3

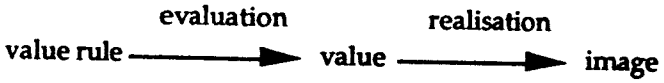
Software Tools Using Dependency

Definition-based (definitive) programming sounds like a new subject, but many software tools, even some that are very popular, use concepts similar to definitions. These software tools include spreadsheets, some document processing software, some graphics editors and the make utility. Such tools can be seen to represent good areas of application for the Definitive State-Transition (DST) model. Their success can also encourage us to pursue definitive programming. This chapter examines how they use dependency information and compares their approach with our use of definitive principles.

3.1. Spreadsheets

3.1.1. Basic Concepts of a Spreadsheet

The functionality of the spreadsheet has increased tremendously since the first spreadsheet program VisiCalc in the 1970's. It can now be used as a database. It can also produce colourful graphs and so is becoming a presentation tool. However, the basic concepts of spreadsheets have not changed. A spreadsheet is basically a collection of cells located on a rectangular grid. Each cell may be referenced by its position on the grid. Each cell may store a formula returning a real value or a string of characters. This formula may or may not contain references to other cells. If it does, then when those cells referenced change value, the formula will be recalculated automatically. The value obtained will then be displayed according to a format rule (often chosen by menu selection). For example in a financial setting, a value often represents the currency and so is appropriately displayed as a number with two decimal places and preceded by a pound sign. In short, the image that appears in a cell has gone through the following process:



In many ways a spreadsheet program is similar to a script of definitions:

- Each definitive variable is analogous to a cell in a spreadsheet.
- A definitive variable may be defined in terms of explicit values or by a formula (value rule) in terms of other definitive variables.
- One of the definitive notations, Scout, is intended to perform a similar role to the spreadsheet format rules (see Chapter 4 and 5).
- Both definitions and spreadsheet programs define uni-directional data dependency.

Although a spreadsheet is very similar to a definitive script, we can see the essential differences between spreadsheet programming and definitive programming in the following three aspects: the variable names, the range functions and operations, and the order of evaluation.

3.1.1.1. Variable names

Superficially, the variable naming system in a spreadsheet is very simple – the name of the cell is the position of the cell on the spreadsheet. The convention is that the name of a column is a group of letters, A-Z, AA-AZ, BA-BZ etc. Column A means the first column, column B is the second, column AA is the twenty-seventh, column AB is the twenty-eighth and so on. The name of a row is an integer starting 1 or 0 depending on the spreadsheet. On deeper analysis, the variable naming system is much more complex.

We can insert a row of cells at any position of a spreadsheet. By doing so, other cells at or below the insertion point will be moved down one row. This means that the cells at or below the insertion point will obtain new names. Similar situations are the insertion of a column, deletion of a row or a column. Therefore, the first observation is that the name of a cell in a spreadsheet may change over time.

An implication of changing a variable's name (cell name) is that the formulae in other cells may require corresponding changes. Suppose that a series of cells C2 to C5 (denoted by C2..C5) are intended to show the multiples of C1. The definitions are:

Cell	Formula
C1	3
C2	C1 + C1
C3	C2 + C1
C4	C3 + C1
C5	C4 + C1

The insertion of a row should, and in a spreadsheet typically does, redefine the definitions of the cells to the following:

Cell	Formula
C2	3
C3	C2 + C2
C4	C3 + C2
C5	C4 + C2
C6	C5 + C2

The change of formula is, however, not always desirable. For instance, if one moves cells C2..C5 to D1..D4, one may like to leave references to C1 untouched. That is:

Cell	Formula
C1	3
D1	C1 + C1
D2	D1 + C1
D3	D2 + C1
D4	D3 + C1

This can be achieved by other conventions of the naming scheme. If a $ sign is put before the coordinate of the cell, that coordinate will not be subject to change. For example:

Formula of cell C2 before move	Formula of cell D1 after move
C1 + C1	C1 + C1
$C1 + $C1	$C0 + $C0[1]
C$1 + C$1	D$1 + D$1
C1 + C1	D0 + D0

[1] If the row number starts from 1, the formula will be "=$C1 + $C1" instead.

