list. The piece of textual code for getting the key rows is shown below.

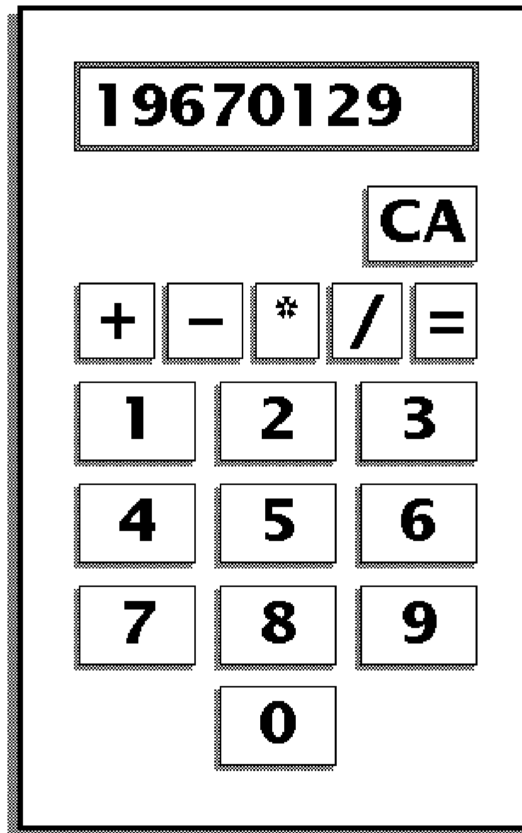


Figure 31: An interactive arithmetic calculator. This calculator application allows the user to perform simple arithmetic operations. The user can enter operands by pressing the ten digit buttons or enter operators by pressing the five available operator keys. A "CA" key is used to reset the calculator.

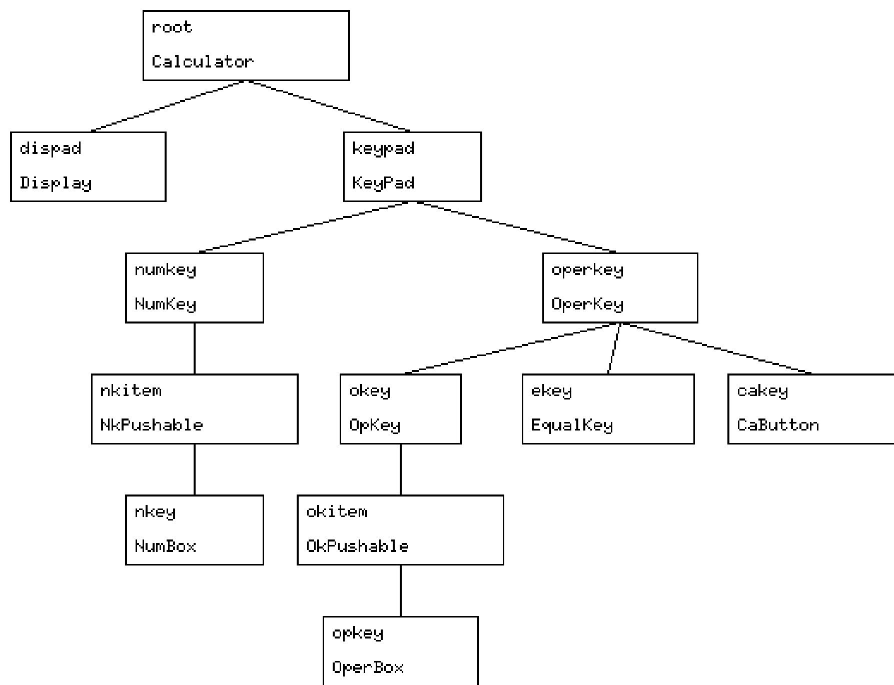
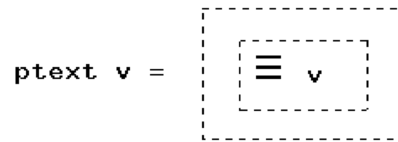


Figure 32: The architecture diagram for the interactive calculator example.

```
t = Font hugeBoldText (Text myId).
```



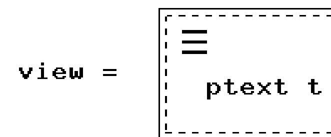


Figure 33: The mixed-form view specification for subview “nkey”.

