4

The Scout Notation

The simplest way of applying the Definition-based State-Transition (DST) model is to develop definitive systems similar in kind to simple spreadsheet software. In such systems, the only source of transition of state is via direct redefinition by the user. Definitive systems differ in the data types and operators that they employ in their definitions. Definitions of specific data types and operators are particularly suitable for specifying states in different applications. Each such notation is called a *definitive notation.*

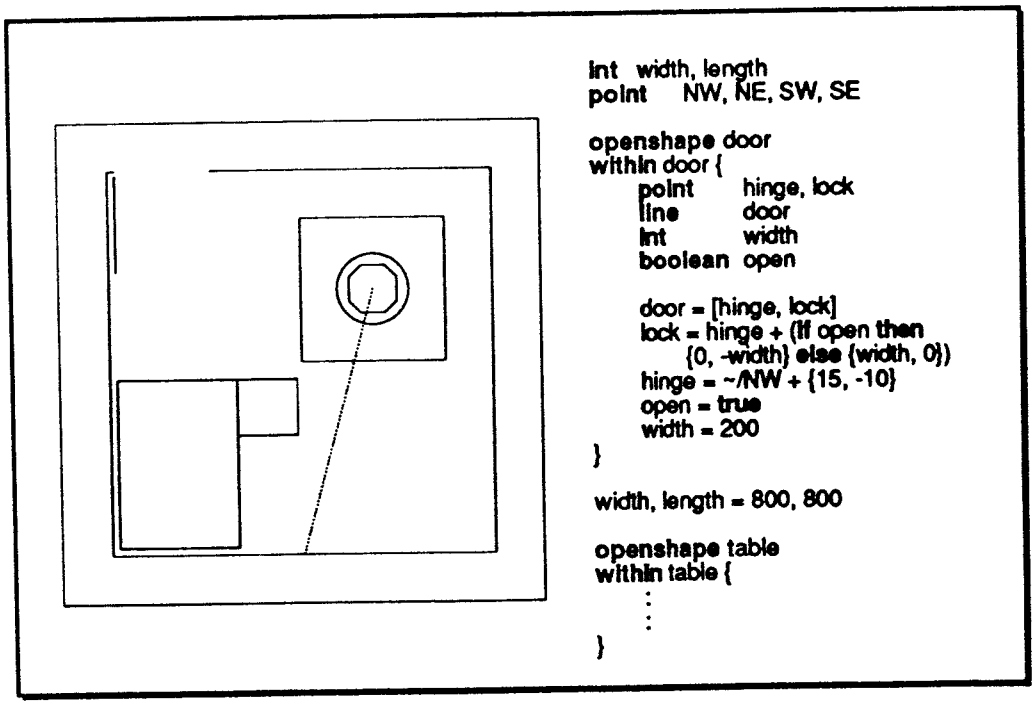
As explained in the last chapter, a typical definitive notation differs from an electronic spreadsheet in that variables in a definitive notation have their own names and are independent entities. Variable names in a spreadsheet on the other hand are signified by their geometric locations on a table of cells. As in a spreadsheet, a variable in a definitive notation can be defined explicitly by a value or implicitly by a formula in terms of other variables. A collection of declarations of variables and their definitions is called a *definitive script.*

Since every definitive notation has its own application domain, each definitive notation has its own right of existence. This chapter describes a definitive notation called *Scout* which I have designed and implemented. The development of this notation has made important contributions to our understanding of the modelling property of definitive notations and to the integration of definitive notations, but this chapter only describes Scout as a definitive notation for the fulfilment of its original function – describing the screen layout – and leaves the discussion of its other contributions to later chapters. The aim of this chapter is to introduce the Scout notation for further discussion later. Both the design and the implementation of the Scout notation will be described.

4.1. The DoNaLD Notation

The Scout notation is unique amongst the existing definitive notations in that it provides complementary display information for other definitive notations. To appreciate this role of the Scout notation fully, it is best to briefly introduce another definitive notation. The notation to be introduced is DoNaLD.