The flexibility in the referencing system in a spreadsheet is an advantage of the tabular arrangement of cells. The new reference of a cell can be calculated simply by adding the offset of the cell displacement to the original reference.

3.1.1.2. Range Functions and Operations

The spreadsheet also takes advantage of the regularity of the variable names in providing a rich set of range functions and commands. A range in a spreadsheet is a collection of cells enclosed by a rectangle defined by the upper left and lower right cells in the region. Functions such as @sum (summation of all the cells in the range), @stddev (standard deviation) and commands such as fill a range with incremental values are usefully defined for ranges.

The benefit of having ranges is that a variable number of cells may be addressed together. If a cell is inserted in a range whose summation is performed elsewhere, there is no need to alter the formula for the summation. The range of the summation is automatically extended by the re-adjustment process of reference names described earlier. Without range functions, a summation would require an additional term in the expression.

3.1.1.3. Order of Evaluation

Is the order of evaluation important in a spreadsheet? Order of evaluation is not important so long as the formulae contain no circular referencing. Unfortunately, a spreadsheet normally allows circular referencing. As a result, spreadsheets need commands to specify the recalculation order (row order or column order) and the stopping condition. The stopping condition is either when the evaluation result stabilises or the user-specified maximum number of iterations is reached.

3.1.1.4. Differences Between a Spreadsheet And a Definitive Script

The first difference between a spreadsheet and a definitive script is in the interpretation of formulae. The fact that spreadsheets allow circular referencing indicates that spreadsheet programming tends to interpret the value rules as methods of maintaining the values of the cells rather than stress the 'real-world semantics' of the cells. In effect, a value rule in a spreadsheet may serve as a procedural computational device. Definitive programming in contrast tries to maintain that formulae are statements about the relationship between definitive variables.

The second difference is that a definitive script describes the relationships between variables whilst a spreadsheet basically describes relationships between the values on particular locations or relative locations on the spreadsheet. Depending on the way the 'variable names' within a formula are defined, the formula relates the current cell with cells on particular locations or relative locations of the spreadsheet, not fixed cells. This could be a reason for classifying the spreadsheet as a visual programming language [Myers89]. On the other hand, a definitive variable is similar to a conventional variable in that the variable name always refers to the same entity. Because of the geometric referencing characteristic of spreadsheets, spreadsheet programming can take advantage of the tabular arrangement of cells. Range functions can be easily implemented in a spreadsheet but are not that trivial in definitive programming. However, a definitive variable is usually named after what it is meant to represent in the real world. A variable name has a message to tell which is often more important than the location of its visual representation.

3.1.2. Appraisal of Spreadsheet Programming and Definitive Programming

Whilst many people today are still treating the spreadsheet as a user-friendly interface, some like Kay and Kokol do treat the spreadsheet as a programming paradigm. Not

only that, they consider that the spreadsheet is in fact an ultra-high level language [Kay84, Kokol88]. Some like Hewett and Green also suggest that many features of electronic spreadsheet are helpful in rapid prototyping and notation design [Hewett89, Green89]. We find that many of their arguments are also applicable to definitive programming.

3.1.2.1. What-If

The usage of spreadsheets has been expanded over the years. A frequent use of spreadsheets is in modelling and simulation, which has been applied to engineering, chemistry, neural network, ecology, physics and psychology [Hewett89]. The *what-if* feature of the spreadsheet naturally makes it a tool for forecasting when some condition will pertain in the future. Typical examples are financial modelling and sensitivity analysis which takes advantage of this 'what-if' ability [Jackson88].