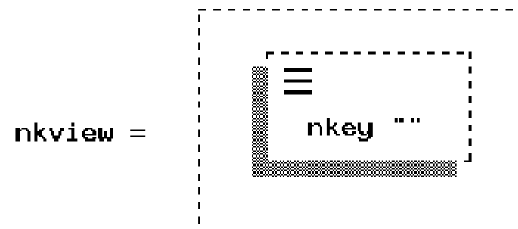


Figure 34: The mixed-form view specification for “nkview” which adds a shadow to “nkey”.

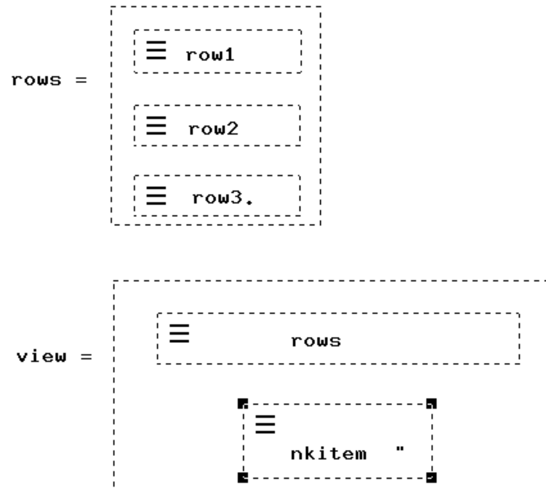


Figure 35: The mixed-form view specification for subview "numkey".

```
digits1 = ['1', '2', '3'].
digits2 = ['4', '5', '6'].
digits3 = ['7', '8', '9'].

keyrow1 = map nkitem digits1.
keyrow2 = map nkitem digits2.
keyrow3 = map nkitem digits3.

row1 = beside(insertBetween keyrow1 (space 10)).
row2 = beside(insertBetween keyrow2 (space 10)).
row3 = beside(insertBetween keyrow3 (space 10)).
```

These three key rows are then put one above another to compose the "numkey" subview. The graphical view specification for "numkey" is shown in Figure 35.

The "dispad" subview is responsible for displaying the operands and the results. The operands and results can be obtained from the "getReg1" request.

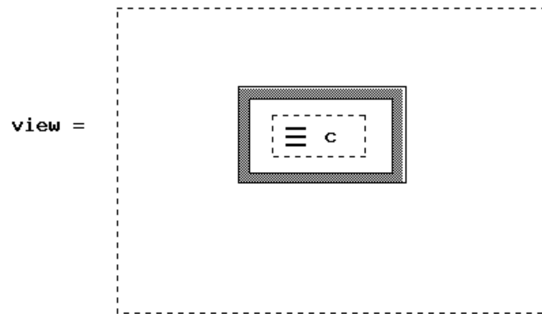


Figure 36: The mixed-form view specification for subview "dispad".

"NumText getReg1" converts the numerical value into a display view of a piece of text. The mixed-form view specification for "dispad" is shown in Figure 36.

Similarly, we can get the "keypad" subview. By combining the "dispad" and the "keypad", we eventually get the whole interface of the calculator.

This calculator example illustrates that we can fluidly mix the graphical and textual primitives in user interface specifications. Being able to mix graphical and textual primitives, we can combine the advantages of both forms of primitives. Specific to this example, we use textual primitives to map a button subview to a list of digits to get a row of buttons. Drawing such a row of buttons one by one using graphical primitives will be boring and inconvenient. On the contrary, when dealing with the relative positions or sizes of component views, we choose to draw graphical primitives to avoid numerical coordinates.

