6

Integrating Definitive Notations

Although a simple definitive notation may be good at addressing a particular aspect of an application, a stand-alone definitive notation has limited usage. Dealing with a large problem often requires the cooperation of several definitive notations. This chapter describes and evaluates our effort to integrate several definitive notations into a single system. The essence of the integration is the design and implementation of the Scout notation so that pictures described by other definitive notations can be combined to form a screen display. For this reason, we have called our effort at integrating definitive notations *the Scout Project.*

6.1. Motivation for the Scout Project

A motivating example of visualisation in mathematical research is considered. This example is based upon [BYAB91], where the mathematical context is discussed in

greater detail. In particular, the example illustrates the visualisation of combinatorial structures associated with arrangements of four lines.

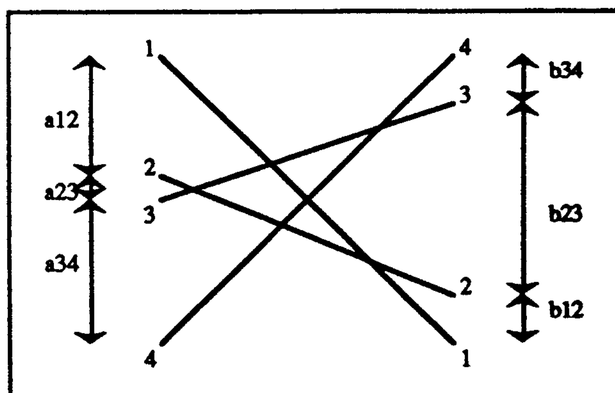

Figure 6.1: An Arrangement of Four Lines

In Figure 6.1, the line arrangement represents a sequence of transpositions of the permutation 1234 to 4321. The sequence can be obtained by interpreting the crossings of the top, the middle and the bottom pair of lines as transpositions of the first, the second and the third line pairs. In this case, on scanning the arrangement from left to right, the crossing sequence is then 213231. The crossing sequence corresponds to a shortest path, or a *geodesic*, in the Cayley diagram for the symmetric group S_4 as depicted in Figure 6.3. Two geodesics that differ only in the order of disjoint transpositions, such as 213231 and 231213, are equivalent. A *poset* to represent its equivalence class can be derived from a geodesic. Figure 6.2 is the poset representing the equivalence class of the geodesic 213231.