In order to adapt to the application, it is sometimes necessary to extend the underlying algebra of a spreadsheet. DYNAGRAPH is a spreadsheet-based interactive simulation modelling system. There are functions in DYNAGRAPH, such as table look-up functions, forecasting functions and delay functions, that are particularly designed for modelling and simulation of multi-period planning. "It would be wearisome, if not impossible, to use the popular spreadsheets *for the job*" [Anonum88].

Since definitive programming also maintains that a variable will be updated whenever one or more of the variables on which it depends is updated, *what-if* is also a prominent feature in definitive programming. Modelling and simulation is also a major application area of definitive programming. In fact, many of our papers discussed with examples the application of definitive programming in modelling and simulation [BBY92, BY92, BSY90, BRY90, BNS88].

Definitive programming faces a similar issue to the spreadsheet – the need to extend the underlying algebra for particular kinds of modelling and simulation. From the beginning, definitive programming has involved special-purpose *definitive notations* [Beynon85]. These definitive notations have underlying algebras specially designed for certain applications.

3.1.2.2. Rapid Prototyping

Hewett [Hewett89] suggested some considerations for developing rapid prototyping environments. The environment provided by a spreadsheet addresses these considerations:

(A1) Eliminate or reduce the need for both developer and user to attend to I/O details during prototype developments.

(A2) Allow the developer flexibility in creating alternative user views of the prototype.

(A3) Make it easy for the developer to change underlying relationships and parameter values, and to introduce simplifying assumptions.

(A4) Require limited programming from the developer during the process of prototype development.

(A5) Make possible easy replication of differing versions of the interface for comparative examination and testing by both developer and user.

(A6) Allow the developer to support the user's intuitive understanding of the task though direct representation of significant features of the task environment in which the system will be used.

(A7) Allow for the development of separate interface modules and layers, and links among those functional units.

(A8) Offer the pedagogical value of being accessible to, learnable by and modifiable by others, including users, under some circumstance.

Hewett also points out some limitations of spreadsheet regarding rapid prototyping:

(L1) a limited number of cells,

(L2) lack of support for building up and modifying new interface primitives,

(L3) lack of support for the development of a set of higher level design abstractions,

(L4) little or no provision for tracking the history of the design process,

(L5) lack of debugging facilities,

(L6) no means to constrain the end-users' interaction.

Definitive programming has much in common with spreadsheets concerning the advantages listed above. Empirical evidence from the use of our software prototypes by the undergraduate project students indicates that, apart from advantage (A5), which is unique to the spreadsheet and derives from its convenient copying and moving facilities, the advantages of spreadsheets cited by Hewett are shared by definitive programming. Since definitive notations provide a richer set of data types, operators and visual representations, some advantages such as flexibility in creating alternative user views and pedagogical value are further enhanced by definitive programming.

At the same time, definitive programming eliminates or relieves most of the limitations of spreadsheets. With respect to (L1), instead of a fixed size table of cells, definitive programming allows unlimited number of variables. With respect to (L2), our current definitive system can incorporate the definitive notations Scout and DoNaLD which can be used as tools for developing graphical user interfaces. With respect to (L3) and (L6), definitive programming is not confined to writing a set of

definitions; the ADM is an example of a definitive programming language which has higher-level control over sets of definitions. Not much improvement to (L4) and (L5) though except providing logging facility in our system.

To summarise, definitive programming retains most of the advantages and eliminates or relieves most of the limitations of spreadsheets. Therefore, according to Hewett's argument, definitive programming is a competitive tool for rapid prototyping.

3.1.2.3. Cognitive Dimensions

Green ([Green89]) generalises Streitz's observation of 'writing is rewriting' ([Streitz88]) to 'design is redesign' and 'programming is reprogramming'. Since a programming exercise involves frequent re-evaluation and modification, the amount of information that can be extracted from the notation with respect to re-evaluation and modification becomes important. Some cognitive dimensions for programming notation design are discussed in [Green89]:

1. *Hidden/Explicit dependencies* – how easy is it to cross-reference related information?

