7 Problematic issues and possible solutions

In this chapter, some of the problematic issues and limitations of MWDS that have been encountered in the study and development of definitive models are reviewed, and some possible solutions are suggested.

7.1 Problematic Issues in MWDS

Many problems have emerged from studying and interacting with various definitive models. This is because the computer tools for supporting MWDS are still under-developed and limited. The designer may have to understand the limitations and nature of the tools before constructing a model. Constructing the model involves representation of a referent, and if the designer cannot conceptualise the situation well enough this may result in an inefficient use of scripts and make the model very complicated. Many features associated with MWDS, such as freezing a definition and triggered actions, can cause difficulties in comprehending and debugging the model. The entire discussion can be divided into three sections respectively devoted to construction, maintenance and evaluation efficiency concerns.

Section 7.1.1, 7.1.2 and 7.1.3 deal with the problems of construction, maintenance and efficiency in the evaluation of definitive scripts. A particular concern is the difficulty that traditional programmers find in MWDS, which stem from the novelty of the principles used.

7.1.1 Problems with constructing a model with definitive scripts

Although the concept of open-development dependency has been widely used in computer science (cf. Chapter 1), programmers, typically coding in a procedural paradigm, do not fully understand the principal concept of using definitive scripts to support model building. This may be partly because the characteristics of definitive principles and scripts are different from those of traditional programming. MWDS, principally based on the concept of dependency, offers a radically different style of modelling (cf. Chapter 2). Its emphasis is on using a script to represent state as observed and experienced (cf. Chapter 5). The use of a definitive script can mediate between a computer representation and human cognition. The philosophy behind definitive scripts is consistent with Ghezzi and Jazayeri's view of the role of a programming language:

> "programming languages are regarded as the tools for communication not only with computers but also with people" [GJ98],

in that a definitive script is intended for interpretation by both the computer and the human. MWDS takes Ghezzi and Jazayeri's idea further than they envisaged; not simply developing a means to make a program intelligible to the human reader, but supplying an artefact that can be used interactively to complement inspection of scripts as an aid to comprehension. It promotes a new vision for computer science in which MWDS has a central place.

Building a software application with most traditional programming languages, (e.g. Pascal, C and C++) is typically associated with the characteristic activities identified in the waterfall model of the software life cycle: requirements analysis and specification, software design, implementation, verification and validation and maintenance. The idea that software development is a sequential combination of phases is far too simplistic to do justice to practice, but the problems of bringing coherence and integrity to the abstract and practical products of these various characteristic activities are real enough. Programming languages are used in implementation, when the algorithms and data structures for the modules that form the entire application are defined and coded. Traditional programming design typically involves the decomposition of the entire program into logical components. Each component can be a procedure, an object or an agent according to the underlying programming paradigm. In traditional software design, there is a clear separation between the abstract development process and the development product [Sun99].

MWDS, in contrast to traditional approaches, focuses on using a definitive script to represent an external referent according to its state as observed and experienced at any particular point in time. This script can be changed and refined to correspond to changes in the modeller's perspective.

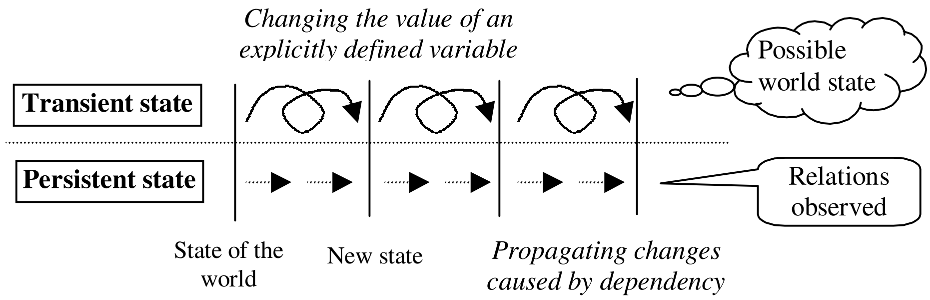

Figure 7-1: The state changing in definitive scripts

MWDS involves two kinds of state change:

- one involves changing the value of an explicitly defined variable; this typically relates to what is designated as transient state in Figure 7-1. The values of dependee variables in the new state will be updated according to the particular pattern of dependency framed by the persistent relationships associated with implicitly defined variables.
- the other involves changing the persistent relationships; this will change the pattern of dependency, and the dependee variables in the new state will be updated to correspond to this new pattern of dependency (cf. Figure 7-1).

The process of modelling with definitive scripts involves creating and communicating experience through a computer-based artefact [BCSW99]. It is an experimental test-bed for interactive activities in which there are no distinctive phases as in a software cycle. A modeller initially designs a model by representing what they perceive and observe in a particular situation and then gradually redefines those observables or introduces new observables (cf. Chapters 3 and 5). In the early stages, when the objectives of the model-building may be unclear and the representations for basic observables in the referent have be devised, the model development is typically slow and potentially frustrating. Once the model development is well-established, the experiential character of the modelling activities becomes more prominent. At this stage, the recognition of meaningful relationships relies on the subtlety of the mechanisms that create associations between one experience and another and this can itself be stimulated by interaction with and modification of the definitive model. This exploits what Turner [Tur96] characterises as a ‘blending’ of experiences: it can lead to rich and engaging representations but can also result in incongruous or funny associations. The model does not need to be well defined before actual coding, as is required in a traditional programming paradigm. The modeller may find it hard to initiate modelling from a MWDS perspective if he/she has been trained in and is familiar with a traditional programming paradigm.