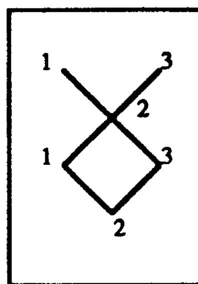

Figure 6.2: A Poset Representing the Line Arrangement in Figure 6.1

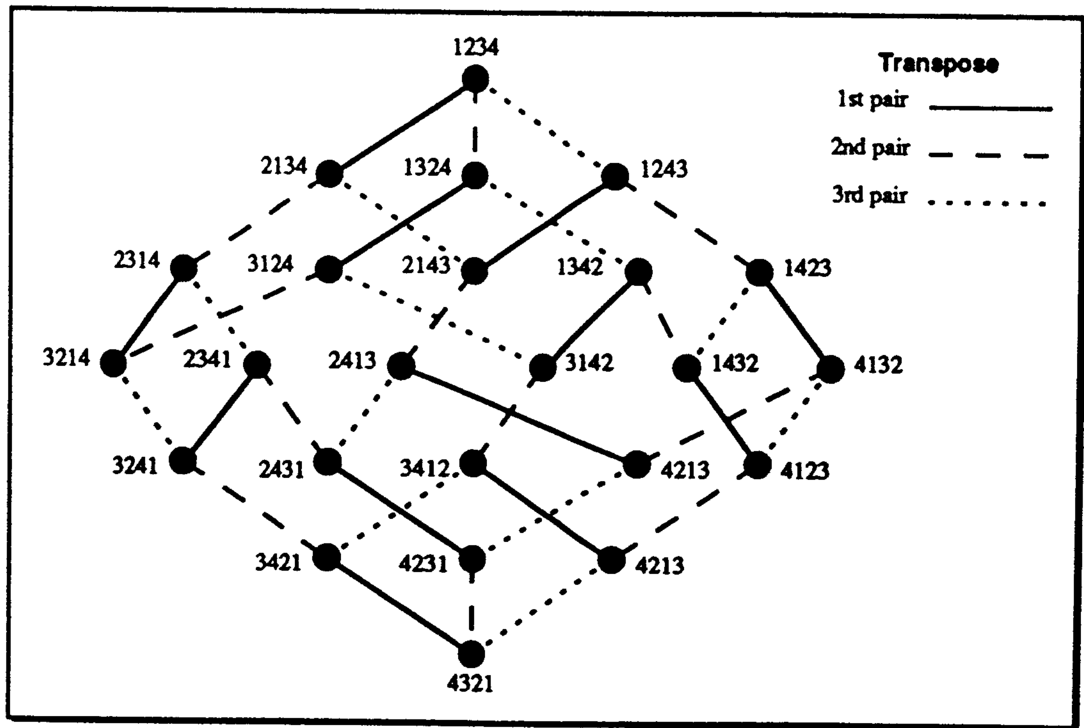

Figure 6.3: An S_4 Graph

A typical mathematical research activity is to explore the relationships between the line arrangements, the posets and the geodesics by varying the ratios a12:a23:a34 and b12:b23:b34. Therefore, related figures similar to Figure 6.1, 6.2 and 6.3 have to be displayed at the same time. This what-if exercise is conveniently dealt with by definitive notations. However, this visualisation problem does pose two challenges concerning the use of definitive notations.

The first challenge is the choice of definitive notations to describe the figures. We have DoNaLD, a definitive notation for line drawing, which suits the display and manipulation of the line arrangements. We also have ARCA, a definitive notation for displaying combinatorial diagrams, which is most suitable for visualising posets and Cayley diagrams such as S_4. But when we think of how to transform a poset into a group of geodesics of displayable form (such as the string "<213231, 231213>"), we need more powerful and more general functions and data types than those available in both DoNaLD and ARCA. In the last chapter, we explained that one of the advantages of definitive notations is their being specific in design. We cannot, therefore, expect to

build a definitive notation which has general data types and operators and, at the same time, includes all the special data types and operators of DoNaLD and ARCA. Realistically, variables in one definitive notation should be able to reference those of another definitive notation. Two conclusions can be drawn:

1) In the definitive paradigm, the way one variable relates to another is expressed by means of a definition of the variable written in terms of the other. A variable in one notation should, therefore, relate to a variable in another notation by means of definition as well.

2) Because we need to set up references across definitive notations, and these linkages are fundamentally definitions involving variables from different definitive notations, all the variables should be put into a common definition store. This implies that the definitive notations should be implemented in such a way that all the definitions will be translated into a common form.

The second challenge presented by the visualisation problem is the need for organising the display. DoNaLD and ARCA at their early stages considered only the display of line drawings or combinatorial diagrams in isolation. How these pictures related to each other and the wider considerations for integrated use in an application were ignored. This means that both DoNaLD and ARCA generate independent pictures on the screen. But in this context, several pictures are going to be displayed in the same application. Comments and labels are also needed to identify and explain them. Another definitive notation for describing screen layout is essential.

In addition to these two challenges, a definitive notation for general mathematical computation and string manipulation is obviously needed to complement other special-purpose definitive notations. EDEN can be this notation provided it is used in a disciplined way. This is because EDEN has imperative features as well as definitions.

There have been no previous attempts to integrate definitive notations. The definitive notations that are currently available have not been designed and implemented with integration in mind. The aim of the Scout Project is to build bridges between definitive notations so that they can be evaluated harmoniously within a single system.