2. *Viscosity* – how easy is it to make localised changes? For example, how easy is it to insert a new formula into a spreadsheet cell?

3. *Premature Commitment* – how easy is it to develop a program in a mental generative order?

4. *Role-expressiveness* – how easy is it to infer the roles of different parts of a program from the program fragments themselves?

5. *Hard Mental Operations* – how easy is it to understand the individual programming constructs? For example, are there constructs such as the eval and quote functions in Lisp and pointers in C that are particularly hard to understand?

Green observes that a typical object-oriented programming system, Smalltalk-80, does not score very well under the testing of the above dimensions. It seems, on my evaluation, that definitive programming may get a higher score.

Hidden/Explicit Dependencies – In a definitive script, dependencies can be extracted from the definitions easily. Take EDEN as an example. Although the EDEN environment does not disclose long range dependency, the '?'-command (query command) does provide local information about both forward and inverse dependency.

Viscosity – Green observed that inserting a new formula into a spreadsheet cell is very simple but actions that entail rearranging the layout are quite another matter (for instance, introducing a new row may have the side-effect of corrupting the value rules associated with other rows). A definitive notation is similar to a spreadsheet in that assigning a new formula to a definitive variable is straightforward. Because definitive variables are not subject to the same geometric conventions and constraints as spreadsheet cells, new variables can be introduced very simply. At the same time, it should be noted that there is no operation on a definitive script that corresponds to introducing a row into a spreadsheet. Also, as explained in §3.1.1.2., the geometric conventions of a spreadsheet offer greater expressive power.

Premature Commitment – The order of definitions appearing in a definitive script is irrelevant, and so is the mental order of program design. Variables can be redefined at any stage, and in any order. In chapter 5, we show a top-down design strategy of a screen layout. On the other hand, we can, for example, incrementally add on new meters on the panel of the vehicle cruise control simulation in chapter 7 – this reflects a bottom-up approach of program design.

Role-expressiveness – In one respect, definitive notations show high role-expressiveness. Definitive notations have application-specific underlying algebras. The role of the definitions is implanted in the design of the underlying algebra. In another respect, definitive notations are low in role-expressiveness. Since the order of definitions is unimportant, a definitive script can be difficult to understand. For instance, the definitions for the lamp on a table can be far away from the definitions for the table itself. Automatic reordering of definitions can only help to a limited extent. Seemingly the most natural way of sorting the definitions is sorting by dependency. However, this may not be the best way depending on the occasion. In one context, one may like to group all the definitions concerning screen layout together but in another to group the definitions about the visualisation of a variable together no matter how many different definitive notations are involved. Nevertheless, if a definitive script is properly organised, because of the application-specific nature of definitions, definitive notations should attain a very high role-expressiveness.

Hard Mental Operations – Definitive programming, at its present stage, still shares the simplicity a spreadsheet enjoys. The particular examples of hard mental operations cited by Green, such as pointers and indirection in C, eval and quote functions in Lisp, are absent. Whether there are any hard mental operations requires further research by the cognitive psychologists.

In summary, the virtues of spreadsheets with respect to the cognitive dimensions suggested by Green are retained by definitive notations whilst some of the disadvantages of spreadsheets have been overcome.

3.2. Document Preparation Software

The most prominent use of dependency in a document preparation system is in the definition of styles. In a style-based document preparation system, style information

can be associated with individual characters, paragraphs, sections or the whole document [JB88]. Changes in the definitions of the styles will affect the presentation of the text; for a style-based WYSIWYG (What You See Is What You Get) editor, these changes will be reflected interactively on the screen. A typical style-based WYSIWYG editor is Microsoft Word. In Microsoft Word, a new style can be defined upon an existing style. Using this thesis as an example (as it is prepared using Microsoft Word), the first paragraphs following the headings are in Normal style; the subsequent paragraphs are in NL style. Normal and NL are defined as:

Normal :- Font: Times 12 Point, Justified, Line Spacing: 24pt, Space Before 12pt

NL :- Normal + Indent: First 0.5in

If the line spacing of the Normal style is redefined to 12pt (single line spacing, say for printing the first draft), the paragraphs with the NL style also become single line spaced.