DoNaLD is a definitive notation for two dimensional line drawing. The notation was defined in 1986. The full specification of DoNaLD can be found in [BABH86]. The first prototype of DoNaLD was developed by Edward Yung in 1987, but not all features in [BABH86] were implemented. In subsequent enhancements ([Chan89, Parsons91]), some new data types and operators have been introduced, yet some features in the original specification remain unimplemented. However, the conceptual framework for definitive notations is sufficiently brought out by the current DoNaLD prototype. An illustrative example of DoNaLD, which is used many times in our publications, is a description of a room. Figure 4.1 shows a part of the room specification and the graphical visualisation of the entire room.



```
int  width, length
point    NW, NE, SW, SE

openshape door
within door {
     point      hinge, lock
     line       door
     int        width
     boolean    open

     door = [hinge, lock]
     lock = hinge + (if open then
          {0, -width} else {width, 0})
     hinge = ~/NW + {15, -10}
     open = true
     width = 200
}

width, length = 800, 800

openshape table
within table {
     .
     .
     .
}
```

Figure 4.1: DoNaLD Script of a Room

In the current DoNaLD implementation, there are eight data types: *integer*, *real*, *boolean*, *point*, *line*, *circle*, *ellipse* and *shape*. DoNaLD variables are typed variables. A *point* variable corresponds to a point on the screen; a *line* variable corresponds to a line. The same is true for *circles* and *ellipses*. A *shape* variable corresponds to a group of elementary graphical elements on the screen. Since *integer*, *real* and *boolean* are not graphical items, variables of these kinds do not have any visible form on the display.

The geometric aspects of the graphical items are determined by the values of the variables. The value of a *line* variable, for instance, prescribes the location of the two end-points on the screen. The DoNaLD system employs an orthogonal Cartesian coordinate system: the lower bottom corner of the screen is the origin, the x-axis goes from left to right and the y-axis goes upwards in the scale of one unit per pixel.

No part of the DoNaLD notation is dedicated to the presentation of the graphical items. That is to say, there is no formalised way of controlling, for example, the colour of a line in DoNaLD. Our convention is that a line in DoNaLD will appear to be a solid black line with unit thickness. In our current prototype, however, attributes can be

associated with a variable to control the line style (if it is a line) and colour. These attributes are also defined in the definitive style. Redefining the attribute of a variable will automatically take effect on the screen.

4.2. Motivating the Design of Scout

The name Scout is an abbreviation of 'SCreen layOUT'; the Scout notation is a definitive notation for describing screen layout. It is a notation concerning how to display information, i.e. a definitive state, on the screen. There are two main motivations for designing the Scout notation: to present the definitive state in a user-specified manner and to supplement the display information for other definitive notations.

4.2.1. Visualisation of State

The value of a variable is typically represented in a computer by a sequence of 0's and 1's. This binary representation is not particularly useful in high-level programming. Values need to take another form of representation in order for the user to understand. The value of a variable storing the room temperature has one thousand and one presentations: angular displacement of a meter, length of a bar, seven-segment digital readings, colour code, and so on. The choice of presentation method is up to the designer and sometimes can be selected by the users.

The role of definitive notations is to describe state. The external presentation of the internal state is determined by the implementation of the definitive systems. Current systems exploiting dependency typically use different ways of presentation for different types of variables. For example, in a spreadsheet, there may be cells (variables) that display textual strings and other cells that display numerical values; their values will be represented on the display by strings of characters and strings of digits respectively. DoNaLD provides other examples: a *point* variable with value {x, y} will appear as a dot on position x steps right and y steps up from the origin; a *line* variable with value

[{x1, y1}, {x2, y2}] will be represented by (what appears to be) a collinear set of dots on the screen. Of course, the string "{10, 20}" or *a dot on screen* are alternative presentations of the same value, but obviously, displaying a dot in a particular location may be much more appropriate than displaying a string anywhere on the screen. However, this is not enough to justify the way current definitive systems operate: fixing the presentation formats of the variables during the design and implementation of the system.