6.2. Scope of the Scout Project

The scope of the Scout Project is:

1) to design and implement a definitive notation for describing screen layout – Scout;

2) to modify the implementations of the current definitive notations so that they can be evaluated together;

3) to provide guidelines for future development of definitive notations.

The design and implementation of the Scout notation has already been discussed in Chapter 4. The rest of the chapter will concentrate on the general framework and other guidelines for implementing definitive notations.

6.3. Implementation of Definitive Notations

Spreadsheets give useful insight into the definitive paradigm. In a spreadsheet, each cell is a variable of type string or number. By defining a cell with a formula, the system will automatically recalculate the formula whenever any cell it depends on is changed. The up-to-date value of the cell is then displayed in the cell. We can imagine that there are implicit actions which update the values of the variables on the screen. Similarly, there are such implicit actions in the implementations of definitive notations. For example, each graphical object in DoNaLD has a representation on the screen. For a point variable there is a dot to represent it; for a line variable, there is a linear set of points and so on. So there will be a plot_point action in the implementation of DoNaLD

to be called automatically when a point variable is updated, and likewise a plot_line action for a line variable.

Section 6.1 indicates that a general-purpose definitive language is required so that any definitive notation can be translated into it. Besides, as we have just explained, this language should also allow user-defined actions to be defined for displaying variables of different data types. A definitive language satisfying both requirements is EDEN (cf. [Yung89]).

6.3.1. Steps for Implementing a Definitive Notation

The following steps must be taken for implementing a definitive notation in EDEN. This method has been used for implementing Scout and DoNaLD. Among the two notations, the implementation of DoNaLD will illustrate the implementation method more clearly. Therefore, the examples associated with each step are all taken from the implementation of the DoNaLD notation.

1. Derive a scheme for translating variable names into EDEN variable names. For example:

DoNaLD name	EDEN name
table	_table
table/drawer	_table_drawer
table/drawer/width	_table_drawer_width

2. Emulate the data types and operators using EDEN data types and user-defined functions. Almost inevitably this will make use of the list structure in EDEN because list is the only complex data type in EDEN. For example:

DoNaLD type	EDEN type
integer	integer
point	['C', integer, integer]
line	['L', point, point]

DoNaLD operator	EDEN operator/function
div	/

+ (vector sum)	func vector_add {
	para p1, p2;
	return ['C', p1[1] + p2[1], p1[2] + p2[2]];
	}

3. The underlying algebra of the target notation has been implemented through steps 1 and 2. To complete the implementation, the required implicit actions are emulated using EDEN's user-defined actions. For example:

DoNaLD code	EDEN action specification
integer i	No action[1]
point p	proc P_p: _p { plot_point(&_p); }[2]
line L	proc P_L: _L { plot_line(&_L); }

4. Write a preprocessor to translate scripts in the definitive notation into EDEN in the way implicitly defined by steps 1 to 3.

6.3.2. Run-Time Structure

The run-time structure of the implementation of a typical definitive notation such as is described in §6.3.1 is depicted in Figure 6.4. The library contains the EDEN implementation of the underlying algebra together with the functions and procedures useful for the EDEN actions.

---

[1] No action required because integer variables do not have any graphical representation in DoNaLD.

[2] This & operator is similar to that in the C language, it returns the address of the variable. Plot_point and plot_line are EDEN (user-defined) procedures which do the plotting.

