

# Design and Implementation of a Hard Real-Time System Design Tool

By Lihua Zhao

A research paper submitted in partial fulfillment of the  
requirements for the degree of Master of Arts in  
Interdisciplinary Studies

Major Professor : Dr. T. G. Lewis

Department of Computer Science  
Oregon State University  
Corvallis, Oregon 97331

November 16, 1993

# Design and Implementation of a Hard Real-Time System Design Tool

Lihua Zhao

Computer Science Dept.  
Oregon State University  
Corvallis, OR 97331-3902  
zhaol@storm.cs.orst

## Abstract

We describe a CASE tool for designing hard real-time applications, called HaRTS. The design tool supports a hierarchical design diagram which combines the control and data flow of a hard real-time application. The design hierarchy separates a design into self-contained subdesigns. Yet, the design can be flattened to give you a global view. In a distributed environment, the hierarchy provides a natural way for assigning subdesigns to different processors. The design diagram is quite intuitive, and yet it can be automatically translated into Ada™ code and analyzed for scheduleability.

This design tool has been implemented on Macintosh using an object-oriented application framework, called Objex. Our experience has demonstrated that such an object-oriented framework is quite useful in developing GUI applications like HaRTS. The framework facilitates us in developing object-oriented code which has better understandability. Essentially, every object you see on the screen has a direct correspondent in the program. This helps both programmers and maintainers understand the dynamic behavior of the program. The software reusability of Objex, supported through inheritance and dynamic binding, provides a powerful mean for reducing software development cost and improving software quality.

## Table of Contents

1 Introduction .....	1
1.1 The Design Tool – HaRTS .....	3
1.2 Implementation Environment .....	4
1.2.1 Objex.....	4
1.2.2 The User-Machine Interaction Model .....	7
1.3 Deriving HaRTS .....	9
2 The HaRTS Design Diagram .....	10
2.1 Basic Design Components .....	10
2.2 More Design Components .....	15
2.2.1 Control Operators and Guards.....	15
2.2.2 Data Operators.....	17
2.3 Flattening a Design .....	18
3 The Implementation of HaRTS .....	20
3.1 The HaRTS User Interface .....	20
3.1.1 Creating New Designs .....	24
3.1.2 Decomposing a Composite Box.....	26
3.1.3 Flattening a Composite Box .....	29
3.2 The HaRTS Class Hierarchy .....	30
3.2.1 HaRTS Data Model Class Hierarchy .....	31
3.2.2 HaRTS Shape Class Hierarchy .....	37
3.2.3 The Scene Class.....	40
3.2.4 The GraphicsView Class .....	42
4 Conclusion and Future work .....	45
References .....	46
Acknowledgements .....	48

## 1 Introduction

This report describes work that is based on the design diagram for hard real-time applications in [11]. The author designed and implemented the user interface and storage structures of a hard real-time system design tool, called HaRTS, which supports the design diagram. The implementation used the Objex framework [6, 8, 9] and as a result demonstrated that Objex works.

In a hard real-time system, the computer periodically gets information from the environment through sensors, updates its internal system states based on the inputs and the current internal states, and generates control commands to change the environment through actuators.

The single most important requirement for a hard real-time system is that it must make correct response to environmental changes within specified time intervals, called *deadlines* [1, 4, 5]. Those deadlines, which by the design specification must absolutely be met by the operational system, are called *hard deadlines*. Missing any hard deadline can lead to catastrophic results. Nuclear power plant control, missile control, etc., are examples of such applications. Due to the characteristics of hard real-time applications, deterministic system behavior is a must.

A hard real-time software system consists of a set of state machines, also called tasks or processes, which cooperate to achieve the system goal. These state machines are responsible for reading the inputs from environment, storing and updating the internal system states, and generating control commands to control the environment. Each state machine is composed of a subset of the system states (data) and an algorithm for the state transformation. We call these state machines tasks.

There are two kinds of tasks in hard real-time software: periodic and sporadic (aperiodic) tasks. During run time, all tasks can be in only one of two states: active or inactive. The tasks in the active state are eligible to be scheduled to execute. The tasks in the inactive state are not. A periodic task enters its active state periodically. On the other hand, a sporadic task becomes active by responding to some event. An active task returns to its inactive state after it

finishes one execution. We shall use the term request or trigger to mean that an inactive task enters its active state. Each time a task is executed, we say an instance of it is generated. A task meets its deadline if only if all of its instances meet their deadlines.

## 1.1 The Design Tool -- HaRTS

We build a graphical design tool for hard real-time applications, called HaRTS, which supports a hierarchical design diagram which combines control flow and data flow [11]. The combination enables us to easily obtain a whole picture of a hard real-time application, which is difficult to achieve through examining separate control flow diagram and data flow diagram.

The design method supported by HaRTS is different from earlier graphical design methods in many aspects [11]. Traditional graphical software design methods normally separate control flow from data flow [12]. Nor do they support the strict timing requirements of hard real-time applications. [1] introduced a graphical computation model which supports strict timing constraints. However, it is still based on separate control flow diagram and data flow diagram.

The design hierarchy separates a design into self-contained subdesigns. Yet, the design can be flattened to give you a global view. When one processor can not satisfy the timing and precedence requirements of a design, the hierarchy provides a natural way for assigning subdesigns to different processors in a distributed environment. The design diagram is quite intuitive, and yet it can be automatically translated into Ada™ code and analyzed for scheduleability [11].

It should be pointed out that our design methodology is based on the needs of real applications. For example, a missile flight control application played a major role in the design of HaRTS. We shall use the simplified version of this application as an example in our presentation.

Developing a hard real-time application with deterministic behavior has been a difficult problem. Under the traditional cyclic-executive approach [5], programmers need to unnaturally cut the code into certain sized pieces that fit into time frames of a schedule. Putting code pieces by hand into the right time frames and in the right order is a time-consuming and error-prone process [4]. This painful process must be repeated when the code is modified or

updated. On the other hand, under our approach, a hard real-time application is developed by "What You See Is What You Get". The design requirements are captured by the design diagram which can be automatically analyzed and scheduled.

## 1.2 Implementation Environment

Our hard real-time design tool has been implemented using an object-oriented application framework called Objex. In this section, we briefly describe Objex and a user-machine interaction model adopted by Objex, called Model-View-Control model.

### 1.2.1 Objex

The best known method of improving programmer productivity is to reuse existing code rather than reinvent it [12]. Application framework is one of approaches to achieving this goal.

Objex was designed and implemented by a team in Oregon State University [8]. Figure 1.1 shows an overview of Objex. Objex consists of three parts: the application framework classes, the data structure class library, and the shape class library. They are all built on top of the Macintosh Toolbox. Figure 1.2 shows the class hierarchy of Objex. Figure 1.3 and 1.4 are the data structure class hierarchy and the shape class hierarchy, respectively.

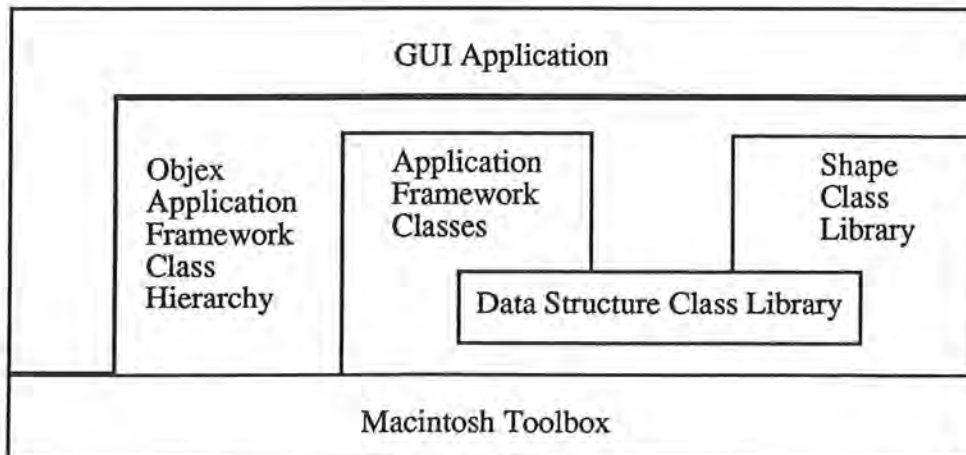


Figure 1.1 The Objex application framework architecture

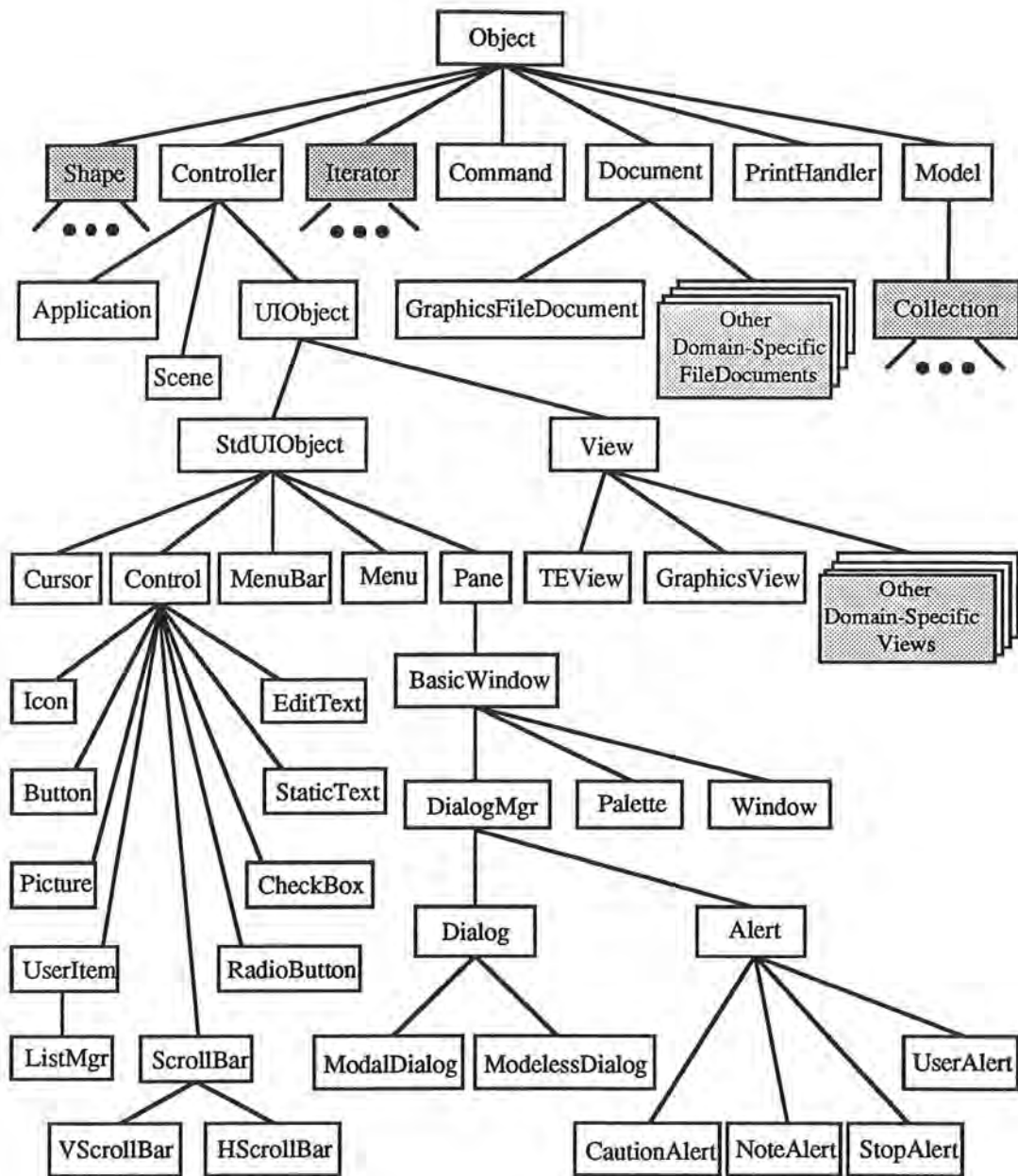
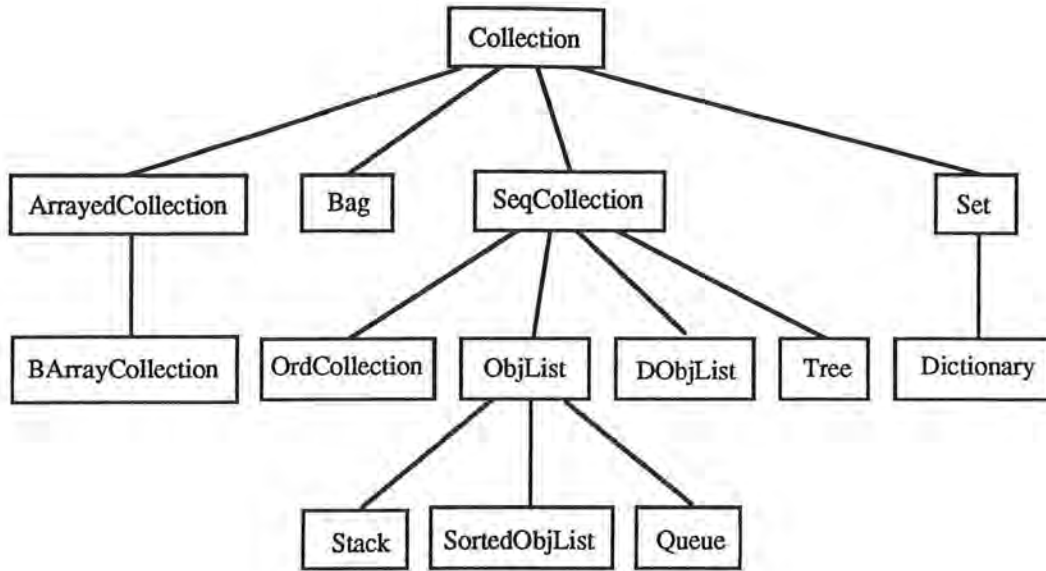