We do not always want values of the same type to have the same presentation. This becomes obvious when we compare the representations of the same data types in two different notations. Take the *integer* type as an example: an integer value is not displayed on the screen in DoNaLD, but in another context, such as a spreadsheet, a string of digits will be displayed in a cell that records an integer value. Even within the same definitive notation, there are cases when we would like to see different (variants of the same) representations for the same type of values. Lots of examples can be found from the DoNaLD notation: in a floor plan, we may like the line denoting a wall to be thicker than the line denoting a door; we may like to use dotted line to represent some flexible linkage, say an electric cord; or we may like to define a box whose corners are specified with reference to the centre of the box but we do not want to see a dot in the middle of the box (i.e. the visualisation of the variable of the central point).

The Scout notation addresses the problem of presentation of data by using definitions to describe the output formats of a variable. With definitions, a persistent link between the internal model and its external representation is achieved. The observed changes of variables can be synchronized with internal state changes. Scout definitions are therefore performing a function analogous to the format rules in the spreadsheets. In fact, Scout allows more flexible control over the output format.

4.2.2. Assumptions of Definitive Notations

Most hardware primitives cannot be altered by software. For example, the location of every pixel on the physical display is determined by the manufacturer. The correct interpretation of a definitive notation has to take into account the particular hardware characteristics. The user of a definitive notation cannot interpret a definitive script precisely unless he knows about the assumptions made by the notation designer. There has to be in effect a "contract" between the notation designer and the programmers – perhaps specified external to the system through a manual.

The screen described in the Scout notation is actually not the physical screen. It is, in principle, describing an imaginary screen. As for other definitive notations, there is a mapping from the imaginary screen to the physical screen. In the design of Scout, the intention is to create a close correspondence between the two screens – for instance, a point in the imaginary screen should map to a point on the physical screen so that Scout can use the maximum resolution of the display unit. However close the correspondence is, clarification beyond the scope of the notation is still required. For instance, a point in a TTY screen has a slightly different interpretation from a point in a workstation – a point in a TTY screen is large enough to display a character but a point in a workstation cannot.

When writing programs, the programmer may prefer having computer hardware with particular physical characteristics. It is our hope that we can, by means of the definitive notation Scout, create an interface between the programmer's preferred environment and the environment provided by the actual machine. This can be illustrated with reference to DoNaLD.

Writing a DoNaLD specification has to take into account assumptions about the physical output device in use. Such assumptions include the size and the resolution of the screen and the location of the origin. Consider, for instance, the task of specifying

a square in the middle of the screen. Listing 4.1 is a plausible specification if the origin is located at the centre of the screen.

```
openshape square
within square {
      line   N, E, S, W
      integer size
      size = 100
      N = [{size,size}, {-size,size}]
      E = [{size,size}, {size,-size}]
      S = [{-size,-size}, {size,-size}]
      W = [{-size,-size}, {-size,size}]
}
```

Listing 4.1: A DoNaLD Description of a Square

If the origin is located at the bottom-left corner, shifting has to be done in order that we can see the whole square. A variable centre may be added to Listing 4.1 to do the shifting (see Listing 4.2). However, the value of centre (say {300, 200}) cannot be written down without prior knowledge of the size of the display (say 600×400). Moreover, the output on the display will genuinely be a square only if the vertical and the horizontal axes carry the same resolution.

```
openshape square
within square {
      line N, E, S, W
      integer size
      point centre
      size = 100
      centre = {300, 200}
      N = [{size,size}+centre, {-size,size}+centre]
      E = [{size,size}+centre, {size,-size}+centre]
      S = [{-size,-size}+centre, {size,-size}+centre]
      W = [{-size,-size}+centre, {-size,size}+centre]
}
```