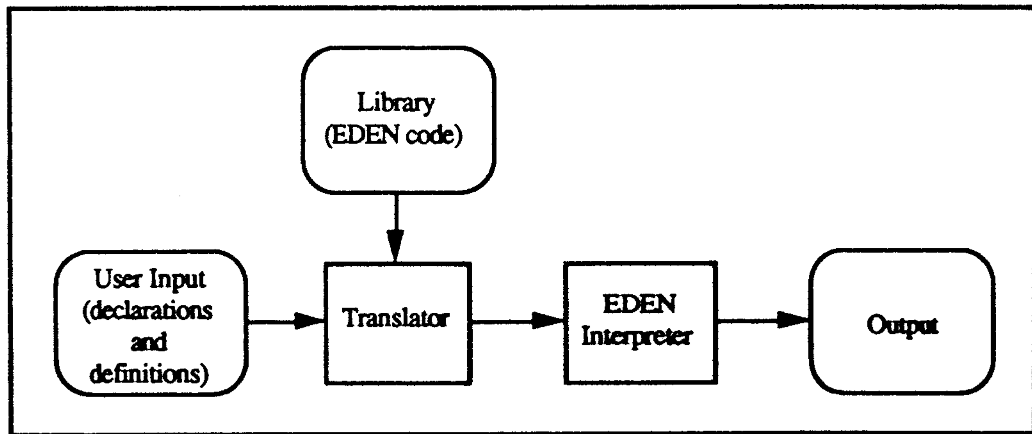

Figure 6.4: Run-time Structure of a Definitive System

The integrated Scout system is implemented in this fashion. Figure 6.5 shows the run-time structure of the Scout system.

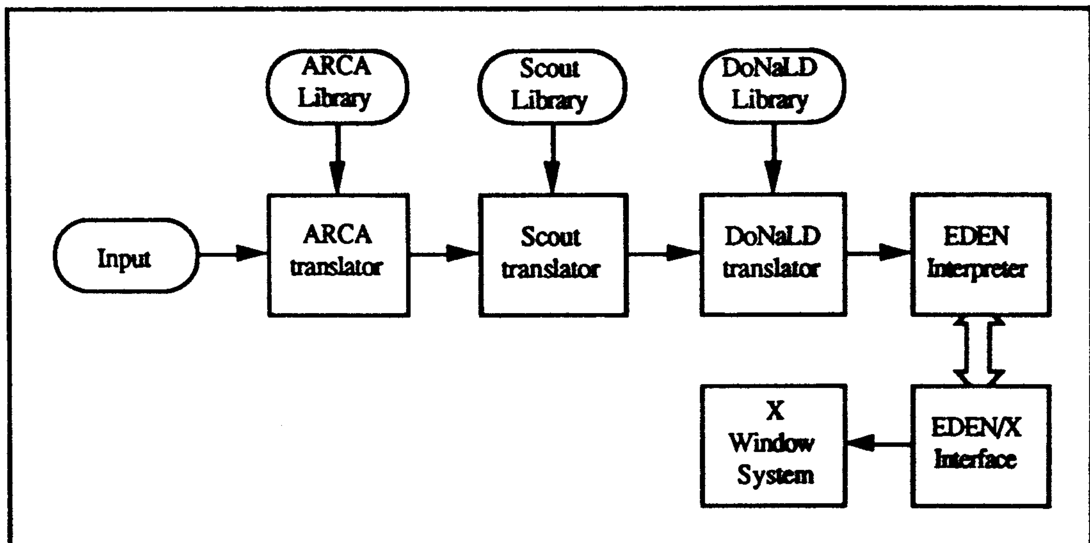

Figure 6.5: Run-time Structure of the Scout System

The Scout system is implemented in a UNIX environment. The user's input is pipelined through a series of ARCA, Scout and DoNaLD filters (the ordering of the filters is not important) which translate ARCA, Scout and DoNaLD definitions into EDEN definitions. These definitions together with the functions and actions in the libraries are interpreted by the EDEN interpreter. Graphical outputs are generated by an EDEN/X interface which is actually an X Window client. When the graphics display needs to be updated (i.e. some EDEN actions generate graphics output), the EDEN

interpreter will interact with the EDEN/X interface, which in turn will interact with the X Window server to produce graphics images.