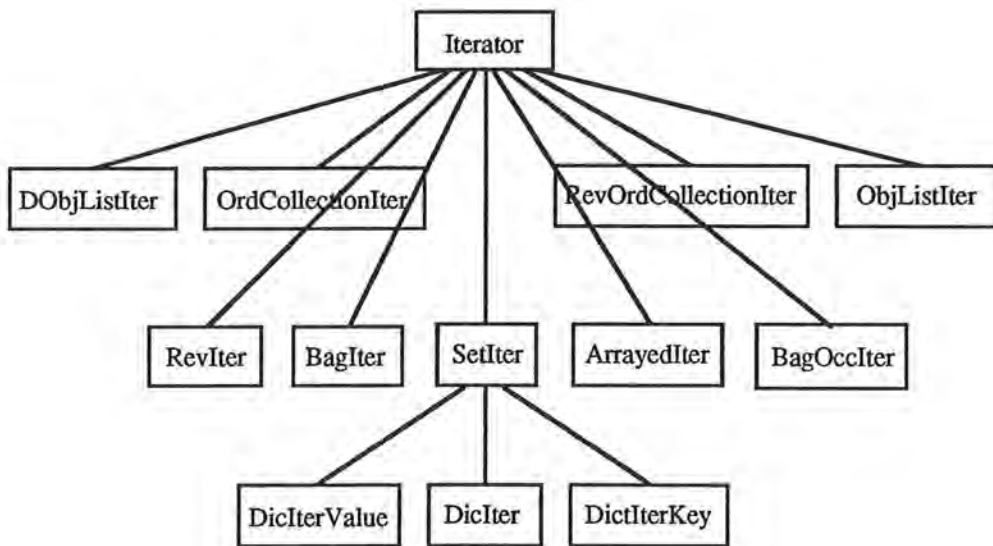
Figure 1.2 The Objex application framework class hierarchy

The application framework classes define much of a Macintosh application's standard user interface, generic behavior, and operating environment. Note that the root of the data structure class hierarchy in Figure 1.3 is the Collection class in Figure 1.2 and the root of the shape class hierarchy in Figure 1.4 is the Shape class in Figure 1.2.





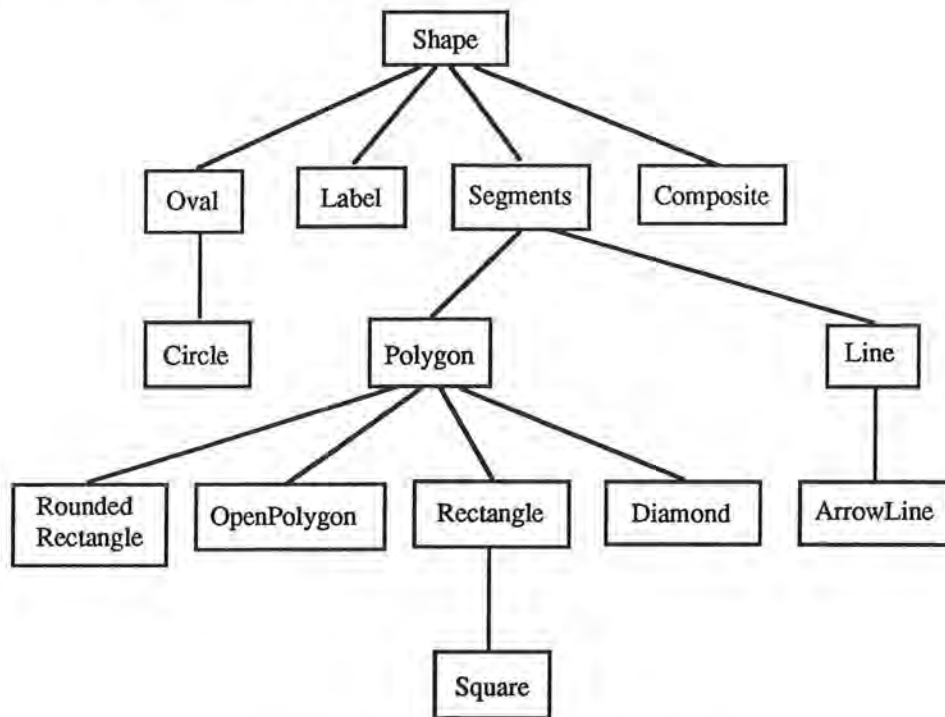
**Data Structure Class Hierarchy**



**Iterator Class Hierarchy**

**Figure 1.3** The data structure and the corresponding iterator class hierarchy

The data structure class library supports general data structures, such as array, list, set, stack, and queue. Corresponding to the data structure class hierarchy, there is an iterator class hierarchy. The iterator for each collection object is a mechanism used to inspect each element in the collection [6]. It performs some basic operations such as inserting, retrieving, etc.. Each collection class has a corresponding iterator.



**Shape Library Class Hierarchy**

**Figure 1.4** The shape class hierarchy

The shape library supports various different kinds of graphical shapes, such as rectangle, circle, line, oval, etc.. Each shape class provides the methods for creating, drawing, growing, etc., the corresponding shapes.

### 1.2.2 The User-Machine Interaction Model

Objex adopts a user-machine interaction model, called the Model-View-Controller (MVC) model, which is first used in the Smalltalk-80 environment [10]. The MVC paradigm handles a set of interactive objects from three related

classes: Model, View and Controller. The Model class (object) contains the domain specific data structure manipulated by a GUI application. The View class (object) renders all or parts of the domain specific data on the screen. The controller class (object) is responsible for accepting asynchronous inputs from users and passing appropriate messages to the model and view objects [8]. Figure 1.5 illustrates message passing in the MVC paradigm. Views and controllers can have only one model, but models may have many views and controllers. Views and controllers are generally tied closely together.

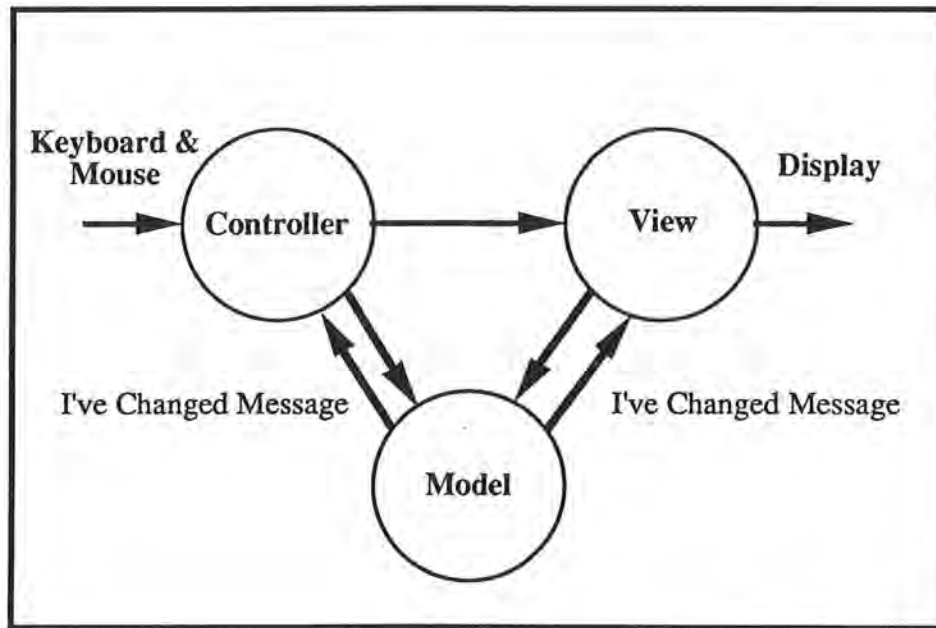


Figure 1.5 Model-View-Controller Communication

When multiple views of the same data model need to be manipulated simultaneously, the power of the MVC paradigm is realized. This is illustrated by the example in Figure 1.6 in which the data model has multiple views. The change made to the data under one view is automatically reflected in the other views. Existing views can be modified and new views can be added at any time .

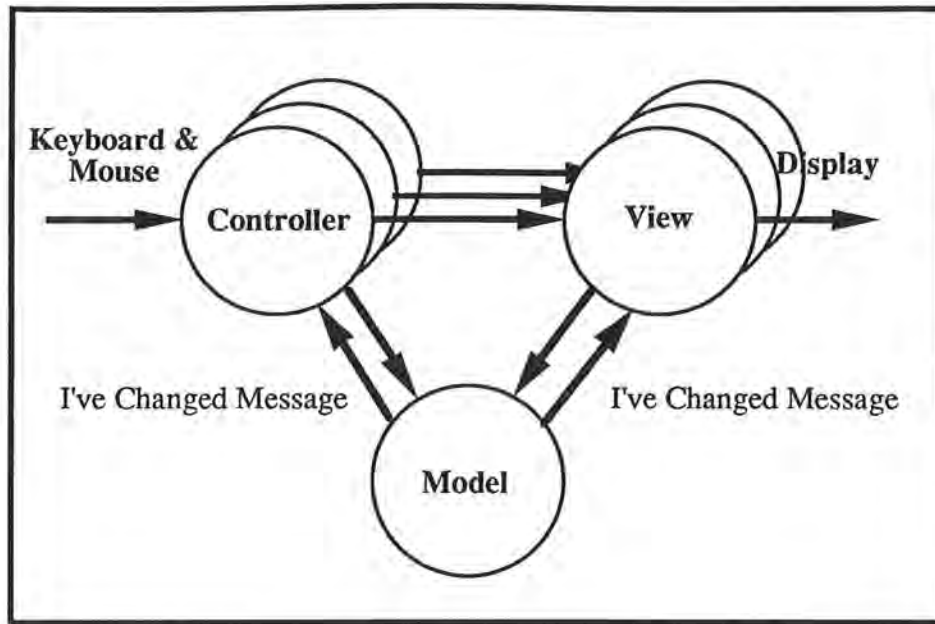


Figure 1.6 A Data Model with Multiple View-Controller Pairs

### 1.3 Deriving HaRTS

HaRTS is derived from Objex through the process of specialization, i.e., subclassing, overriding and extending the existing classes. For example, we subclassed the Rectangle class to create a new class called MyRectangle. The reason for the subclassing will become clear as we move on.

In our implementation of HaRTS, the reusability of Objex is not only reflected in reusing code, but also in reusing design. For example, inheriting the MVC user-machine interaction model for HaRTS has greatly reduced our effort in design and programming. Like all Macintosh applications, HaRTS supports the standard Macintosh user interface, such as windows, menus, dialogs, etc., and generic application features, such as open/save a file, undo/redo a command, print a window, etc..

The rest of this report is organized as follows. In section 2, we introduce the HaRTS design diagram. In section 3, we fully describe the implementation of HaRTS. In section 4, we conclude this report and suggest the future work.

## 2 The HaRTS Design Diagram

In this section, we describe the HaRTS design diagram. We first introduce the basic design components of the design diagram and the design hierarchy through an example application. Then, we describe the rest of the design components. Finally, we show that a hierarchical design can be flattened to give the global view of the design.

### 2.1 Basic Design Components

The design of a hard real-time application can be represented as a set of boxes, arrows, operators and associated text which together define its control flow, data flow, and timing constraints. Figure 2.1 shows the basic components of a design diagram. The box represents a system state transformation function. The control-in arrow on the top carries the control stimuli flowing into the box and the control-out arrow on the right carries the control stimuli flowing out of the box.

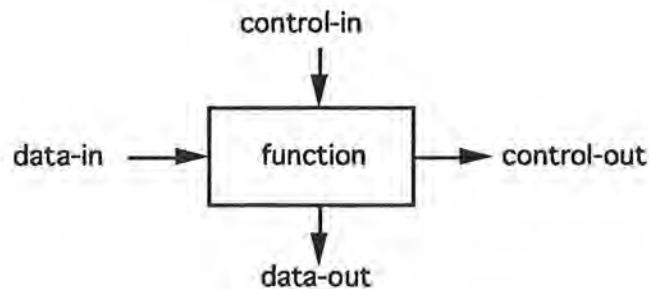


Figure 2.1: The basic components of our design diagram.

The control stimulus flowing into the box triggers the function to execute, and when the execution finishes, a new control stimulus is generated and flows out to trigger other functions to execute. The control stimuli of an application are generated either by the *control driving sources* (i.e., timers or external events), or by completing the execution instances of a box. Once a control stimulus is generated, it flows along the control arrow until it reaches a box or a control operator.

The data-in arrow on the left carries the data flowing into the box when the function is executed and the data-out arrow on the bottom carries the data flowing out of the box when the execution finishes. Each data arrow has an associated variable which represents the data store for the corresponding system state. There may be more than one data-in/data-out arrow attached to a box.

The design components are organized hierarchically as shown in Figure 2.2. The boxes with thinner borders are called *atomic boxes* and represent functions. On the other hand, the boxes with thicker borders are called *composite boxes* and are used to organize the design hierarchically. Each composite box represents a set of lower-level design components. In a complete design, each composite box has a corresponding design page showing its decomposition.

There is a special composite box for each design, called *context box*, which is at the highest design level and represents the interface with its environment. The context box appears on the *context page*.