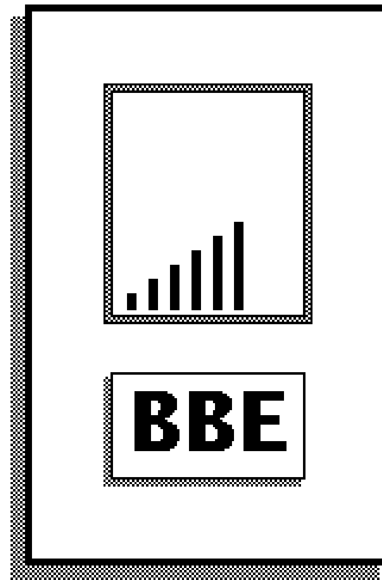


Figure 37: The interface of BBE audio facility

6.3 A BBE facility

BBE is a new audio technique invented by the BBE Sound Ltd. This technique is used to remedy high frequency sound distortion. Thus, the sound coming from an audio system with BBE is crystal clear. In the past, professional audio systems used to have this BBE facility. Now, it is becoming a standard feature for customer audio systems. The example presented here implements the interface of the BBE facility in mini systems.

The interface of the BBE is shown in Figure 37.

The simple interface contains a BBE level display and a BBE button. The user can press the BBE button repeatedly to select one of the four levels, or the off position to suit his preference. A series of small bars of increasing heights are displayed to reflect the BBE level.

In this example, the view function specifications for the shadow, the grayed border and the button are all similar to the previous examples. However, the specification for the BBE display deserves some extra attention. Two small bars of increasing heights are used to represent each BBE level. According to the level value, these small bars are dynamically displayed. The piece of code for the small bars is shown in the following:

```
bar h = Style Filled (At origin ( x 3, y (6*h)) (Box(noView))).

off = bar 0.
level1 = beside(insertBetween  (map bar [1,2]) (space 5)).
level2 = beside(insertBetween  (map bar [1,2,3,4]) (space 5)).
level3 = beside(insertBetween  (map bar [1,2,3,4,5,6]) (space 5)).
level4 = beside(insertBetween  (map bar [1,2,3,4,5,6,7,8]) (space 5)).
```

In this specification, function "bar" yields a small filled rectangle of a certain height affected by the parameter "v". By mapping this bar function to a list of integers, we can get a series of bars of increasing heights. For example, "map bar [1,2,3,4]" yields four bars of heights 6, 12, 18 and 24 respectively. There are two points in this specification where graphical primitives are not suitable. First, it is impossible to specify the numerical parameter "v" graphically in the definition of the "bar" function. Secondly, though possible, it is cumbersome to specify the series of bars using graphical primitives. What we can do is to draw a series of

filled boxes one by one. On the other hand, for some static parts of the interface display, such as the box, the border and shadow, we can conveniently use graphical primitives to specify them. The specifications remain to be intuitive, easy to comprehend and easy to modify. In this way, we combine the merits of both textual and graphical primitives.

This BBE example presents a case that the graphical primitives are difficult, if not possible, to handle (dynamic pictures). This is one of the inherent problems of purely visual languages and interface builders. However, with the Clock mixed-form language, we can get around with this problem by using textual primitives. This example reflects, from another facet, the benefits of combining both graphical and textual primitives in one language.

6.4 Conclusion

Through a series of simple user interface development examples using mixed-form programming in Clock, this chapter demonstrates the benefits of combining textual and graphical primitives fluidly in a single language. As graphical constructs whose size, location and appearance can be directly manipulated are readily available in the language, the developer can easily achieve the desired graphics free of coordinates and the difficult conversion process from picture to textual code. The equal treatment of text code and pictures in the language ensures a fluid mixture of the two in mixed-form programs. The problems caused by sharp division between graphical and textual parts of the program are avoided. The developer can freely pick up either form of primitives where appropriate. Generally speaking, graphical primitives are natural for picture specification while textual primitives

are more appropriate for specifying dynamic and functional part of the program.

7 Summary and Conclusion

This chapter first provides a summary of the work presented in this thesis. Some implications are then discussed. The final section suggests directions for further work.