Listing 4.2: Another DoNaLD Description of a Square

Scout is a definitive notation for describing screen layout. It enables us to put down the assumptions about the required display so that the programmer can work on an imaginary screen that suits his purpose. Listing 4.3 illustrates how the Scout notation can be used to control the realisation of an image onto the physical screen. The top half of Listing 4.3 is the DoNaLD description of a square defined in a preferred

environment (i.e. Listing 4.1); the bottom half shows a Scout window in which the mapping between the preferred environment and the actual display is defined. The first three attributes of the specification of the sqr window – *type*, *box* and *pict* – specify that the DoNaLD picture "don" is displayed in the region prescribed in the box with the two opposite corners {0, 0} and {600, 400}. The attributes *xmin*, *ymin*, *xmax* and *ymax* describe the coordinate system within this window. The window sqr can display the region bounded by the lines x = xmin, y = ymin, x = xmax and y = ymax. In Listing 4.3, *xmin*, *ymin*, *xmax* and *ymax* are defined in such a way that the origin of the DoNaLD picture corresponds to the centre of the window and one unit in the DoNaLD picture corresponds to one pixel in the actual screen display.

```
%donald                    // beginning of DoNaLD script
viewport don               // name of the DoNaLD drawing
openshape square
within square {
      line N, E, S, W
               integer size
               size = 100
               N = [{size,size}, {-size,size}]
               E = [{size,size}, {size,-size}]
               S = [{-size,-size}, {size,-size}]
               W = [{-size,-size}, {-size,size}]
}
%scout                     // beginning of Scout script
window sqr = {
      type: DONALD,
      box:  [{0,0}, {600,400}],  //geometry of the window
      pict: "don",         // name of the DoNaLD picture
      xmin: -300,          //
      ymin: -200,          // geometry of the imaginary
      xmax: 300,           // DoNaLD display
      ymax: 200            //
}
```

Listing 4.3: A Sample Scout Fragment

With Scout, other definitive notations, such as DoNaLD, can be thought of as working with an imaginary screen that has idealised properties such as unbounded size, unlimitedly fine resolution and infinite number of colours. This imaginary screen concept is very important in ensuring that models can be built with minimal hindrance.

The output of the definitive script is obtained through a mapping from the imaginary screen to a physical screen. The actual output is different from the imaginary one both because there are constraints in the physical screen and because we may like to set some limits on the actual output (for example, to show the region bounded by the lines x = 0, x = 1000, y = 0 and y = 1000 of a DoNaLD picture in a particular region of the physical screen with resolution of 4 square units per pixel). Definitive notations like DoNaLD and ARCA do not give the user control over how the imaginary screen maps to the physical screen. For this reason, assumptions about the mapping have to be fixed at the implementation stage when these notations are to be used on their own. The Scout notation, on the other hand, presents supplementary information about the mapping for other definitive notations so that more control over the output is possible.

4.3. The Design of Scout

A preliminary design for the Scout notation was described in my final year undergraduate project report [Yung88]. In its original form, Scout was designed for laying out text; it was subsequently enhanced to interface with DoNaLD and ARCA.

The choice of the Scout notation primitives is based on the assumed nature of the screen and the manner in which it will be used in particular applications. For instance, a newspaper layout may require multi-column format, whilst rectangular boxes suffice for a typical window-based application. For these tasks, high resolution graphical display is a reasonable assumption. Provision for colour display is also preferred.

Complicated drawings are assumed to be handled by other definitive notations. In fact, the purpose of designing different definitive notations is to ensure that different kinds of application can be addressed in appropriate notations. DoNaLD, CADNO [Stidwill89] and ARCA are some definitive notations designed for describing different kinds of graphics. Other definitive notations for describing graphics are also

conceivable. It is not our intention and it is not appropriate to use Scout to describe every single detail of what the screen will display. Scout, therefore, does not provide a set of drawing primitives but is intended to deal with simpler tasks such as screen layout, scaling and other operations at the pixel level. Scout's special role is to specify where and how these images described by other definitive notations are displayed. In principle, the role of Scout should be confined to coordinating the use of different definitive notations to form a display, but because there is no definitive notation for displaying text so far and text is almost indispensable in any serious application, a part of the Scout notation is concerned with the display of text strings. However, it is intended to develop other definitive notations to deal with more complicated text displaying tasks such as those encountered in desktop publishing or word processing applications.