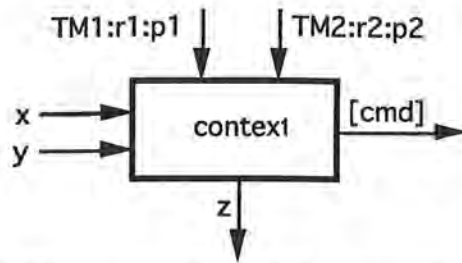


Figure 2.2(a): The context box of the example application.

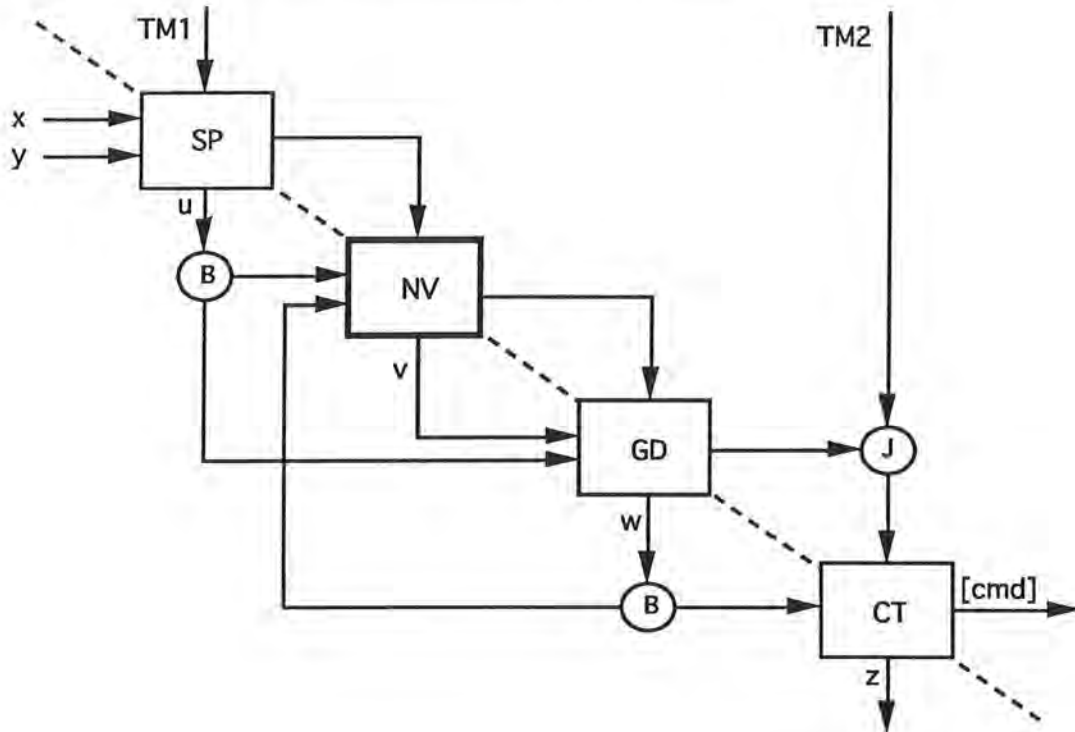


Figure 2.2(b): The decomposition page of the context box.

The design components appearing on the decomposition page of a composite box are called its *direct components* which may in turn be composite boxes themselves. The container-component relation constitutes a design hierarchy and is transitive. The direct components of a composite box is at a design level one lower than its own level. Note that some arrows and texts may appear at different design levels. For example, the control-in/out arrows in Figure 2.2(a) are exactly those arrows entering/exiting the control part of Figure 2.2(b) from the top/right and the data-in/out arrows in Figure 2.2(a) are exactly those arrows entering/exiting the data part of Figure 2.2(b) from the left/bottom. These arrows connect the design components on different design pages. In the following, components either refers to direct components or all

components by the transitive container-component relation, depending on the context.

Figure 2.2 shows the simplified design diagram for a missile control application, where SP stands for sensor processing; NV navigation; GD guidance and CT control. Figure 2.2(a) is the context box of the application which is driven by two timers TM1 and TM2 with the periods  $p_1$  and  $p_2$ , respectively, reads two external inputs  $x$  and  $y$  from two sensors, generates one external output  $z$ , and sends the control command to the actuator, where the external output  $z$  is used. The two timers start to count at different times  $r_1$  and  $r_2$ , where  $r_1$  depends on the length of system initialization and  $r_2$  is offset from  $r_1$  by some constant. Figure 2.2(b) is the decomposition page of the context box.

As illustrated by Figure 2.2(b), each design page consists of three parts: the functional part, the control part and the data part. The functional part is shown along the diagonal consisting of the boxes. The control part is shown above the diagonal and the data part below the diagonal. Note that the dashed line is not part of the design. It is only used to show the diagonal.

Each composite box constitutes a self-contained subdesign. The text in it describes the functionality of the subdesign whose interface with other parts of the design is defined by the attached arrows. See box NV in Figure 2.2(b). The arrows on the top/right of a composite box carry control stimuli into/out of the box. The arrows on the left/bottom carry data into/out of the box. However, these control stimuli and data are for its components instead of itself, in contrast to an atomic box.

It should be noted that a control arrow is a control-in arrow to its destination, but it is a control-out arrow to its source. A control stimulus flows from its source to its destination. All control stimuli are originated from the external control driving sources: timers or sporadic events.

In addition to atomic boxes, guards and control operators also act on control stimuli and appear in the control part, which will be introduced in section 2.2.1. However, to interpret the design in Figure 2, we need to explain the control join operator in Figure 2(b) which is drawn as a circle with a J. The stimuli flowing into a control join along the in-arrows flows out along the out-arrow to the same destination (box CT in Figure 2(b)).



Similar to a control arrow, a data arrow is a data-in arrow to its destination and a data-out arrow to its source. The variable associated with the arrow is updated by its source and is used by its destination.

In addition to data arrows and variables, data operators also appear in the data part, which will be introduced in section 2.2.2. However, to interpret the design in Figure 2.2, we need to explain the two data branch operators in Figure 2(b) each of which is drawn as a circle with a B. The variable associated with a data branch is updated by its source and is used by more than one destination.

We now interpret the design in Figure 2.2. The external inputs  $x$  and  $y$  are to be sampled at the regular rate of  $1/p_1$ . Each time  $x$  and  $y$  are sampled,  $u$  must be recomputed by function SP with the new values of  $x$  and  $y$ . After SP, the navigation related functions contained in box NV must be executed, whose decomposition will be shown in section 2.3. Then, function GD and CT must be executed in that order for guidance control. The internal state  $u$  is updated by SP and is shared by GD and some components of box NV. Similar interpretation applies to  $w$ .  $v$  is updated by some components of box NV and is used by GD. In addition to being driven by timer  $TM_1$ , function CT is also driven by timer  $TM_2$ . When CT finishes its execution, it sends a control command to the actuator, where the external output  $z$  is used.

Up to now, the reader should already obtain a clear picture about our design diagram. We now further illustrate the design hierarchy through Figure 2.3. In the next section, we shall introduce more design components. The context box 'ABC' in Figure 2.3 is decomposed into three boxes 'A', 'B' and 'C' which are at the design level one lower than the context box. Boxes 'A' and 'C' are themselves composite boxes and their decomposition pages are also shown in Figure 2.3. And so forth. Generally speaking, the "top" diagram is the more "abstract", while the "bottom" diagram is the more "concrete". Correspondingly, the top level hides the greatest amount of detail, while the lowest level exposes the greatest amount of detail.

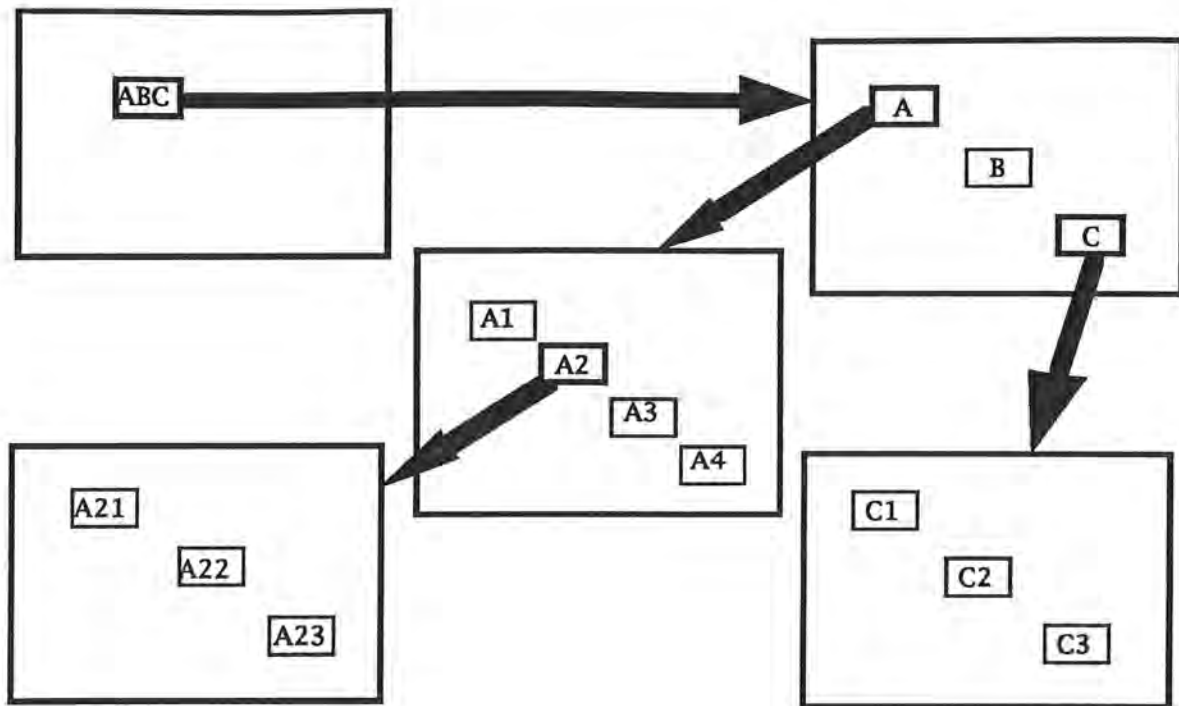


Figure 2.3 Hierarchical decomposition of a design

## 2.2 More Design Components

We now introduce more control/data operators and guards. Guards are introduced together with control operators because they all appear in the control part and are typically used together.

### 2.2.1 Control Operators and Guards

Control operators act on control stimuli. There are six kinds of control operators: control branch, control join, repeat, skip, if, and case operators. A control branch is drawn as a circle with a B. See Figure 2.4(a). The stimulus flowing in from the in-arrow is duplicated at the operator, one for each out-arrow, and each resulting stimulus flows out along its out-arrow to its own destination. A control join operator is drawn as a circle with a J. See Figure 2.4(b). The stimuli flowing into a control join along the in-arrows flows out along the out-arrow to the same destination. See Figure 2.2.

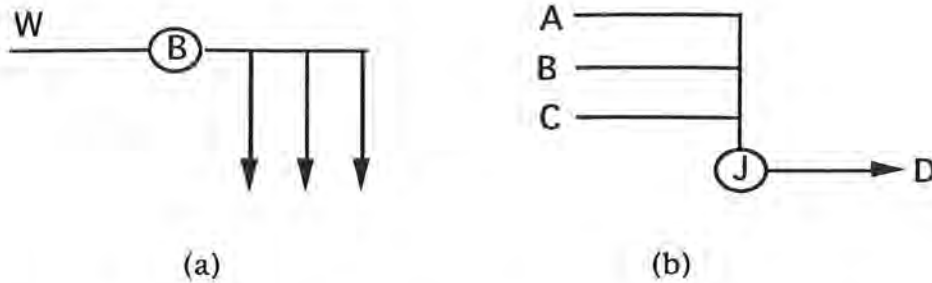


Figure 2.4 (a) A control branch operator; (b) A control join operator.

An if operator is drawn as a circle with an IF and a case operator is drawn as a circle with a C. See figure 2.5. A sequence of if/case operators connected one after another can be used to specify more than two choices. A sequence of connected case operators has a single variable associated with them, which takes a set of mutually exclusive values. On the other hand, the conditions associated with if operators are more flexible. The conditions are given in the brackets.

The expressions associated with if and case operators are *guards* which specify different system operation modes. The global variables in them are updated in those functions which determine mode changes. The control stimulus entering a if/case operator flows out along the out-arrow whose associated guard has true value. An example illustrating how case operators and guards are used to specify different system operation modes will be given in section 2.3.