7.1 Thesis Summary

The purpose of this thesis is to demonstrate that mixing textual and graphical primitives within a functional language framework provides a useful alternative to purely textual or visual languages in graphical user interface development. The thesis aims to incorporate a set of graphical primitives, whose visual attributes and geometric arrangement can be directly manipulated to compose the desirable view, into the Clock view language. A prototype graphical editor has been developed to support the mixed-form program editing. A series of user interface examples using the mixed-form view language are provided to show its expressiveness and convenience.

7.1.1 Problems with Purely Textual or Visual Languages

The task of developing user interfaces is not a trivial one because of their highly graphical and interactive nature. User interface development requires powerful facilities to support the design of the interface graphics and to ease the task of programming the control structure which is much different from that in batch-based applications. Lots of constraints among objects on the interface and between the interface and the application parts ought to be maintained by the system rather

than by explicit programming. It is ideal to merge all these facilities into a single user interface development language.

Traditional textual languages are handicapped in composing and specifying the graphics part of user interfaces. Textual primitives give poor representation of graphics. The big gap between graphics and their textual representation mandates an unintuitive and error-prone conversion process. This conversion process usually needs to be repeated a significant number of times in order to obtain the desirable pictures.

Visual Languages provide graphical primitives to construct programs. Graphical primitives are intuitive and appropriate in specifying the graphics part of user interfaces as they offer more direct representation. However, graphical primitives are very limited in expressing complex program control structures.

Some user interface builders employ a hybrid approach. They provide a tool through which graphical objects can be directly manipulated and arranged to compose the desirable pictures. The graphical part is then somehow connected to the application code. There is a sharp separation between the interface graphics and the code that implements the application's functionalities. The highly interactive nature of graphical user interfaces implies substantial interactions between the graphics part and the functional part. The number of constraints involved is significant. The sharp separation makes the connection between graphics and code a bottleneck and leaves the constraint maintenance to explicit programming by the developer. Moreover, user interface builders are incapable of specifying dynamic interfaces in which the number and appearance of the graphical com-

ponents are changeable at run time.

7.1.2 The Mixed-form Visual/Textual Language

This thesis aims to combine both textual and graphical primitives in the framework of a functional programming language. It is argued that a language with textual and graphical primitives fluidly incorporated in a functional paradigm serves as a promising candidate for user interface development language.

The main motivation for the work presented in this thesis was based on the following observations. Graphical primitives are appropriate for specifying pictures and geometrical constraints. Textual primitives are appropriate for specifying control structures. Though separating the interface part and the functional part is necessary, a seamless connection between the two is mandated by the highly interactive nature of user interface software. Such a connection should be maintained by the system rather than by low-level explicit programming.

A set of graphical primitives were identified and incorporated into the Clock view language. Thus, the developer can freely pick up either form of primitives where appropriate in user interface specifications. The functional style of the Clock language ensures equal treatment of pictures and textual code. Display views are treated as first class values. Thus, a picture can be inserted wherever a common expression is allowed. On the other hand, textual expressions can be inserted into pictures as long as their evaluations yield pictures. The underlying functional paradigm enables fluid mixture of textual and graphical primitives. The mapping constraints between system states and the interface as well as ge-

ometric constraints among interface objects are all maintained by the system. The developer only needs to use either textual or graphical primitives to specify the expected results. How those results are achieved is left to the system.

A prototype graphical editor was developed to support mixed-form program editing. The developer can directly manipulate graphical primitives and mix them with textual primitives to specify graphical components of the interface and their geometric constraints. The editor also serves as a preprocessor to the mixed-form program. It translates all the graphical primitives in a mixed-form program into their textual equivalents.

The mixed-form language surpasses purely textual languages in easy graphics specification. It is superior to purely visual languages in specifying behavior of the interface. Compared to other hybrid approaches, it supports an efficient and system-maintained connection between the graphics part and the functional part of the interface. Moreover, the mixed-form language is capable of specifying dynamic interfaces whose number of components is changeable at run-time.

7.2 Future Works for the Clock System

This section discusses possible extensions of the work presented in this thesis.