6.3.3. The Choice of Graphics Interface

In the first implemented definitive notation – ARCA, output was generated via a plotter. After EDEN had been developed, more definitive notations were prototyped. At that time, only the SunView window system [Sun84] was available and hence DoNaLD, ARCA, CADNO and the preliminary version of Scout were developed on SunView. While other definitive notations used the SunCore window library for graphics display, the original Scout used the WW library from Rutherford Appleton Laboratory [Martin87] to simplify the implementation. At that stage, interfacing a graphical library required the inclusion of a large number of graphics library functions and routines as part of the EDEN built-in functions. This meant that a few versions of EDEN were created, each customized for one particular graphics library. Moreover, the definitive notation implementation libraries had to be developed for each version of EDEN. All these made maintenance of the system difficult. Later, two more powerful network window systems became available – the NeWS window system [Sun87] and the X Window system [GO90, MAS88]. In view of the coming thrust of more powerful and standard window systems, changing the window environment was unavoidable. Instead of creating yet another version of EDEN, a more flexible scheme for interfacing with the window system was derived – a graphics interface was separately built. Such separation is a standard issue in user interface management systems [HH89, Foley87, Gray89] and it has a number of advantages:

1. The graphics interface and the EDEN Interpreter can be run in parallel.

2. Other graphics interfaces may be built without modification or enhancement to the EDEN language or other definitive notation implementation library.

3. The graphics interface is reusable by other systems, not necessarily definitive systems.

The X Window system was chosen for our current implementation in preference to NeWS. This is because NeWS is written in PostScript whereas our prototypes were all written in the C language and, more importantly, NeWS was a buggy system. The choice proved wise because X Window is now more popular than any other window system. Currently, our system is supporting X version 11 release 5.

The EDEN/X interface program, called EX, was developed by me. It is linked with EDEN via a *message queue*. *Message queue* is a UNIX System V inter-process communication channel. It allows multi-directional communication between a number of processes. EX accepts a simple command language for creating windows, drawing graphics primitives, displaying text and processing queries such as the size of font in use. So long as this command language does not change, no alteration to the implementations of the definitive notations is needed even if the implementation of EX changes.

6.4. Other Guidelines for the Design and Implementation of Definitive Notations

6.4.1. Bridging Definition

As suggested in §6.1, bridging definitions provide a way of communicating between definitive notations. For example, we can use Scout to specify textual annotations of a DoNaLD picture. Listing 6.1[3] is an example of this kind. The information shown in the window doorButton is "Close Door" or "Open Door" dependent on the value of the

[3] Extract from Figure 1 of [BY90].

DoNaLD variable door/open. The interface between the Scout and DoNaLD notations is a bridging definition[4]:

```
integer DoorIsOpen = DONALD boolean door/open;
```

which, when translated into EDEN, becomes:

```
DoorIsOpen is _door_open;
```

In general, because the implementation of the Scout data types may be different from the implementation of the corresponding DoNaLD data types, type conversion is needed when the bridging definitions are translated into EDEN definitions. So the translated bridging definitions normally take the form:

Scout-variable-name is *convertor(DoNaLD-variable-name)*;

Similarly, bridging definitions can be added to DoNaLD and other notations so that a complicated communication network can be achieved. Bridging definitions are in concept no different from other definitions, and the general rule of non-circularity still applies to them. Care should be taken when writing bridging definitions so as not to violate the rule.

```
integer DoorIsOpen = DONALD boolean door/open;        # a bridging definition
point miscMenuRef = {250, 400};
point doorButtonPos = miscMenuRef + {strlen(plugMenu).c/2, 1.r};
window doorButton = {
        frame:  ([doorButtonPos, 1, strlen(doorMenu)]),
        string:  doorMenu,
        border:  1
};
string doorMenu = if DoorIsOpen then "Close Door" else "Open Door" endif;
```

Listing 6.1: Program Fragment of the Scout Notation

[4] Not implemented yet. In the actual listing, the EDEN definition "DoorIsOpen is _door_open;" is written instead.