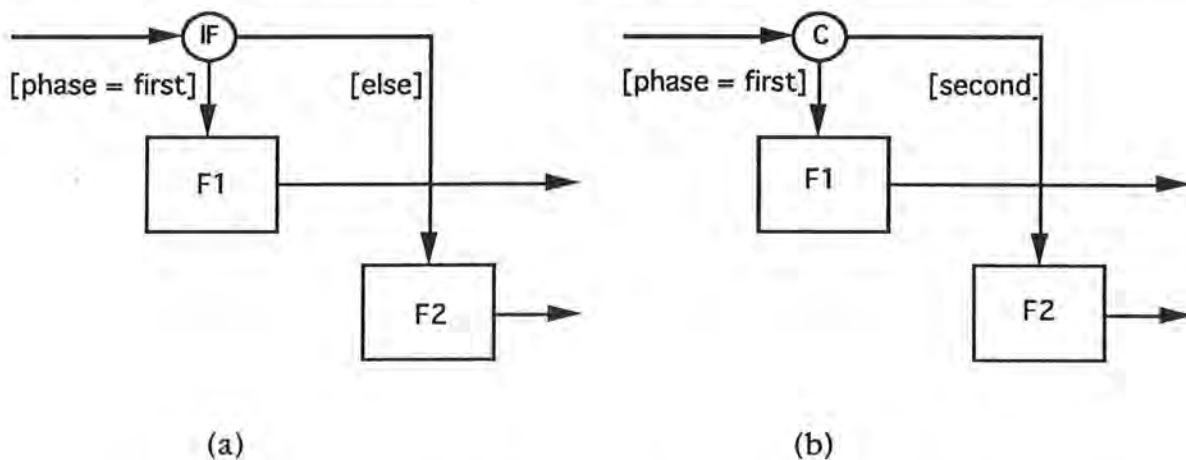


Figure 2.5 (a) An example of IF operator; (b) An example of CASE operator

A repeat operator is drawn as a circle with an R. See Figure 2.6(a). A control stimulus entering a repeat operator repeatedly drives the box linked to the bottom arrow for the number of times specified by the bound in the bracket. Only when the repetition finishes, a control stimulus flows out along the right arrow. If the box is a composite box, then its box-components must be on a single execution path (single-in-single-exit). For example, box F in Figure 2.7(a) is executed four times for each control stimulus reaching the operator and only after that, a control stimulus flows out along the right arrow.

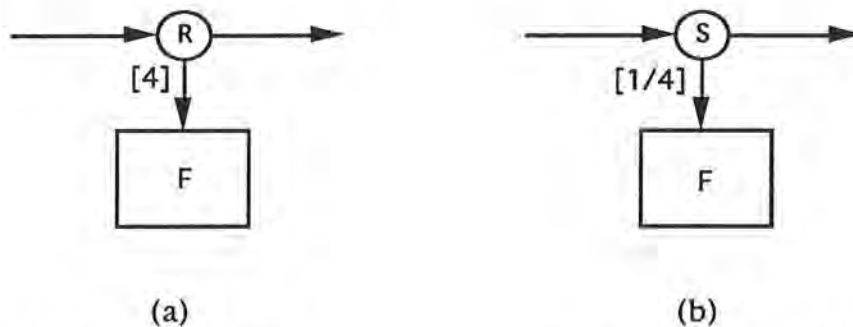


Figure 2.6 (a) An example of repeat operator; (b) An example of skip operator

A skip operator is similar to a repeat operator and is drawn as a circle with an S. However, contrary to a repeat operator, it decreases the frequency by which its destination is executed. For example, box F in Figure 2.7(b) is executed only once for every 4 control stimuli reaching the operator. Those skipped (not-driving-box) control stimuli directly flow out along the right arrow. We need the skip operator because in some applications, some functions do not need to be executed as frequently as their predecessors.

### 2.2.2 Data Operators

There are two kinds of data operators: data branch and data join. A data branch is drawn as a circle with a B and a data join is drawn with a circle with a J. See Figure 2.4. Although a data branch/join and a control branch/join have the same graphical representation, data operators appear in the data part of a design page but control operators appear in the control part. See Figure 2.2. Data operators represent data sharing but control operators act on control stimuli. The variable associated with a data join operator is updated by more

than one source and used by one destination. On the other side, the variable associated with a data branch operator is updated by one source and used by more than one destination.

### 2.3 Flattening a Design

Although the design hierarchy helps to organize a design and focus attention on just enough details at a time, a broader view than a design page allows us to directly see how the components on different design pages are connected and thus helps us examine the design.

Flattening a composite box enables us to obtain a broader view of a design. It is a recursive process: (1) The composite box is replaced by its decomposition page; (2) The resulting composite boxes are recursively replaced by their decomposition pages until the result contains no composite boxes. It should be noted that flattening the context box gives the global view of a design.

Figure 2.7 illustrates design flattening. Figure 2.7(a) completes the design in Figure 2.2, where AT stands for acceleration transformation; VPU velocity and position update; and GM gravity modeling. Figure 2.7(b) shows the result of flattening the context box.

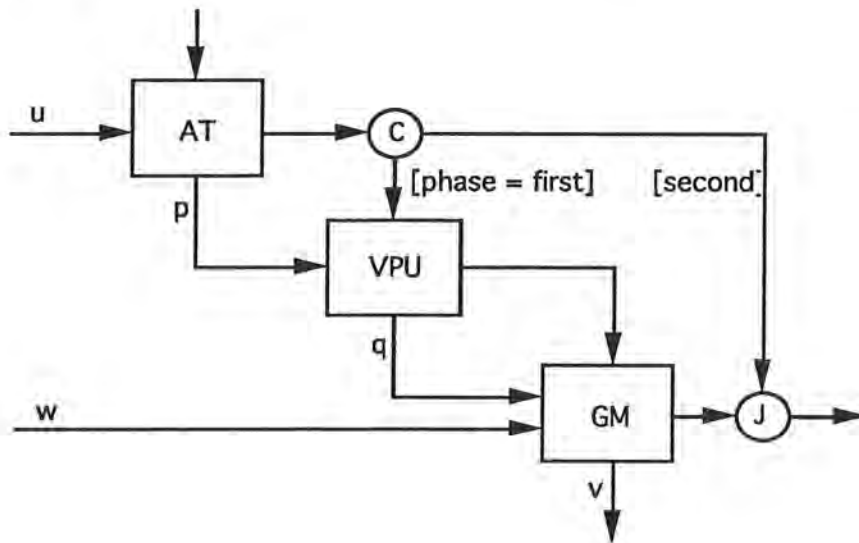


Figure 2.7(a) The decomposition of box NV in Figure 2.2(b)

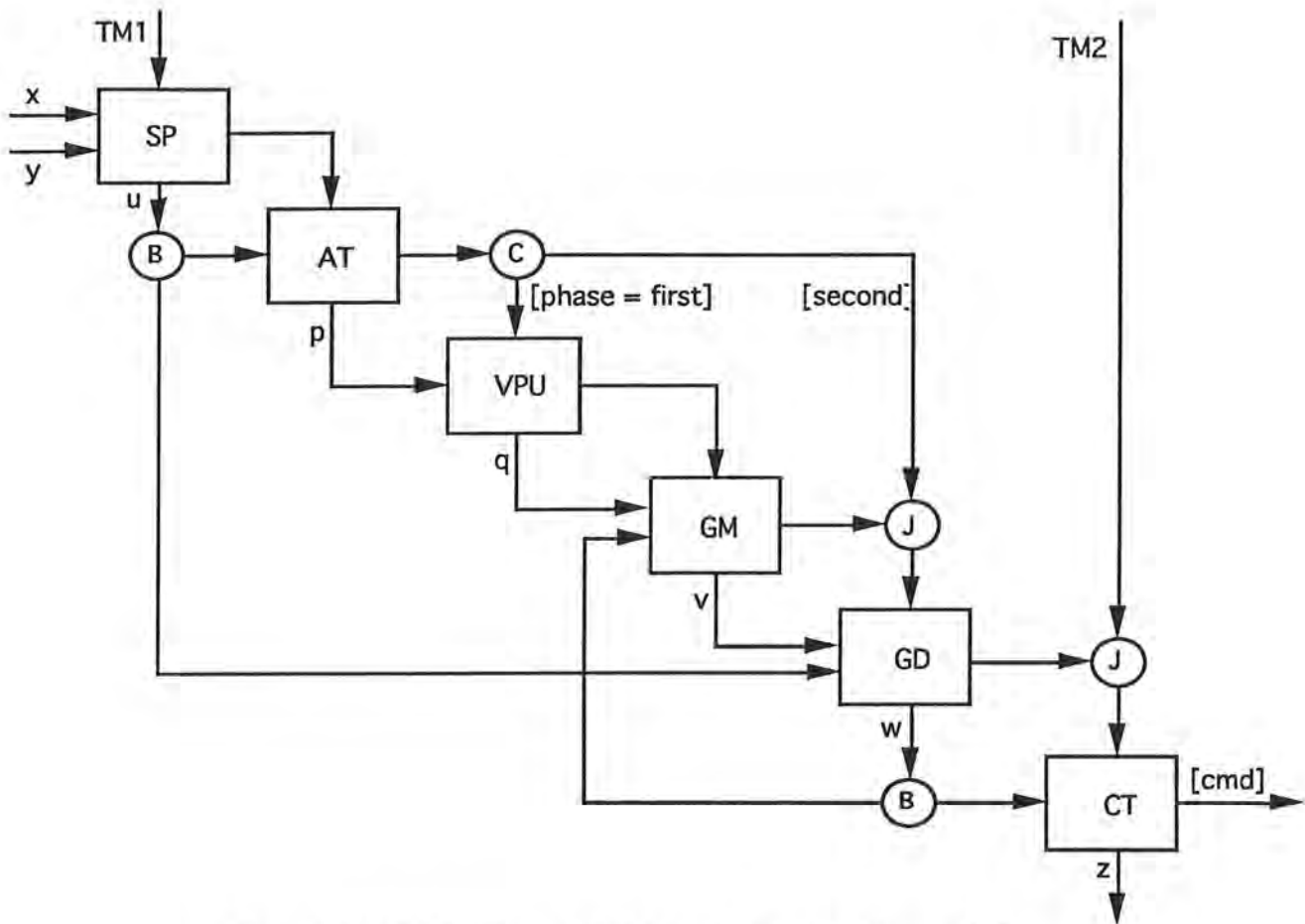


Figure 2.7(b) Flattening the design shown in Figure 2.2

In Figure 2.7, the case operator specifies two different operation modes. Function AT uses  $u$  (the output from the sensor processing) to decide the operation mode. "phase" is a well restricted global variable updated by AT and used by the case operator. In mode 1, function VPU and GM are executed after AT in that order. In mode 2, Function GD is executed immediately after AT.

It should be pointed out that in Figure 2.7(b), the internal state  $w$  is used more frequently than it is updated. In the real application, this is for reducing the number of processors. Furthermore, in the real application, the boxes SP, GD, and CT are all composite boxes. Due to the page limit, we simply present each as an atomic unit here. We keep composite box NV to illustrate design flattening and mode changes.

### 3 The Implementation of HaRTS

HaRTS has been implemented using Objex. In this section, we first show the HaRTS user interface and illustrate how a hierarchical design is created. Then, we describe how HaRTS has been derived from Objex through specialization.

#### 3.1 The HaRTS User Interface

Figure 3.1 shows the user interface when HaRTS is launched. The design process by HaRTS is a top down design process. As a result, when you launch HaRTS, the first thing you see is always the context page which contains the context box. You always start your work from the context page.

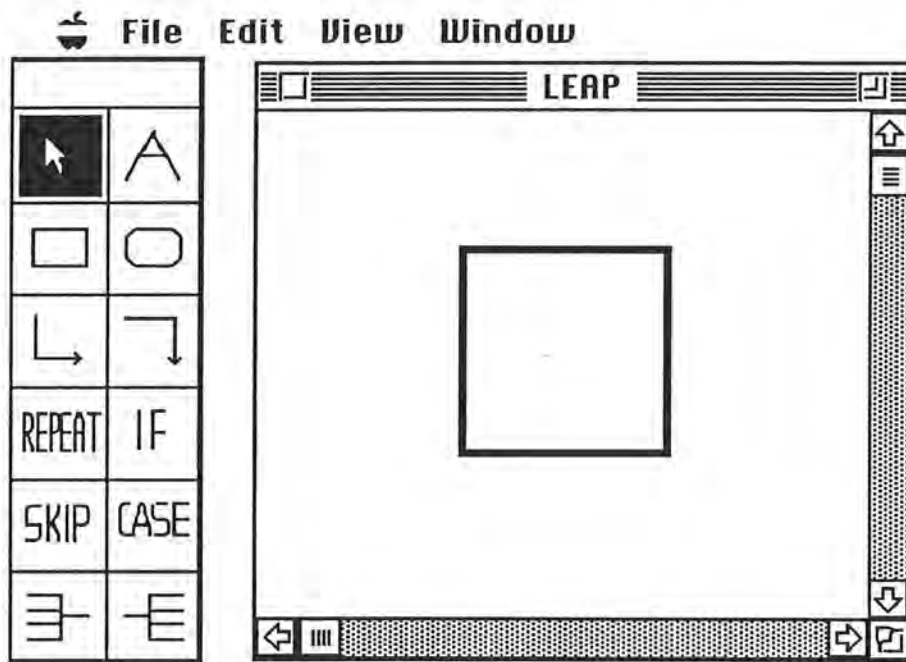







Figure 3.1 The context window when HaRTS is launched.

Before we move on, we want to point out the difference between the terms page and window. The term page is more design-oriented. On the other hand, the term window is more application-oriented. A window can be opened and closed. However, a design page is always there after it is created although at a specific time, it may not be displayed on a window. In the following, we shall

not distinguish the two terms and use them freely. Indeed, when a design page is displayed on a window, the two terms refer to the same thing.

In Figure 3.1 the left rectangle enclosing a collection of iconic symbols is a palette. Like palettes in all GUI applications, this palette is used as the mean for users to specify different operation modes: adding, selecting design components. In the palette there are twelve iconic symbols used to select one of the following different operation modes:

	selection tool
<b>A</b>	adding text
	adding box
	not using now
	adding data arrow
	adding control arrow
<b>REPEAT</b>	adding repeat control operator
<b>IF</b>	adding if control operator
<b>SKIP</b>	adding skip control operator
<b>CASE</b>	adding case control operator
<b>BRANCH</b>	adding data and control branch operator
<b>JOIN</b>	adding data and control join operator

We shall show examples illustrating how to use the palette in the following subsections.

The top of Figure 3.1 is a menubar which follows the standard Macintosh user interface. It includes five menus, four of which are listed in Figure 3.2. We omit the apple menu because it is a standard part of the Macintosh use interface.