The mixed-form Clock view language is based on the underlying textual view language. The previous version of the textual view language has a very limited set of primitives to handle graphics. Thus, the mixed-form view language is also limited to specifying simple pictures with only lines, boxes, fonts and fill patterns. Fortunately, the underlying textual view language has been constantly

augmented with more primitives to handle fancy graphics. The mixed-form view language can be extended by adding in graphical equivalents of such primitives.

A programming tool, ClockWorks [17], has been developed to support the task of creating and manipulating Clock architecture trees. The graphical editor can be integrated into that programming environment. Thus, the developer can choose a node in an architecture tree and invoke the editor to specify its view function. Ideally, the developer should also be allowed to click a node and see the picture it represents.

The graphical editor is implemented in C and X Window/Motif. It involved a lot of explicit programming to maintain geometric constraints among graphical objects and the mapping from a mixed-form program to its internal representation. Many of the difficulties in using imperative programming to maintain constraints have been experienced. As the Clock system supports automatic constraints maintenance, it would be a very interesting task to implement the editor using the Clock language. A successful and easier implementation would demonstrate the powerfulness of the Clock methodology.

7.3 Conclusion

This chapter summarizes the work of this thesis. It is argued that a functional language with fluidly integrated textual and graphical primitives is a good candidate for user interface development language. The last section discussed some possible areas for further research.

References

[1] Alan Borning. Defining constraints graphically. In *Human Factors in Computing Systems, CHI 1986 Proceedings*, pages 137–143, 1986.

[2] James R. Cordy and T.C. Nicholas Graham. GVL: Visual specification of graphical output. *Journal of Visual Languages and Computing*, pages 25–46, 1992.

[3] Joelle Coutaz. Pac, and object-oriented model for dialog design. In *Proceedings of INTERACT'87*, pages 431–436, 1987.

[4] Ephraim P. Glinert and Steven L. Taminoto. Pict: an interactive graphical programming environment. *IEEE Computer*, 17(11):7–25, November 1984.

[5] D. Goodman. *The Complete HyperCard Handbook.* Bantam Books, New York, 1987.

[6] T.C. Nicholas Graham. Conceptual views of data structures as a programming aid. Technical Report 88-225, Department of Computing and Information Science, Queen's University at Kingston, August 1988.

[7] T.C. Nicholas Graham. *Declarative Development of Interactive Systems.* Oldenbourg Verlag, 1995. (to appear).

[8] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[9] Scott Hudson. How programming languages might better support user interface tools. In Brad A. Myers, editor, *Languages for Programming User Interfaces*, chapter 7. Jones and Barlett, 1992.

[10] NeXT Inc. *NeXTSTEP Reference.* Addison-Wesley, 1991.

[11] ParcPlace Systems Inc. *VisualWorks User's Guide.* ParcPlace Systems, 1992.

[12] Scott A. Kirkwood. Programming with pictures as a base data-type. Master's thesis, Queen's University, Kingston, Ontario, May 1993.

[13] Glen E. Krasner and Stephen T. Pope. A cookbook for the using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[14] Mark A. Linton, John M. Vlissides, and Paul R Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[15] Deke McClelland. *Drawing on the Macintosh.* Publishing Resources Inc., 1992.

[16] Microsoft. *Visual Basic Programmer's Guide.* Microsoft Press, 1993.

[17] Catherine A. Morton. Tool support for component-based programming. Master's thesis, Department of Computer Science, York University, Toronto, Canada, June 1994.

[18] Brad A. Myers. Why are human-computer interfaces difficult to design and implement. Technical Report CMU-CS-93-183, Computer Science Department, Carnegie Mellon University, July 1993.

[19] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Human Factors in Computing Systems*, Monterrey, CA, May 1992.

[20] Donald A. Norman. *The Design of Everyday Things*. Doubleday Currency, 1990.

[21] Tore Urnes. A relational model for programming concurrent and distributed user interfaces. Master's thesis, Norwegian Institute of Technology, University of Trondheim, 1992.

[22] Douglas A. Young. *X Window System: Programming and Applications with Xt, OSF/Moti f Edition*. Prentice-Hall, ISBN 0-13-49704-8, 1990.