4.3.1. The Window Data Type

Layout design in Scout is based on the answers to the following three questions: What are the things to be displayed? Where are they to be displayed? And how are they to be displayed? This naturally leads to the concept of the Scout *window* data type. The type *window* is a union of subtypes: one subtype is designed for each definitive notation. Each subtype has a number of fields. The number of fields and the types of the fields may vary depending on which notation this subtype refers to. Generally speaking, a *window* should have fields that define

1) which definitive notation is concerned;
2) what information (e.g. which DoNaLD picture or which text string) is to be displayed;
3) the region of the screen onto which the required information is mapped;
4) the supplementary information that that definitive notation needs.

In the current design and implementation, there are three types of windows – the text window, the DoNaLD window and the ARCA window.

Text Window		
field name	*type*	*description*
type	content[1]	Must be the value TEXT
string	string	The string to be displayed
frame	frame	The region in which the string is shown
border	integer	Width of the border of the boxes of the frame
alignment	just[2]	NOADJ, LEFT, RIGHT, EXPAND and CENTRE are the possible values to denote no alignment, left justification, right justification, left and right justification and centre of the text inside each box in the frame
bgcolour	string	Colour name for the background colour of the text
fgcolour	string	Colour name for the (foreground) colour of the text

where point = integer × integer

box = point × point

frame = *list of* box

DoNaLD Window		
field name	*type*	*description*
type	content	Must be the value DONALD
box	box	The region in which the DoNaLD picture is shown
border	integer	Set the border width of the bounding box
pict	string	The name[3] of the DoNaLD picture
xmin	point	Show the portion of the DoNaLD picture bounded by the points (xmin, ymin) and (xmax, ymax)
ymin	point	
xmax	point	
ymax	point	

---

[1] The type *content* is currently a set { TEXT, DONALD, ARCA }.

[2] The type *just* is { NOADJ, LEFT, RIGHT, EXPAND, CENTRE }.

[3] There is no picture name specified in the original DoNaLD notation, but there is now a statement

```
viewport name
```

required before the DoNaLD definitions to identify which picture these definitions are defining.

• The ARCA window is exactly the same as the DoNaLD window except that the type of window should be declared to be ARCA.

Comments on the window data types:

1) The definitions of the window subtypes above are very simple. Many more attributes, such as background pixmap, font of string and so on might be used to control the appearance of the windows. Actually, there is no real reason why those attributes cannot be included into the Scout windows. The attributes listed above are chosen simply because they are the most commonly used ones. Introducing more attributes reduces the number of defaults that are built into the interpreter, giving the user a higher degree of control over window specification.

2) There is no formal restriction on how to define a region. Nevertheless, the choice of method should be governed by the nature of application. The three available subtypes have already employed two ways of defining regions. In a DoNaLD or ARCA window, a region is defined by a box, whereas in a text window, a region is defined by a list of boxes. A single box is good enough to frame one picture but a list of boxes is required if a long passage of text is to be displayed in multiple columns. Other methods of defining regions might also be employed. For reference, the X Toolkit [MAS88] is primarily designed for rectangular windows; PostScript [Adobe85] defines an arbitrary shaped region by a closed path; a bitmap is commonly used to define a small region such as the shape of an icon.

3) Notice that there are almost no fields in common between the graphics window subtypes and the text window subtype, so the type *window* may be better understood by the abstract formula

$$\text{window} = \text{region} \times \text{content} \times \text{attributes}$$

rather than by a concrete set of fields.

4.3.2. The Display Data Type

A *display* is a collection of windows. Because the windows may overlap, there is a partial ordering among the windows. For simplicity, a *display* is defined to be a list (total ordering) of *windows*.

In general, a *display* variable represents a conceptual screen; only the distinguished *display* variable *screen* denotes the physical screen. There are simple rules for mapping the variable *screen* onto the physical screen:

1) the origin is defined at the top-left corner of the physical screen;

2) the x-coordinate counts from the origin to the right, one unit per pixel;

3) the y-coordinate counts from the origin to the bottom, one unit per pixel.