File	Edit	View	Window
New %N	Undo %Z	DataAccess	Untitled1
Open %O	Redo %R	Class	
Close %W	Cut %H	Text	
Close All %L	Select All %A	Flatten %F	
Save %S	Clean up Window %U		
Save As...			
Page Setup...			
Print %P			
Quit %Q			

Figure 3.2 Menu items.

We now briefly describe the functionality of each menu item.

#### File menu

- New** Open a new window with a context box in it as shown in Figure 3.1 to start a new design.
- Open** Read in a HaRTS design file and open a new window showing the context page of the design. Note that HaRTS can support several designs at the same time.
- Close** Close the front window.
- Close All** Close all the window(s) of the current design.
- Save** Save the current design. If the design has not been saved, do the same thing as **Save As** below.
- Save As** Open a dialog asking users to input a file name and then save the current design to a new file with that file name. Note that the title of the context window of a design is kept to be the same as the file name of the design.
- Page Setup** Set printing parameters.
- Print** Print the design diagram in the front window.
- Quit** Quit.

### **Edit menu**

**Redo** Redo a command such as cutting, dragging, or growing an graphical object.

**Undo** Undo a command.

**Cut** Delete the selected object.

**Select All** Select all objects in the front window.

**Clean Up Window** Recalculate the coordinates of the graphical objects in a window and redraw them so that all the boxes are drawn along the diagonal of the window. Note that when a box is created, the designer can put it wherever he/she wants, not necessarily along the diagonal.

### **View menu**

**DataAccess** Start a data access scene(not implemented)

**Class** Start a class scene(not implemented)

**Text** Start a text scene(not implemented)

**Flatten** Flatten the selected composite box and show the result in a new window.

**Window menu** List all the title(s) of the open windows of the current design.

In addition to the above functionalities, the design tool further supports dragging and growing a graphical object. When a box/operator is dragged, the arrows connected to it are moved along the box/operator. That is, as a box/operator is moved, the coordinates and the sizes of the arrows connected to it are automatically adjusted. It should be pointed out that dragging an arrow has different meanings in different situations, which will be further discussed in the following subsections.

To grow an arrow, you need to point the cursor to the start/end point of the arrow and then press the mouse and move the cursor to the other place. As the result, if the mouse up location is inside other box (or operator), the source /destination of the arrow has been updated. In other hand, the size of an operator is fixed. That is, operators can not be enlarged or shrunk.

### 3.1.1 Creating New Designs

We now illustrate how to add new design components to a design and how to change a design. For example, to add a control-in arrow to the context box in Figure 3.1, the user clicks on the control-arrow icon in the palette, drags the mouse from one point above the context box to the context box. Figure 3.3 shows the context window after a control-in arrow is added. Similarly, a control-out arrow can be added by dragging the mouse from the context box to a point on the right of it. But to add a data arrow, you must first click on the data arrow icon. The data-in(out) arrow is added by dragging the mouse from a point on the left of the context box(the context box) to the context box(a point below the context box).

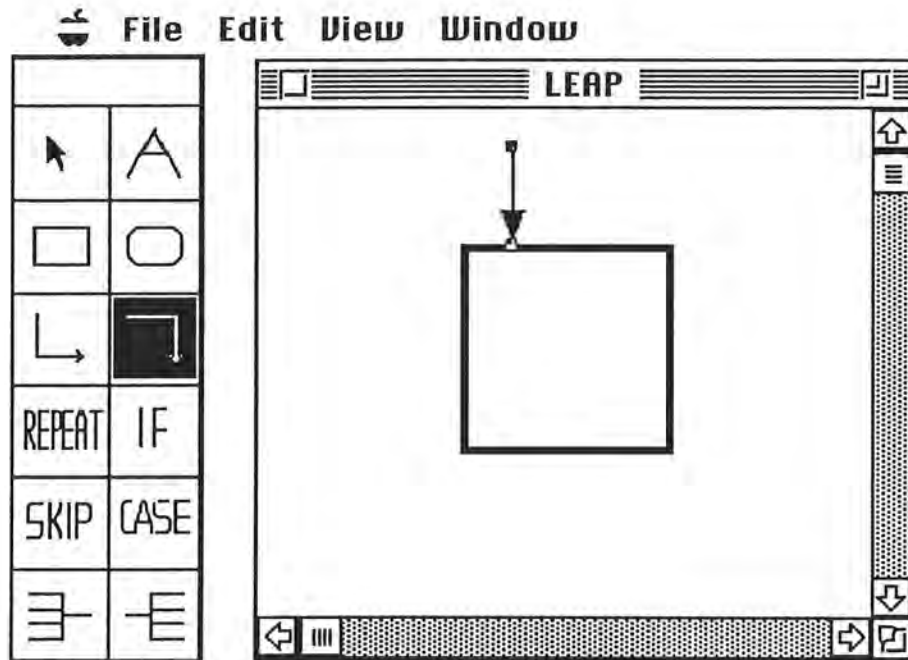


Figure 3.3 The context window after a control-in arrow is added.

To attach text to a box/arrow, first click on the **A** icon in the palette and then click on the box/arrow. After that, a dialog appears to let you type texts. Figure 3.4 shows the context window after the text TM1 is attached to the control-in arrow in Figure 3.3.

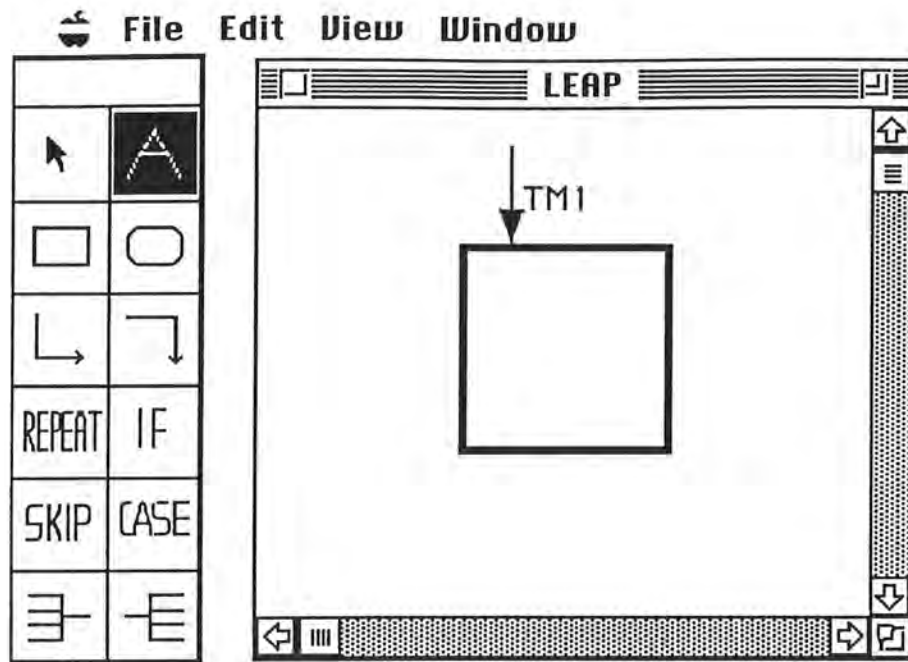


Figure 3.4 The context window after TM1 is attached to the control-in arrow.

Figure 3.5 shows the context window after more design components are added.

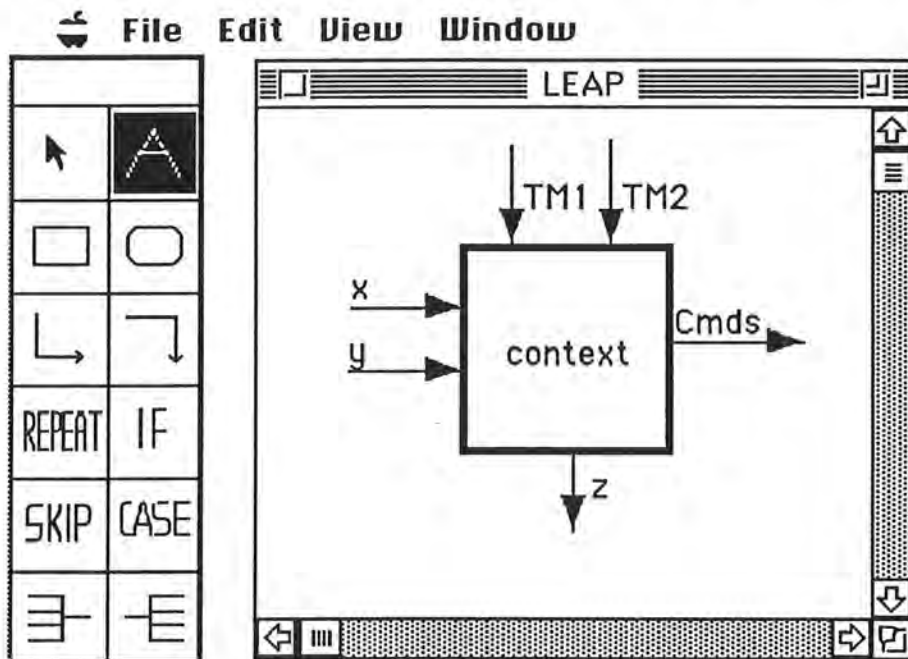


Figure 3.5 The context window after more components are added.

### 3.1.2 Decomposing a Composite Box

Double clicking on a composite box opens a new window which shows the decomposition of the composite box. Figure 3.6 is the decomposition window of the context box in Figure 3.5. Note that the window title is the same as the text shown inside the composite box. The arrows in Figure 3.6 are exactly those arrows connected to the context box in Figure 3.5. However, the arrows in Figure 3.6 are shown in wider pen mode signifying that they are inherited from the context box and have not been attached to any box/operator on this decomposition page, yet. Such inherited arrows are called *external arrows* with respect to the design page. These arrows and the associated texts are examples of a single data model with multiple views.

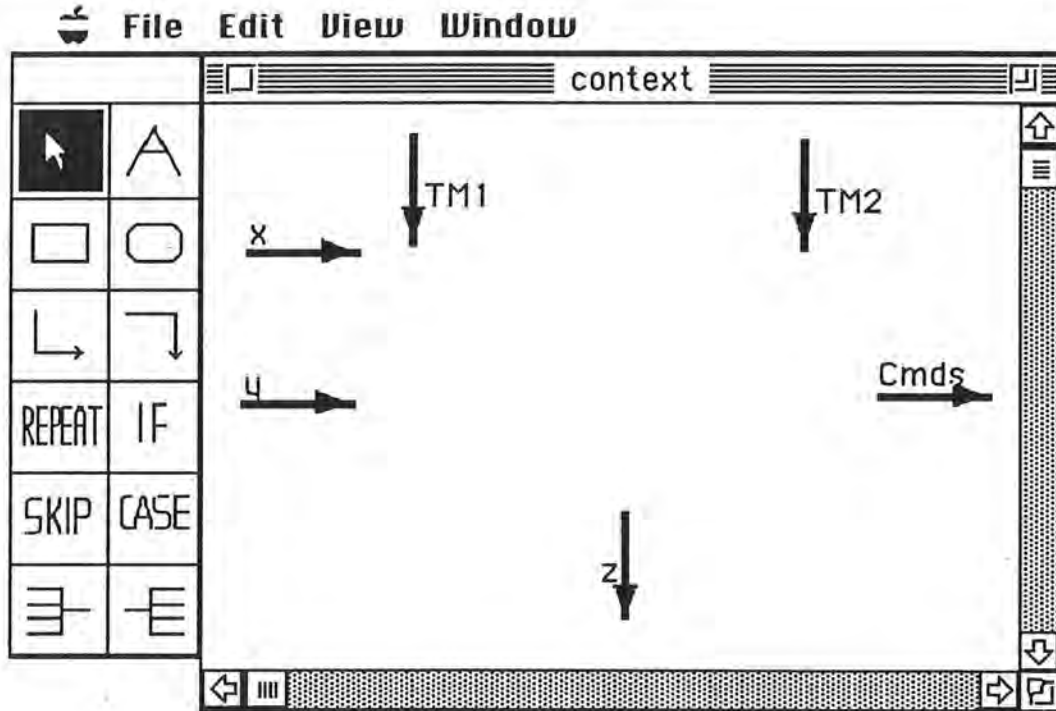


Figure 3.6 The decomposition window of the context box

To add a box to the decomposition window in Figure 3.6, first click on the box icon in the palette and then drag the mouse inside the window. The initial size of the box is determined by the start point and end point of the dragging. Figure 3.7 shows the result after two boxes are added to Figure 3.6. Note that the default type for a newly created box is atomic. To change an atomic box to a

composite, first click on the selection icon in the palette and then double click on the box. After that, a dialog appears asking you to confirm the change.