Lilac [Brooks91] is another style-based editor. Lilac is both WYSIWYG and language-based. Because it is WYSIWYG, a change of style takes immediate effect on the screen, which is a close approximation to the printed output; because it is language-based, a user have more flexibility in defining styles. By providing similar programmability as in Troff and Tex [Knuth84], Lilac allows the user to define complicated styles such as might be used in a periodic table. Lilac has data types and operations specific to document preparation. Its data types are Box, Hglue, Vglue (horizontal and vertical glue between boxes), Hlist, Vlist (horizontal and vertical lists of objects), Num (number), Bool (boolean), Family (font family), Face (typeface) and Font. There are basic operations defined on these data types and user-definable operations can be defined on top of these basic operations. A document is generated by applying the operations (styles) to the text of the document.

It is clear from the form of Microsoft Word style definitions that style definitions can be regarded as definitions in the definitive programming sense. A redefinition of the base style has indivisible effects on all the styles directly or indirectly

defined upon it. Lilac goes further to describe an underlying algebra for styles. This further justifies the claim that many style-based document preparation systems are, theoretically speaking, definitive notations for document preparation.

3.3. Graphics Editors and User Interface Tools

Conventional graphics editors, such as MacDraw, do not make use of the dependency between objects. In graphics tools at the research level, the need for dependency is more commonly recognised. L.E.G.O. is a construction-based drawing language in which an object can be constructed with reference to other existing objects [FP89, FP88, FP86]. Since L.E.G.O. is an imperative language, the use of dependency information between objects has not fully exploited. There are however other graphics editors like Ded [Jeet87], GIPS [CFV88] and NoPumpG [Lewis87] which use dependency in a more declarative manner. The re-construction of an object will update the position or the shape of those objects dependent on it. Some graphical user interface (GUI) tools like ThingLab [BD86], Coral [SM88], RENDEZVOUS [Hill92] and Views [Pemberton92] also use constraints to establish links between graphical objects and between graphical objects and application-generated data.

The basic task of a graphics editor is to enable the user to manipulate and visualise the abstract model of a graphical image. [Beynon85], [Beynon88a] and [Beynon89] argue that *definition* is a suitable abstraction for the task. In fact, both Ded and NoPumpG use uni-directional relationships (*definitions* in our terms) for constructing the abstract model. The kinds of relationships within these graphics systems by-and-large concern the geometry of individual graphical objects. In addition to supporting geometrical relationships, NoPumpG incorporates a system clock that can be used to describe the relationship between graphical objects and time. For this reason, NoPumpG is better known as a tool for animation than as a graphics editor.

Graphical user interface tools resemble graphics editors in that they are both concerned with visualisation and manipulation of data. While the data involved in a

graphics editor is the abstract model of the graphical objects themselves, the data to be visualised and manipulated in a GUI comes from a separate application. Therefore, the relationships between visual object and the application data have also to be addressed in GUI. [BY90] clearly identifies that the visualisation process has characteristics similar to a mechanical linkage. In a mechanical linkage, a change in position in the input end immediately changes the position of the output end; the change of a view of an abstract model should always be synchronised with the change of the abstract model itself. Many GUI tools, such as those mentioned above, use constraints to link the abstract model and its views. The reason of using constraints is not only because there is a close relationship between abstract model and view but also because the interpretation of input is closely related to both abstract model and view. The multi-directional relationship described by a constraint enables a change of a view to effect a change in the abstract model. In our paradigm, we recognise the close relationship between model, view and control but reject models in which one can hurt other person by hitting his shadow. In our method, definition is the link between model and view. The uni-directional nature of definition perfectly describes the relationship between model and view. An input is interpreted in the context of the view but will directly affect the model. The view is updated indivisibly with the change of model. More detailed discussion on our way of handling input will appear in Chapter 7. In short, the GUI paradigm we are employing can be depicted as follow:

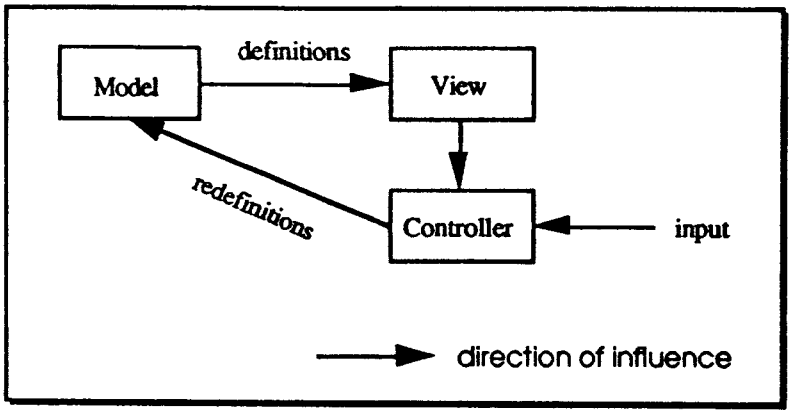


Figure 3.1: Graphical User Interface Mechanism in Our Systems

The view is linked to the model via a set of definitions. The controller interprets the input based on the information obtained from the view definitions and manipulates the model by redefinitions.

3.4. The Make Utility

The last kind of the software to be discussed which make uses of dependency is the make utility. Make is originally a standard UNIX utility for file management, and is now widely used in the PC community. In make, the dependency between files can be specified together with commands for updating the files. A typical use of make is for compilation of programs. The following is a simple example of a makefile:

```
1. calc: main.o function.o
2.         cc -o calc main.o function.o

3. main.o: main.c
4.         cc -c main.c

5. function.o: function.c function.h
6.         cc -c function.c
```

Listing 3.1: An Example of Makefile

This makefile is intended for the compilation of a calculator program. The program is written in two modules, main.c and function.c. Function.c includes a header file function.h. Since the C compiler supports separate compilation, two more files, main.o and function.o, also need to be kept up-to-date. While lines 1, 3 and 5 in the example specify the dependency between the files, lines 2, 4 and 6 specify the way in which the files are to be updated. In UNIX, there is a set of attributes associated with every file. These include the last access time and the last modification time. When make is invoked, it checks the status of the files on the dependency lists. If any file on the dependency list has its last access time equal to its last modification time (which means that that file has recently been updated), the target file has to be updated according to the specified rule. At the beginning of a make session, make generates a dependency tree

and checks the status of the files starting with the leaves of the tree. In this way, duplicate updating of the same file can be prevented. This mechanism is similar to the implementation of our system except that our system is continuously executable. What corresponds to the dependency tree in the case of definitive variables is constantly maintained rather than being regenerated in every evaluation.

Unlike definitions in which data dependency can be inferred from the definitions themselves, the file dependency has to be stated explicitly in what is called a makefile. This is because the commands for file maintenance do not generally disclose which are the source files and which are the target files. The command in line 6, for instance, gives no indication that function.h is one of the depending source and function.o is the target file. This dependency is implicit in the content of function.c and the conventions of the C compiler. However, different commands have different conventions. There is no general rule for inferring the file dependency from a command.

3.5. Theoretical Framework

We have examined some commonly used software tools in this chapter. In these tools, dependency plays a significant role. Perhaps it is their use of dependency that makes them so popular. It is however worth noting that, despite having different histories of origin, these tools express dependency in very similar ways. As we have mentioned in the discussion, the underlying principle behind such software is not dissimilar to that of definitive programming. It is our belief that dependency is not only beneficial in individual applications but is applicable to broader issues of software development. Our research in definitive programming is an effort to develop a theoretical framework of programming that is based on dependency.