It is obvious from the mapping rules that the interpretation of the Scout notation, unlike other definitive notations, is hardware dependent. The same script of Scout definitions may have a slightly different look on a monitor with different resolution and aspect ratio.

4.3.3. Other Data Types and Operators

Because there is a great flexibility in the design of the window data type, the set of data types and operators in Scout may be extended in the future. There are, however, some essential data types in Scout: *integer*, *point*, *window* and *display*. Associated with them are basic operators for integer arithmetic, vector manipulation, list manipulations, construction and selection. The following table shows the basic Scout operators and functions for the four essential data types. All the operators of the Scout notation can be found in Appendix A.

Operators:	+, –, *, /, % (remainder), – (unary minus)
Meaning:	Normal integer arithmetics
Example:	10 % 3 gives 1
Constructor:	{*x*, *y*}
Meaning:	Construct a point
Example:	{10, 20} is a point with x-coordinate 10 and y-coordinate 20
Operators:	+, –
Meaning:	Vector sum and vector subtraction
Example:	{10, 20} – {20, 5} gives {–10, 15}
Selector:	.1, .2
Meaning:	Return the 1st (x-) coordinate and the 2nd (y-) coordinate resp.
Example:	{10, 20}.1 gives 10
Constructor:	{*field-name*: *formula*, *field-name*: *formula*, ..., *field-name*: *formula*}
Meaning:	Constructing a window
Example:	{ type: DONALD, box: b, pict: "figure1" }
Selector:	*.field-name*
Meaning:	Return the value of the field
Example:	{ type: DONALD, box: b, pict: "figure1" }.box gives b
Constructor:	< *W1* / *W2* / ... / >
Meaning:	Constructing a display, if *W1* and *W2* overlap, *W1* overlays *W2*
Example:	< don1 / don2 >
List function:	insert(*L*, *pos*, *exp*)
Meaning:	Insert the expression *exp* in the position *pos* of list *L*
Example:	insert(<w1, w2, w3>, 2, new) gives <w1, new, w2, w3>
List function:	delete(*L*, *pos*)
Meaning:	Delete the *pos*th element of list *L*
Example:	delete(<w1, w2, w3>, 2) gives <w1, w3>
Operator:	if *cond* then *exp1* else *exp2* endif
Meaning:	if *cond* gives non-zero value (true) then returns *exp1* else returns *exp2*, in this context,*exp1* and *exp2* must have the same type.
Example:	if 1 then "Open" else "Close" endif gives "Open"

As mentioned, text layout should ideally be described by another definitive notation. Since that notation does not exist, part of the Scout notation is designed for simple text layout. To this end, Scout incorporates a *text window* subtype. This text window subtype differs from other window subtypes in that the content of the text window subtype is a string defined within Scout rather than a virtual screen prescribed outside Scout by another definitive notation. As a result, *string* becomes one of the Scout data types.

Associated with the *string* data type is a set of operators useful for displaying a text string. String concatenation (//), string length function (strlen), sub-string function (substr) and integer-to-string conversion (itos) are the basic Scout string manipulation functions. There are two postfix operators – .r and .c – which are specially designed. Since the basic geometric unit in Scout is the pixel but the size of a block of text is more conveniently specified as "number of rows by number of columns", it is convenient to introduce functions returning the row height and the column width in pixels. .r is the function meaning "multiply by the row height" and .c is the function meaning "multiply by the column width". These functions are appropriately represented by postfix operators because they work very much like units. For example, {10.c, 3.r} refers to a point 3 rows down and 10 columns right to the origin. A similar consideration influences the design of a *box*, a data type for defining regions. The region associated with a box is sufficiently defined by its top-left corner and its bottom-right corner, and this is a convenient method of definition in the case of graphics. For a block of text, however, the bounding box is more conveniently defined by specifying the top-left corner and the dimensions of the box in terms of number of rows and columns. For instance, [{0, 0}, 3, 10] refers to a box with the origin as its top-left corner which is suitable for displaying three rows by ten columns of text. More examples of this kind can be found in the Jugs example in the next chapter.