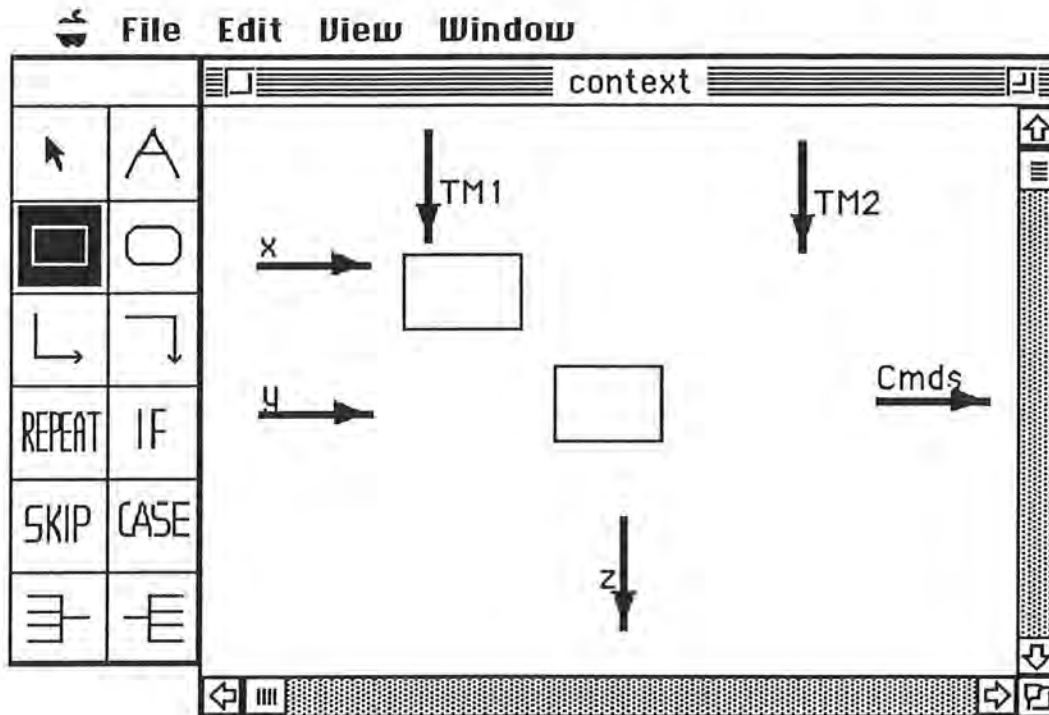


Figure 3.7 The decomposition window after two boxes are added.

To attach an arrow in Figure 3.7 to a box there, you simply need to drag the arrow towards the box until the arrow touches the box. To create a control/data arrow which connects the two boxes in Figure 3.7, first click on the control/data arrow icon in the palette and then drag the mouse from the source to the destination. Figure 3.8 shows the result after some of the arrows in Figure 3.7 are attached to the boxes and a new control arrow is added to connect the two boxes. Those arrows whose source and destination are on the same page are called *internal arrows* with respect to the page. Note that after an external arrow is attached to a box, it is no longer drawn in wider pen mode.

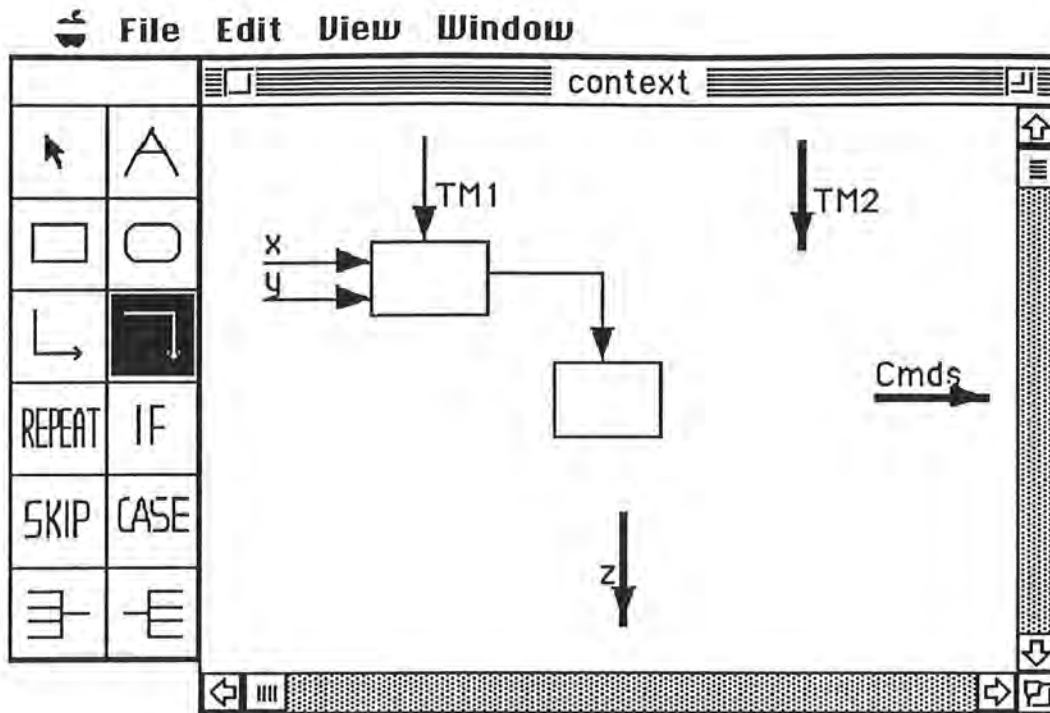


Figure 3.8 The decomposition window after the arrow change.

The boxes in Figure 3.8 can be dragged. When the boxes are dragged, the arrows connected to them are also moved along the boxes and the size of the internal arrow is automatically adjusted at the same time. You can also delete the boxes and the internal arrow. However, you can only move but not delete the external arrows and the associated texts because they are inherited from a higher level design page. In general, a graphical object can only be deleted at the design page where it is created.

Figure 3.9 shows the decomposition window after more design components are added and the external arrows are all attached. Adding (attaching an arrow to) an operator is similar to adding (attaching an arrow to) a box. The only difference is that you need to click on the corresponding operator type. If a branch/join operator is added to a location above the diagonal of the window, it is a control branch/join; otherwise, it is a data branch/join. Note that the palette is omitted in Figure 3.9

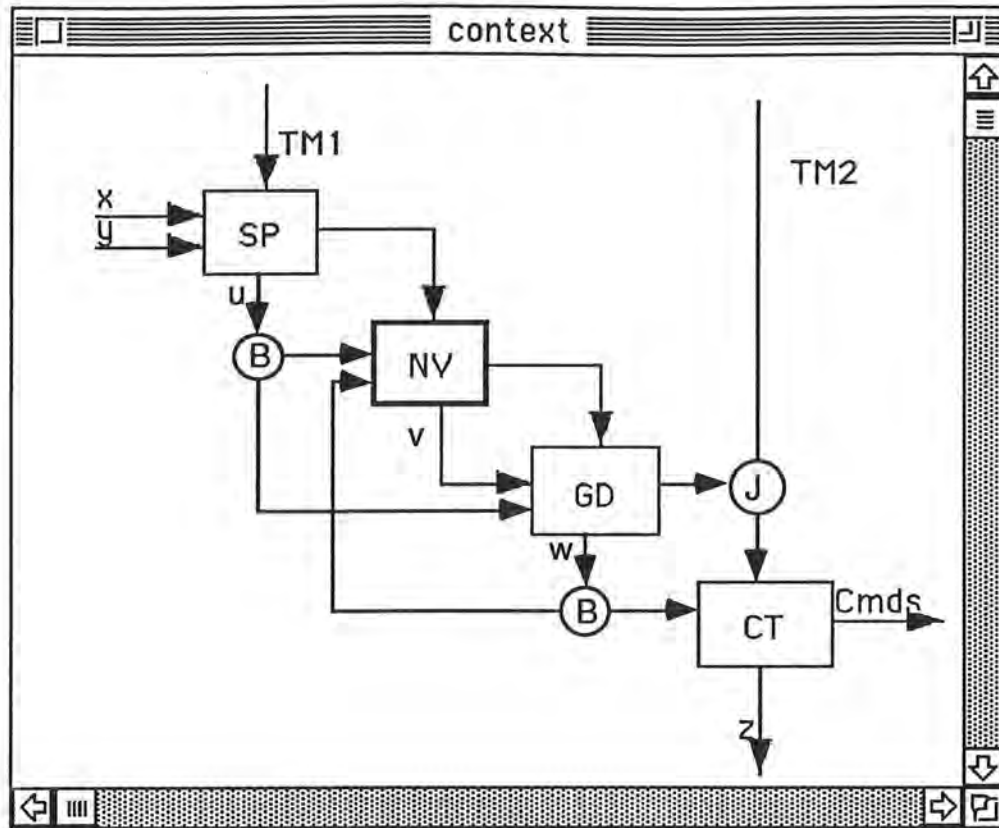


Figure 3.9 The decomposition window after more components are added.

Finally, it should be pointed that the design tool actually supports colored design diagram. Boxes are drawn in blue color; The circles of operators in cyan and the texts in them in black; control arrows in red; and data arrows in green.

### 3.1.3 Flattening a Composite Box

To flatten a composite box, first select a composite box, and then issue the Flatten command from the View menu. After that, a new window is opened showing the result of the flattening. Figure 3.10 shows the result of flattening the composite box.



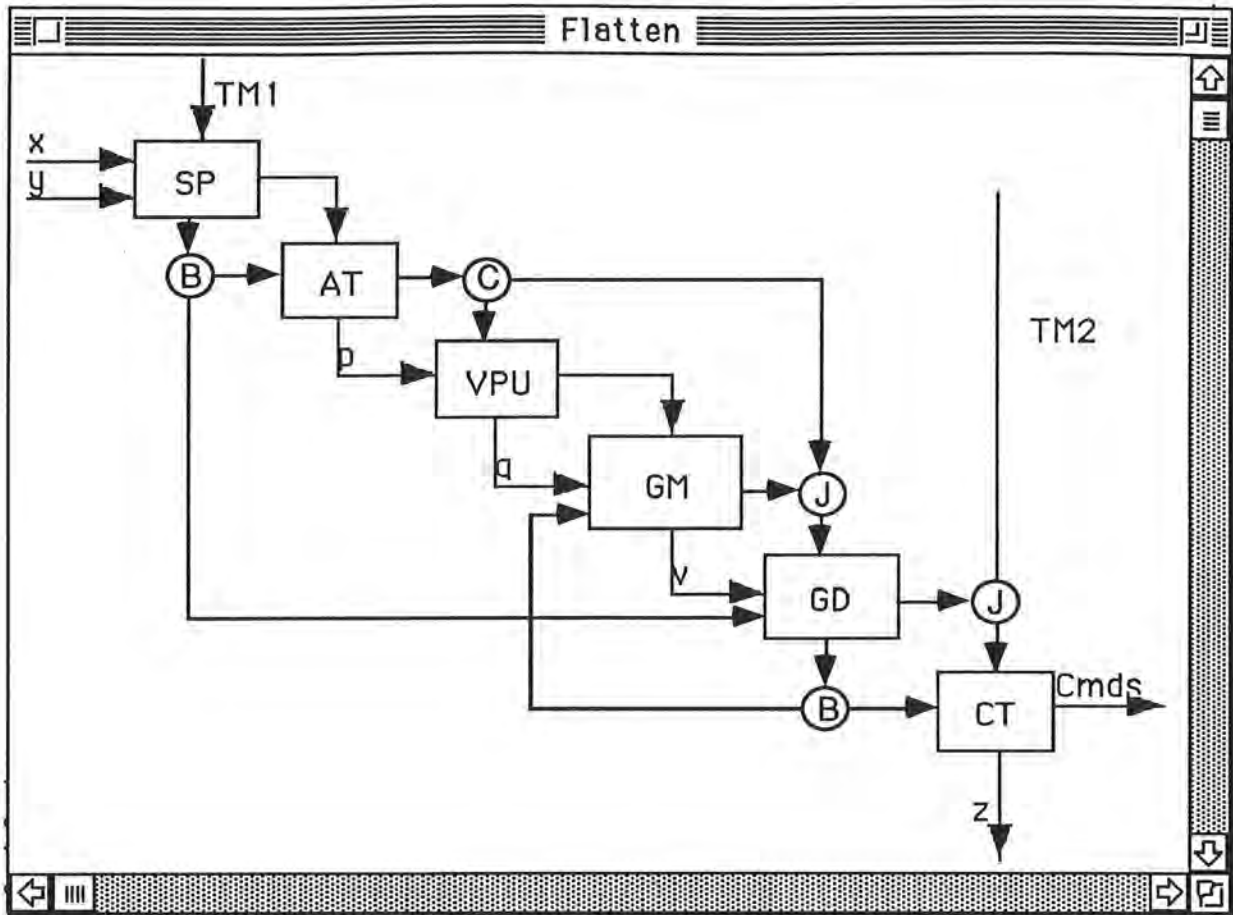


Figure 3.10 The result of flattening the context box.

### 3.2 The HaRTS Class Hierarchy

We now describe how HaRTS is derived from Objex through specialization. Basically, we have extended the Objex class hierarchy to handle our domain specific information. The result of this is the HaRTS class hierarchy.

Implementing HaRTS is a process of specializing Objex through subclassing, overriding, and extending. Some existing classes of the framework can be directly utilized through instantiation. For example, the Application class in Figure 1.2. Other classes need to be subclassed because more data fields or/and new methods are needed, or/and existing methods need to be overridden. For example, the Rectangle class in Figure 1.3 is subclassed. The Rectangle class of the framework offers only a simple rectangle shape but nothing else.

However, the rectangle for the box in our design diagram is much more complex. New data fields and new methods are needed and existing methods need to be overridden. For example, when a box is dragged, the connected arrows need to be moved along with the box.

Furthermore, some brand new classes need to be created almost from scratch. In general, such classes are needed to define the domain-specific data model. For example, a box in our design tool is not simply a rectangle. It has its own semantics. To keep track of its semantic information, (e.g., the direct components of a composite box,) a new class is created directly under the root class Object in Figure 1.2.

It should be pointed out that each graphical object on the screen has two corresponding objects in the memory: one containing the semantic information (e.g., box and arrow interconnection,) and the other containing the graphical appearance information (e.g., shape type and coordinates on the screen). The purpose of doing so is to separate view from data model. For example, each box on the screen has two corresponding objects in the memory, one being the instance of the class which we created from scratch for storing the semantic information and the other being the instance of a subclass of the Rectangle class for keeping the graphical appearance information. In the following we first describe the HaRTS data model class hierarchy and then the associated shape class hierarchy.