6.4.2. Naming Scheme

For historical reasons, each definitive notation is translated into EDEN according to its own naming scheme. For instance, neither Scout nor ARCA changes the variable names during translation, which means that a Scout integer i, an ARCA integer i and an EDEN integer i will all be cast into the same EDEN variable. Name collision may happen to the variable names, function names and action names of the definitive notation implementation libraries. Without precise guidelines, function names such as *int_mult* (integer multiplication) may be unconsciously chosen for the implementation of two similar but distinct functions of two definitive notations. Therefore, some precautionary steps or guidelines on the naming are necessary to supplement the implementation steps listed in Section 6.2.

A possible suggestion is that every variable name begins with characters identifying which definitive notation it belongs to. For example SC_i, AR_i and DO_i may be the translated names of the variable i in Scout, ARCA and DoNaLD respectively.

6.4.3. Switching Between Notations

In order to incorporate several definitive notations into a single system, the Scout system uses a method similar to the way the preprocessors of `roff` (the standard UNIX text formatting language) works. A block of definitions of a definitive notation is preceded by a declaration of the notation name. For example:

```
%scout
...
```
Scout definitions
```
...
%donald
...
```
DoNaLD definitions
```
...
```

The individual translator will translate only the lines from the notation declaration downwards until the declaration of another definitive notation is reached; the remaining lines are untouched. In this method, all the translators are running independent of one another. This provides flexibility for future extension to the system.

6.4.4. Other Considerations

Comments are useful information for maintaining programs. When there is only one notation within a system, how comments are inserted is not important. But when a system incorporates several notations, it is logical to use only one format of comment for the whole system rather than have different formats for different notations. For the same reason, there should be a consensus on the way definitions are separated from one another. Ideally, these considerations should influence the design of definitive notations.

There is a historical problem in that the design and implementation of our definitive notations preceded the integration plan. There has been no consensus on the format of comments and definition separator. A DoNaLD or Scout comment starts with a # sign till the end of the line; EDEN comments are enclosed by a /* ... */ pair; EDEN and Scout terminate a definition with a semi-colon but there is none for DoNaLD or ARCA. Since the # sign is a postfix operator in EDEN, it may be confusing if the # sign is set as an indicator for the start of comment in the unified system. This shows that in choosing the format of comments we should consider whether the comment sign is likely to bear any other meanings in other, may be even future, definitive notations. Since EDEN is the core notation in our system, it is most reasonable to preserve its

style. Another good reason for adapting this convention is that the EDEN comment sign consists of a combination of two symbols, rather than a single symbol that is likely to be used as an operator. Concerning the definition separator, it is usually easier to parse a definitive notation with a definition separator, and EDEN has one. Therefore, we recommend that the /* ... */ pair and ';' should be the standard comment sign and definition separator for all definitive notations.

6.5. Evaluation of the Scout Project

This section describes the current status of the Scout System, and summarises our experience of its design, implementation and application.

6.5.1. Modelling, Understandability and Usability

ARCA, DoNaLD, CADNO and EDEN were originally designed and implemented by different people to run in different environments. With the development of the Scout notation, it becomes possible to bring several definitive systems together in the Scout environment. Currently, the Scout system can incorporate ARCA, DoNaLD, EDEN and Scout notations. More definitive notations can be integrated into the Scout system as time goes by.

The ability to integrate definitive notations allows the breaking down of a large programming task into sub-tasks. Each task may be effectively developed under a suitable definitive notation. Scripts of definitions can simply be put together to form a required program. The only slight modification may be the addition of bridging definitions.

Special-purpose notations can improve software development productivity [BCMZ88, Ramsay87]. They allow the programmers to work in a higher level language closer to the problem domain. The programs created are easier to understand and debug. In exploratory software design especially, shortening the understanding stage is absolutely crucial.

One drawback of using a higher level, domain-specific notation is that some efficiency is lost in execution. Although we may be prepared to sacrifice a certain amount of run-time speed for the sake of quicker development, greater robustness and reliability, run-time efficiency is also a major concern in exploratory program development. This might be used as an argument against developing definitive notations instead of perhaps developing programs in EDEN directly. However, it can be argued that the time efficiency is an insignificant loss. The evaluation of a definitive notation consumes time in two areas: the translation process and the evaluation of EDEN definitions. Since the source notation and the target language (EDEN) are both definitive, the translation time is short. Also, because the principal difference between the original script and the translated one is the change of underlying algebra, the translated script should have the same order of efficiency as if the problem is directly solved in EDEN.