Because displaying a string is different from displaying an image, the way of specifying a region for displaying text is different from that for displaying an image. As attested by the fact that the earlier releases of X Window system version 11 had only primitives for creating rectangular windows, a simple box is adequate for display purposes in most applications. But, when considering the possible application of text display to desktop publishing, we know that a piece of text may be displayed across several regions; defining a region by a box is simply not sufficient.

One proposal is to define a region by a closed path (a wire-frame). This can create an arbitrary shape but this will also cause problems filling in the string. Consider the following *frame*:

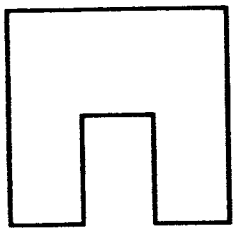


Figure 4.2: A Non-rectangular Region

There are two possible and equally natural ways of filling in a string as illustrated in Figure 4.3a and 4.3b.

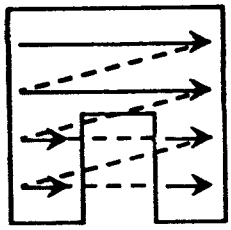


Figure 4.3a

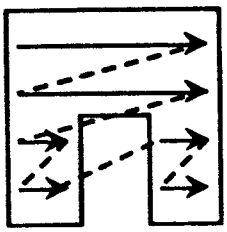


Figure 4.3b

Therefore, this way of defining region leads to ambiguity. Another ambiguity arises when a frame comprises two or more discrete closed paths. It is then unclear which closed path the string should fill first. Our solution is to divide an arbitrary shape into subregions, each of which is a box. The definition of a region will then be a ordered

list of boxes. For example, the region depicted in Figure 4.4 is interpreted in such a way that a string should be filled in the first box first, then the second, then the third.

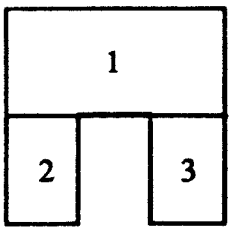


Figure 4.4: A Way of Partitioning a Non-rectangular Region

This "frame = list of boxes" definition of region is not perfect. For instance, if two boxes overlap (which may depict the overlapping of two sheets of the same document), which box should be put on top is still ambiguous. However, except for serious desktop publishing, this definition of region should be adequate for most applications.

4.4. The Implementation of Scout

Scout is implemented using a previously developed definitive language, EDEN [Yung89]. EDEN – an abbreviation of "Engine for DEfinitive Notations" – is intended to assist the implementation of definitive notations, though it is also a general-purpose programming language in its own right.

A unique combination of language features makes EDEN the most suitable tool for implementing other definitive notations. EDEN has:

- C-like syntax and operators. That is, EDEN has a rich set of efficient programming structures and operators.
- list structure. Complex data types can be simulated using lists.
- user-defined functions and procedures. These are essential for simulating operators in the definitive notations.
- definitions. EDEN provides automatic maintenance of definitions. The values of the EDEN variables will always be kept up-to-date using the dependency

information implicit in the formulae of the variables. Writing a translator for translating definitions in a definitive notation to EDEN definitions effectively creates an interpreter for that definitive notation. But writing a translator is much simpler than writing a definition maintainer.

- actions. EDEN actions are procedures whose execution is triggered by the changes to specified variables rather than being invoked explicitly by another procedure or the user. In implementing Scout, the EDEN actions are used to update the external presentation of the changed values of variables. The function of the EDEN actions can be depicted in the following figure.