### **3.2.1 HaRTS Data Model Class Hierarchy**

Figure 3.11 shows the class hierarchy defining the HaRTS data model. The classes Box, Arrow and Operator are the direct subclasses of the class Object in Figure 1.2. Those three classes and their subclasses are all domain-specific classes.

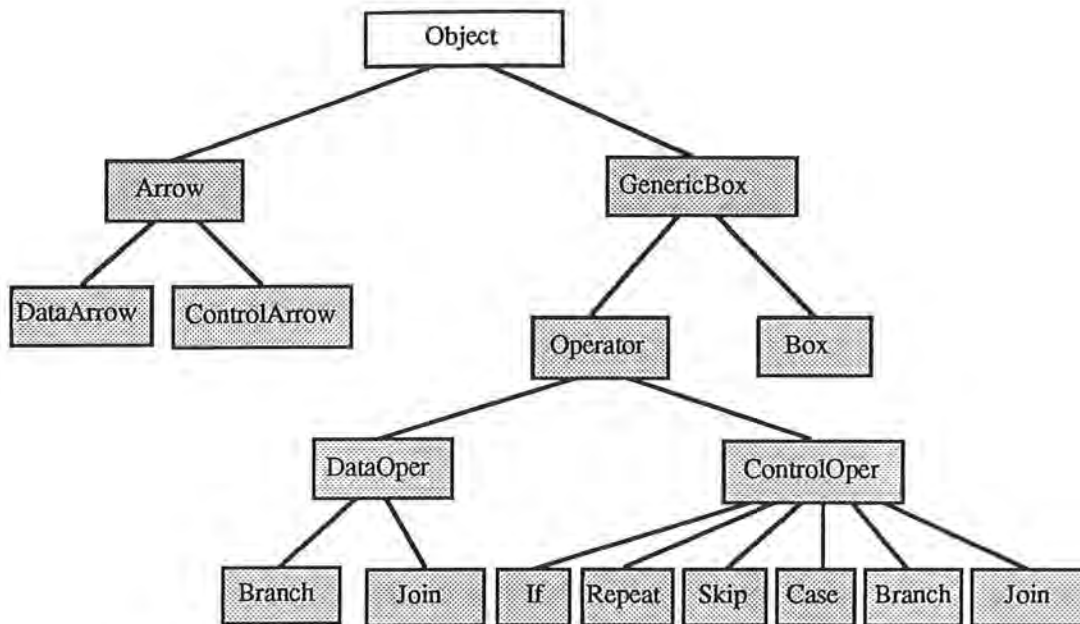


Figure 3.11 HaRTS data model class hierarchy

We now briefly describe the main components (instance variables and member functions) of the domain specific classes.

### The Box Class

#### Instance Variables:

- |                  |  |
|------------------|--|
| - fIdentity      | the unique id of the box   |
| - fName          | the text shown inside the box                                      |
| - fBoxType       | the box is either atomic or composite                              |
| - fShape         | the shape of the box   |
| - fContainer     | the direct container of the box                                    |
| - fTheWindow     | the decomposition window of the box if it is composite             |
| - fDataInList    | a list for the data-in arrows connected to the box                 |
| - fDataOutList   | a list for the data-out arrows connected to the box                |
| - fCntrInList    | a list for the control-in arrows connected to the box              |
| - fCntrOutList   | a list for the control-out arrows connected to the box             |
| - fComponentList | a list for the direct box components if the box is a composite box |

- fOperatorList a list for the direct operator components if the box is a composite box

#### Member Functions:

- DoubleClick() change the box type to be composite if it is an atomic box or decompose the box if it is already composite
- FlattenABox() flatten the box, only useful for a composite box
- AddDataInArrow() add a data-in arrow to the data-in list
- AddDataOutArrow() add a data-out arrow to the data-out list
- AddCntrInArrow() add a control-in arrow to the control-in list
- AddCntrOutArrow() add a control-out arrow to the control-out list
- GetIdentity() get the box id
- SetIdentity() set the box id
- DecomposeABox() display the decomposition of the box, only useful for a composite box
- Adjust() inform the arrows attached to the box to adjust their coordinates and sizes as the box moves
- Duplicate() duplicate the box, used when a composite box is flattened
- Cut() delete the box
- UndoCut() undo the delete action
- DoWrite() save the box and the associated arrows to a file
- DoRead() read the box and the associated arrows from a file
- Is\_a() return the object type (e.g., box, arrow, or operator)

#### The Arrow Class

##### Instance Variables:

- fIdentity the unique identity of the arrow
- fText the text attached to the arrow
- fConnecton a dictionary storing the connection information of the arrow

Note that an arrow may appear on more than one design page. Each item of the dictionary contains a

source, the source id, a destination, the destination id and a shape of the arrow.

For example, Figure 3.12 shows an arrow with three different views. The window at the back shows the context page. The window at the middle show the decomposition of the contest box. And the window at the front shows the decomposition of the composite box 'A'. Although the three arrow shapes on the different windows attach to the different boxes, they belong to the same arrow. This is an example of a data model with different views. Each item in the dictionary for an arrow stores the information for one view of the arrow. Other examples of arrows with multiple views can be found in the above figures, from Figure 3.3 to Figure 3.10.

It should be pointed out that the reason for us to keep both the source/destination and its id in the dictionary is that when read a design from a file, we obtain the ids first and we need to use the ids to recover the design hierarchy.

#### Member Functions:

- AddAConnection() add an item to the dictionary when a new view of the arrow is generated, due to the decomposition of a composite box
- SetSrcByShape() set the source of a dictionary item given the shape

Note that an arrow may have one source and several destinations or vice visa. For example, in Figure 3.12, the source of the arrow is the external environment. The destinations are the context box, the box 'A', and the box 'B', depending on the view of the arrow.

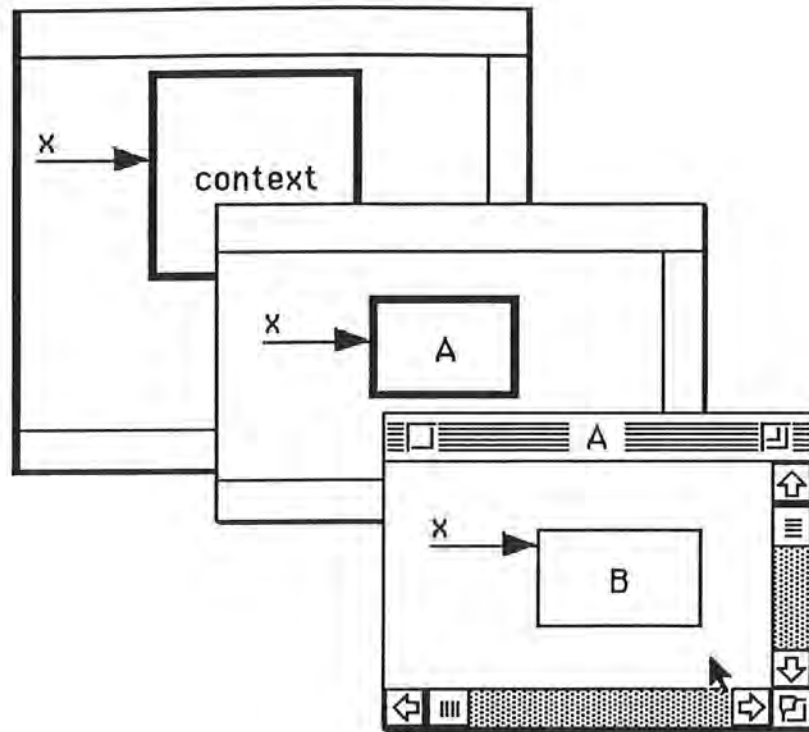


Figure 3.12 An arrow with three different views

- GetSrcByShape() get the source of a dictionary item given the shape
- SetDestByShape() set the destination of a dictionary item given the shape
- GetDestByShape() get the destination of a dictionary item given the shape
- SetSrcIdByShape() set the source id of a dictionary item given the shape
- GetSrcIdByShape() get the source id of a dictionary item given the shape
- SetDestIdByShape() set the destination id of a dictionary item given the shape
- GetDestIdByShape() get the destination id of a dictionary item given the shape
- GetShapeBySrc() get the shape of a dictionary item given the source
- GetShapeByDest() get the shape of a dictionary item given the destination

- SetText()                    attach text to the arrow
- GetText()                    get text attached to the arrow
- Duplicate()                 duplicate the arrow, used when a composite box is flattened
- Cut()                         delete the arrow
- UndoCut()                  undo the delete action
- DoWrite()                  save the arrow to a file
- DoRead()                  read the arrow from a file
- Is\_a()                        return the object type

The DataArrow and ControlArrow classes are the direct subclasses of the Arrow class. Some of the member functions of the Arrow class have been overridden in its subclasses, e.g., Cut(), UndoCut(), Duplicate() and Is\_a(), etc..

### The Operator Class

#### Instance Variables:

- fIdentity                    the unique id of the operator
- fDataInList                 a list for the data-in arrow(s) connected to the operator
- fDataOutList                a list for the data-out arrows(s) connected to the operator
- fCntrInList                 a list for the control-in arrow(s) connected to the operator
- fCntrOutList                a list for the control-out arrows(s) connected to the operator
- fContainer                 the direct container of the operator
- fShape                      the shape of the operator

#### Member Functions:

- Adjust()                    inform the arrows attached to the operator to adjust the coordinates and sizes as the operator is moved
- Cut()                        delete the operator
- UndoCut()                  undo the delete action
- Duplicate()                 duplicate the operator used when flatten a composite box

- DoWrite()                    save the content of the operator to a file
- DoRead()                    read the content of the operator from a file
- Is\_a()                        return object type

All the data operators and control operators are the subclasses of the class Operator. Note that the fields fDataInList and fDataOutList of the Operator class are used only by the DataOper subclass and fCntrInList and fCntrOutList are used only by the CntrOper subclass. The reason to keep the four fields in the Operator class is that this facilitates our implementation of flattening a composite box. It should be pointed out that this way, an operator can be treated just like a box.

### 3.2.2 HaRTS Shape Class Hierarchy

Corresponding to HaRTS data model class hierarchy, there is a HaRTS shape class hierarchy, which is shown in Figure 3.13. All the classes in this hierarchy are the subclasses of the Shape class in Figure 1.4. In addition to inheriting from the superclass, our classes contain new data fields, new methods and/or overridden methods.

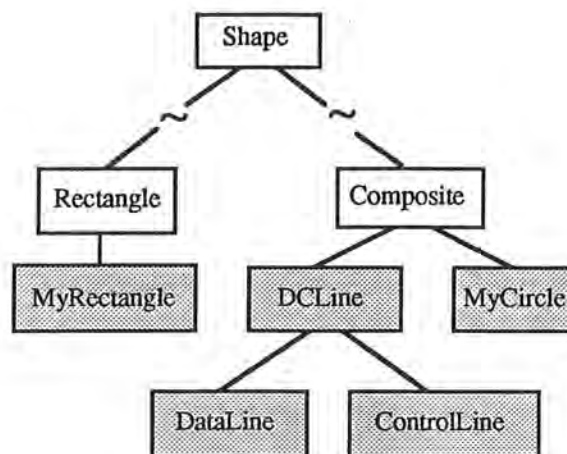


Figure 3.13 HaRTS shape class hierarchy

For each class in the model class hierarchy, there is a correspondent in the shape class hierarchy. This correspondent relationship is shown in Figure 3.14. Note that all the operator classes correspond to the MyCircle class.



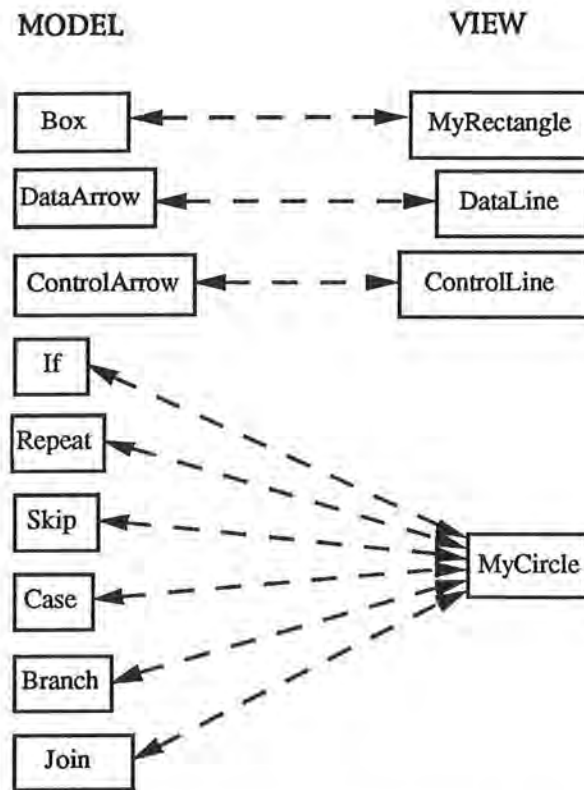


Figure 3.14 The correspondence between the model and shape classes.

### MyRectangle Class

The MyRectangle class provides the graphical shape for the class Box.