The designers of multi-paradigm programming languages such as LOOPS and NIAL need to consider the pros and cons of practising mixed programming styles [Bobrow85, Smillie88, KKW87, MGJ84]. Similarly, we have to justify the integration of several notations in relation to the issue of usability: Is it realistic to learn so many notations and then use them? Part of our justification is a familiar argument that is advanced for mixing programming styles: improved expressiveness can be bought at the cost of introducing diversity of notation. Our approach has some advantage over mixing programming styles since all the notations in our integrated system are based on the same definitive programming framework. Learning a new definitive notation is like learning new vocabulary whilst learning a new style of programming is like learning new grammatical rules. Therefore, the learning time for definitive notations should be reasonably short. Moreover, the definitive notations are application-oriented. The new vocabulary to be learnt should be familiar jargon that is easily picked up by the potential users. For this reason, it should be easy for someone familiar with the intended application to learn the underlying algebra of any particular

definitive notation. It is, however, reasonable to expect the subsidiary parts of the notations, such as comments and definitions separator, to conform to the same convention.

One aim of developing definitive systems is to assist exploratory software design by applying definitive principles to modelling the state of the real world (i.e. using definitions to capture the relationship between objects). Through observations and experiments by redefinitions, the problem under consideration can be more easily investigated. By integrating definitive notations, different aspects of a problem can be modelled in the most appropriate notations and the scripts can be efficiently combined. Hence, the effectiveness of definitive state modelling is maximized.

6.5.2. Support for Iterative Design

The visualisation problem described in §6.1 has been programmed using the Scout system. The script and its output are shown in Appendix E. Note that the organisation of the script reflects the manner of its development; it is not well organised. You can find, for examples, a fragment of DoNaLD definitions in the midst of some ARCA definitions; the declarations of the variables are not grouped together; definitions of the same notation are not grouped together. The haphazard form of the script emphasises the suitability of the definitive system for interactive use in program development. Since only the last definition of each variable is significant, the ordering of the definitions is irrelevant and a programmer need not write the script in a particular order. One simple example found in the visualisation example is that the scaling factor, *m*, in the specification of poset was 50 at one stage. After observing the temporary output on the screen, it was set to 100 to get a better picture.

6.5.3. User interface

When switching from one definitive notation to another, the name of the new input notation has to be declared (see §6.4.3.). Whilst developing or experimenting with a

definitive script, switching between definitive notations is common. Experience tells us that it is very easy to forget to switch to the new input notation before entering a definition. Although the system already provides different prompts for different definitive notations to remind the user which definitive notation it is expecting, the user tends to ignore the prompts.

A better solution would be to provide a multi-window user interface[5]. This interface would provide one window for each definitive notation, one for system messages and one for the script of definitions. There would then be no need to type in notation declarations (such as %donald) by hand. Instead, the user would place the mouse in the appropriate window before entering a definition. The interface would determine whether a switch between notations was needed, and if necessary make the switch automatically.

It is desirable to have a window for the overall script and separate windows for individual definitive notations. On the one hand, it is important to keep a log of what the user has done and the system's response. On the other hand, since different definitive notations are specialised in and responsible for modelling parts of a problem, it is desirable to group definitions in the same definitive notation together, so that the analysis of each aspect of the problem may be done without the distraction from interpolated definitions of other definitive notations. In the visualisation example, the definitions for the line arrangement in DoNaLD, for the S_4 graph in ARCA and for the screen layout in Scout address different concerns and can be better understood if they appear in different sections of the program.

---

[5] A previous attempt was made to improve the user interface of the DoNaLD system [Iu89]. However, that attempt focused on improving the programming environment for the DoNaLD notation rather than on the integration of definitive notations.