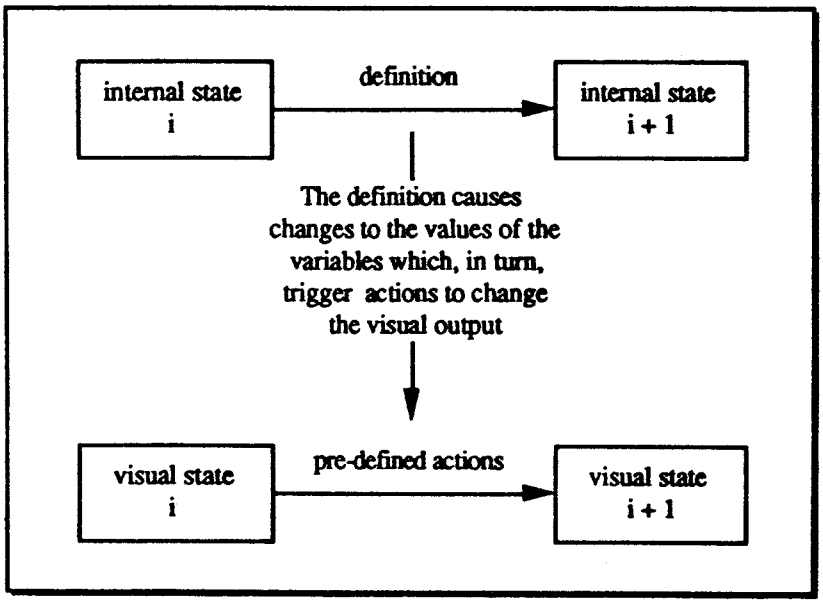


Figure 4.5: The Role of EDEN Actions in the Implementation of a Definitive Notation

As a simple illustration,

```
proc update_A : A
{
      writeln("A is ", A);
}
```

defines an action update_A. It will be invoked whenever A changes value. This action will print out a line expressing the new value of A.

The implementation task for Scout is to create a translator from Scout to EDEN. The translation is carried out according to the scheme to be discussed in Chapter 6,

where issues concerning the integration of definitive notations are considered. The implementation involves writing an EDEN library to simulate the Scout data types and operators and writing a program for translating Scout definitions to EDEN definitions.

The integer data type in Scout and the arithmetic operators have direct equivalents in EDEN. The point data type is an ordered pair (a list of two integers). The box is fundamentally a pair of points, there is however a need to write an EDEN function for the point-row-column form of defining boxes. This function will take in a point and two integers as its arguments and return a box. Both the frame data type and the display data type are lists. The window data type is simulated by a list of the union of all the attributes of all the window sub-types. These attributes are enumerated data types, which can be represented by integers, strings, a primitive data type in EDEN, or one of the data types already mentioned. In this way, all the data types are directly or indirectly supported by the EDEN primitives. Hence the translation of Scout definitions into EDEN definitions is fairly straightforward.

The last part of the implementation involves the writing of the actions for maintaining the visual display. There is only one variable in Scout – screen – that has a visual representation. This is because only the display variable screen represents the physical display screen. For this reason, only one EDEN action is needed in the implementation and it is triggered by the variable screen.

There have been two versions of the Scout implementation. The first version was developed on the SunView window system and second on the X Window system. The SunView version uses the WW library from Rutherford Appleton Laboratory [Martin87] so as to simplify the implementation. The WW library is chosen because it has a function to format a string within a box. The reason for redeveloping Scout for the X Window system will be discussed in detail in Chapter 6. Because the X Window system does not provide that string displaying function, a similar function has to be written. In fact, we have built a separate program to interface between EDEN and the

X Window system. In this interface program, commands like creation of a box, displaying a string in a box and other line drawing commands are available. This interface program will be discussed in more detail in Chapter 6.

The SunView version is an early version; it does not have provision for combining the use of several definitive notations; the only window subtype is the text window. In the X Window version, graphics described by other definitive notations can be mixed in the same screen display. As mentioned, the Scout interpreter is not intended to draw the details of the pictures of other definitive notations. Rather, the Scout interpreter generates an X-window for each Scout graphics window. Scout controls the size and the scaling factors of these windows and the DoNaLD and ARCA interpreter draw the details in them. In other words, DoNaLD and ARCA are drawing on a transformed space whose transformation is determined by Scout – this corresponds to what conceptually Scout should do.