#### Instance Variables:

- fTheObj                      the box represented by the shape

#### Member Functions:

- Drag()                      move the box shape on the window and inform the box
- Grow()                      grow the box shape on the window and inform the box
- GetLabelPosition()      calculate the coordinates of the text attached to the box shape
- Is\_a()                      return the shape type

## DCLine Class

The DCLine class is the subclass of the Composite class. 'DC' means data and control. The purpose of using the Composite class is to make a new shape from several existing shapes. In our implementation, the shapes of the data and control arrows are the combinations of the simple shapes. For example the shape of the data arrow shown in Figure 3.15 is the combination of a line from a to b and an arrow line from b to c.



Figure 3.15 A example data arrow shape

### Instance Variables:

- fTheObj                    the arrow represented by the shape
- fLabel                    the text shape attached to the arrow shape
- fExternal                the type of the arrow shape: internal or external
- fInOut                    the direction of the arrow shape: entering the page from the top or left or exiting from the right of bottom, useful only for an external arrow
- fStartPt                 the start point of the composite shape
- fEndPt                    the end point of the composite shape

### Member Functions:

- Drag()                    drag the shape on the window
- Grow()                    grow the shape on the window
- GetLabelPosition()    calculate the coordinates of the text attached to the arrow
- DrawingRubberShape() draw the rubber shape
- Resize()                 Adjust the size of the shape according to the new coordinates
- Is\_a()                    return the shape type

Both the DataLine class and the Controlline class are the subclasses of the class DCLine. The methods GetLabelPosition(), Grow(), Resize(), and Is\_a(), etc., have been overridden in the subclasses.

### MyCircle Class

The MyCircle class which supports the shapes of all the operators is the subclass of the Composite class. The shape for each operator is the combination of a circle shape and a label shape. For example, the shape of the IF operator shown in Figure 3.16 is the combination of a circle shape and a IF label shape.



Figure 3.16 A example IF operator

#### Instance Variables:

- fTheObj                      the operator represented by the shape

#### Member Functions:

- Drag()                      move an operator shape on the window and inform the operator

- Is\_a()                      return the shape type

### 3.2.3 The Scene Class

In this section, we briefly describe the scene class of the framework which we found quite useful in event handling and coordinating user interface instances.

A scene object connects a menubar, a palette, and a serial of windows [9]. These standard user interfaces are tied together through a scene. Figure 3.17 shows an example of scene, which contains a menubar, a palette, and two windows.

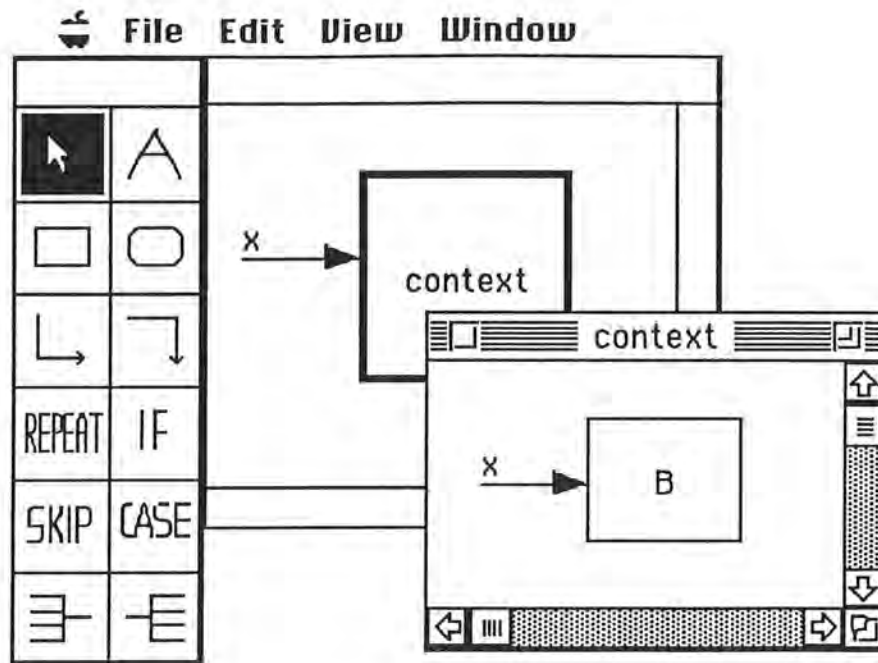


Figure 3.17 A example of scene

For each design, there is a corresponding scene object which connects its menubar, its palette, and its windows. Note that the number of windows of a design is in general dynamically changing during the design and as a window is opened or closed, it is added to or deleted from the scene. The scene object is responsible for coordinating the interactions among the user interfaces (i.e., the menubar, the palette, and the windows,). This saves programmers much effort in event handling and organizing user interfaces.

We now illustrate the usefulness of the scene concept through the example in Figure 3.17, where the selection icon in the palette is currently being selected. Now, if the user clicks on the window at the back, it becomes active. With the new front window, users may select a different icon in the palette, say, the 'rectangle' icon. Then, if the user switches back to the original front window, the status of the palette is automatically recovered, that is, the selection icon becomes selected again. It is the scene object that is responsible for remembering the different palette statuses associated with the different windows.

The scene in Figure 3.17 is for one design. Now, assume that the user opens another scene (another design) by selecting the Open item in the File menu. See section 3.1. As a result of this opening, the original front scene becomes inactive and this is reflected by hiding the menubar and palette of the scene and making its windows inactive. The new scene becomes the active scene. Its menubar and palette are visible and its context window becomes the front window. Now, if the user switches back to the first scene by clicking on one of its windows, its menubar and palette become visible automatically and their statuses are restored. Again, it is the scene object that is responsible for coordinating the change from one scene to another.

It must be pointed out that the Scene class should not be directly instantiated. You must subclass it to use it because the following three methods of the Scene class: CreateWindow(), CreateMenus() and CreatePalette() are virtual classes and must be overridden. The reason that these three methods are virtual is that different applications have different menus and different palettes.

The MyScene class, shown in Figure 3.18, is the subclass of the Scene class. The methods CreateWindow(), CreateMenus() and CreatePalette() have all been overridden in the MyScene class. The CreateWindow() is responsible for creating a new window and adding the window to the scene. The CreateMenus() is for reading the menus from the resource file, and adding them to the menubar and then drawing the menubar. The CreatePalette() is for reading the palette from the resource file and then drawing the palette.

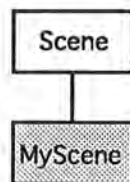


Figure 3.18 HaRTS Scene class hierarchy

### 3.2.4 The GraphicsView Class

The GraphicsView class of the framework is responsible for handling events related to the graphical objects shown inside a window. For example, select or

drag a graphical object inside a window, etc.. MyGraphicsView is the subclass of the GraphicsView class, in which new data fields and new methods have been added and some existing methods have been overridden.

In terms of the MVC paradigm, HaRTS class hierarchy is the domain specific model (M). HaRTS shape class hierarchy corresponds to the view (V). The GraphicsView class corresponds to part of the control (C). The other classes corresponding to the control include the Scene class, the Window class, the Menu class, etc.. These classes except the Scene class can almost be used directly and as a result, we omit discussing them here.

#### Instance Variables:

- fFirstSelectedShape    the first shape selected when the mouse is pressed down to add a new data/control arrow
- fSecondSelectedShape    the second selected shape when the mouse is up in adding a data/control arrow

#### Member Functions:

The following methods are new methods:

- CreateABox()            create a box
- CreateALabel()        create a label (object) and attach it to a box/arrow
- CreateADCArrow()      create an arrow
- CreateAnOperator()    create an operator
- AttachAnArrow()      attach an external arrow to a box/operao
- ChangeSource()        update the source of an arrow when the source of the arrow is changed
- ChangeDestination()    update the destination of an arrow when the destination of the arrow is changed
- ReadABox()            read a box from a design file
- ReadAnArrow()        read an arrow from a design file
- ReadAnOperator()     read an operator from a design file
- Flatten()             detect the flatten command, open a new window and then inform the selected composite box to flatten itself

- CleanUpWindow() clean up a window by recalculating the coordinates of the graphical objects so that the boxes are shown along the diagonal of a window

The following methods are overridden methods:

- UserNewShape() create the application-specific shape
- UserBeforeTrackMove() do application-specific initializations after the mouse is pressed down but before the mouse is moved, e.g., storing the initial position of the mouse, etc.
- UserDuringTrackMove() do application-specific actions after the mouse is pressed down and moved but before the mouse is up, e.g, drawing the rubberband of a shape, etc.
- UserAfterTrackMove() do application-specific actions after the mouse is up following a mouse down, e.g, creating a box or arrow depending on the current palette status, etc.
- UserCheckAction() determine the application-specific action type, e.g., creating a box or an arrow, etc.
- DoubleClick() keep track of user clicking actions to distinguish between single click and double clicks
- ReleaseMouse() determine the user action type, e.g., dragging, growing, etc.
- DoWrite() save the current design to a design file
- DoRead() read from a design file and recover the hierarchical design
- Cut() delete the selected object
- Undo() undo a command
- Redo() redo a command

## 4 Conclusion and Future work

We have described a hard real-time system design tool, HaRTS, which supports a hierarchical design diagram. The design hierarchy separates a design into self-contained subdesigns and yet, the design can be flattened to give a global view. In a distributed environment, the design hierarchy provides a natural way for assigning subdesigns to different processors. The design diagram combines the control and data flow of a hard real-time application and, as a result, is quite intuitive.

We have illustrated the HaRTS user interface and described the implementation of HaRTS. HaRTS has been implemented on Macintosh through specializing an object-oriented application framework, Objex. Our experience has demonstrated that Objex is quite useful in implementing GUI applications like HaRTS.

The framework helps me in developing object-oriented code which has better understandability. Essentially, every object you see on the screen has a direct correspondent in the program. This helps both programmers and maintainers understand the dynamic behavior of the program. The software reusability of Objex provides a powerful means for reducing software development cost and improving software quality.

I must point out that there are some detailed implementations which are not mentioned here. For each HaRTS specific class, in addition to the instance variables and methods which we have listed in section 3.2, there are still more instance variables and methods. For example, flattening a composite box is quite complex. In the box class description in section 3.2.1, we have simply mentioned the method FlattenABox(). Reading a design from a file is another example. Recovering the design hierarchy from a linear file is not easy. The interested readers can find more implementation descriptions in the header files of the program.

There is still much work left to be done to complete the framework. As pointed out in the introduction, the HaRTS design diagram can be automatically translated into Ada™ code and analyzed for scheduleability [11]. The current



implementation of HaRTS should be extended to incorporate code generation and design analysis.

We believe that a kind of software database storing frequently used algorithms for hard real-time applications, maybe in the form of a class hierarchy, will be very useful in reducing the cost of developing such applications. Once such a software database is added to our design tool, the functions specified in atomic boxes can be either retrieved from the database or directly input by programmers. As shown in section 3.2.3, there is a class scene reserved for this purpose.

Furthermore, the design diagram is essentially control-oriented. The variables associated with the data-in arrows of a box are read when a control stimulus reaches the box and the variables associated with the data-out arrows are updated when the box finishes its execution. To clearly see where each data is used, a different view, data view, of the same design is needed. Under this view, data access specifications are drawn around data stores. As shown in section 3.2.3, there is a data scene reserved for this purpose.

## References

- [1] A.K.Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", Ph.D. Thesis, Massachusetts Institute of Technology, May, 1983.
- [2] Jiang Zhu, T. G. Lewis, and Jean-Yves Colin, "Scheduling Hard Real-Time Constrained Tasks on One Processor", Tech. Rep. #93-60-16, Computer Science Dept., Oregon State University, June, 1993.
- [3] Jiang Zhu, T. G. Lewis, Weldon Jackson and Russel L. Wilson, "Design and Analysis of Hard Real-Time Applications in a Uniprocessor Environment", Tech. Rep. #93-60-17, Computer Science Dept., Oregon State University, June, 1993.

- [4] Liu Sha and John B. Goodenough, "Real-Time Scheduling Theory and Ada", *Computer*, Vol. 23, No.4, April 1990, pp 53-62.
- [5] T.P. Baker and A. Shaw, "The Cyclic Executive Model and Ada", *Real-Time Systems Symposium*, Huntsville, AL, Dec. 1988, pp. 120-129.
- [6] Ted G. Lewis, Huan-Chao Keh, Chung-Cheng Luo, "Software Design with Frameworks *Featuring The C++ Objex-by-Design Method*", Computer Science Dept., Oregon State University, 1992 .
- [7] Chung-Cheng Luo and Ted G. Lewis, "Oregon Speedcode Universe 3.0 Programming Manual", Tech. Rep. #91-60-13, Computer Science Dept., Oregon State University, 1991.
- [8] Walter I. Wittel Jr., "Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development", Tech. Rep. #91-60-6, Computer Science Dept., Oregon State University, 1991.
- [9] Qihui Ke, "The Scene Model:Extending Functionalities of the Objex Application Framework", Tech. Rep. #92-60-13, Computer Science Dept., Oregon State University, 1992.
- [10] Goldberg, Adele J. "Smalltalk-80:The interactive Programming Environment", Addison-Wesley, Reading, MA, 1984.
- [11] Jiang Zhu, "Design and Analysis of Hard Real-time System", Ph.D. Thesis, Oregon State University, Nov., 1993.
- [12] Ted G. Lewis, "CASE: Computer-Aided Software Engineering", Van Nostrand Reinhold, New York, 1991.

## Acknowledgements

I would like to thank my major professor, Dr. T.G. Lewis, for his guidance and discussions with me throughout the project.

My special thanks go to Chung-Cheng Luo, an expert on Objex, for his valuable help in implementing HaRTS.

My gratitude to my parents can hardly be expressed in words. Their encouragement and support are the real force that keeps me moving forward.

I am grateful of my husband, Jiang Zhu, for his confidence on me and his support for me to pursue my academia.