Design and strategy issues in constructing a definitive model

The organisation of definitions is very crucial in MWDS since modelling activities in MWDS involve continuously modifying, redefining and interacting with the script. Keeping track of the revisions of the model is one of problematic issues in MWDS. Relevant issues include recording the inter-relationship between the sequences of definitions and redefinitions introduced in the modelling, recording different versions and their time of introduction and recording the order in which actions appear in the script.

The file structure

Designing the file structure to keep definitive scripts is an essential concern in MWDS. There are two possible ways of storing definitive scripts. One involves storing all the definitions in the model in a single file. The other involves distributing the definitive script into several files. Each of these two approaches has advantages and disadvantages.

The single file approach: We have no problem in understanding the script with this strategy if the script is small (cf. the Room Viewer model). But if a model involves a large number of definitive scripts (as in the Car History model {Car94}), the script is very hard to understand and to extract from for re-use. There is no modularity as there is in the several files approach. This makes the redefinition context difficult to identify so that it is hard to debug the script. The script in the file can also be grouped in different ways in order to give various views on its interpretation, as we shall discuss later with reference to the Car History model.

The several files approach: In developing a complex ADM artefact, which consists of many agents interacting through a definitive script, small portions of scripts are introduced into the model bit by bit (cf. the Heapsort model in Chapter 6). The introduction of each script can stem from changes in the modeller's experience during interacting with the model, changes in the external situation and changes in the objectives of the modelling. The focus in introducing a script is typically on its effects where redefinition and action are concerned. The global context for a redefinition may be hard to understand. There are interrelationships between files and we need a 'run file' to record the order in which files are introduced into the model. The several files approach can support a better tracking of the experimental strategies that are featured at each stage of development.

Versions and times in the introduction of scripts

An ADM artefact is open-ended and experientially-based in character. It can be incrementally developed, refined and extended over an unlimited period of time. In part because of the characteristics of the tkeden interface, it is common for the modeller to construct the model interactively by editing and (re)loading files. Recording the time when each file is introduced and modified is very essential. This is because it helps the modeller to organise the order of his/her experimentation with the model. By way of illustration, Figure 7-2 indicates the problems associated with ordering files. In Figure 7-2, File B is redefined (say to fix some definitions) at t_6. We then have to decide whether we should replace File B with File B' or include File B' after File E. The problem with the first option is that some definitions in File B' may refer to definitions in File C, D or E. With the latter option, we must ensure that File B' does not contain definitions of variables that have subsequently been overwritten in File C, D or E.

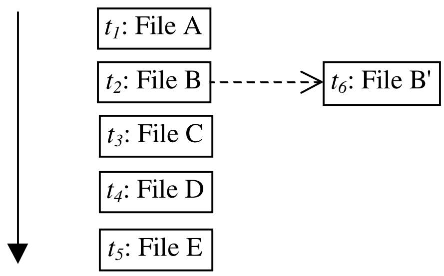

Figure 7-2: The file structures and their timestamps

Another problem relating to the order in which files are introduced stems from the fact that the ordering of actions can be significant. For instance, with the current implementation of the tools supporting MWDS, we may encounter problems with the ordering of actions as illustrated by the following alternative scripts:

/* initial values A=10; B=2; */

Script 1: proc updateA: T {A=B+5;}; proc updateB: T {B=A-5};	**Script 2**: proc updateB: T {B=A-5}; proc updateA: T {A=B+5;};

/* trigger the actions by redefining T */

/* including Script 2 results in A=7, B=2 */	/* including Script 1 results in A=2, B= -3 */

In Script 1, the value of variable A is updated first, then the value of variable B is evaluated using the updated value of A, whilst, in Script 2, the order of updating these two variables is reversed.

Introducing actions that interfere with each other in this way is in general a bad practice in MWDS. It is important that the modeller avoids constructing scripts that feature actions of this kind. Note that the problem illustrated in Scripts 1 and 2 arises from the presence of assignments to A and B; contrast the way in which the redefinitions in Figure 2-1(b) commute.

Freezing definition

Since maintaining dependency is very expensive, it is necessary to suspend or delay the updating dependency until all needed values are given. Freezing definition is a technique to cut off the dependency network in the model. It can be used to disconnect the dependency network once the task is done. For instance, in the Timetable model (cf. Chapter 4), when the visualisation of the model is properly set up, we can freeze the definitions:

StaffSelect_myList is stafflist;   StaffSelect_myCaption is stafflist;

by assigning their literal values to them, thus:

StaffSelect_myList = stafflist;   StaffSelect_myCaption = stafflist;

Thereafter, changing stafflist does not affect the visualisation of the model. However, these definitions can be restored at any time if the modeller needs to change the visualisation of the model to reflect a change to stafflist. In order to effectively use freezing definitions, the modeller needs to understand thoroughly its possible side-effects and to be careful to avoid breaking essential dependencies.

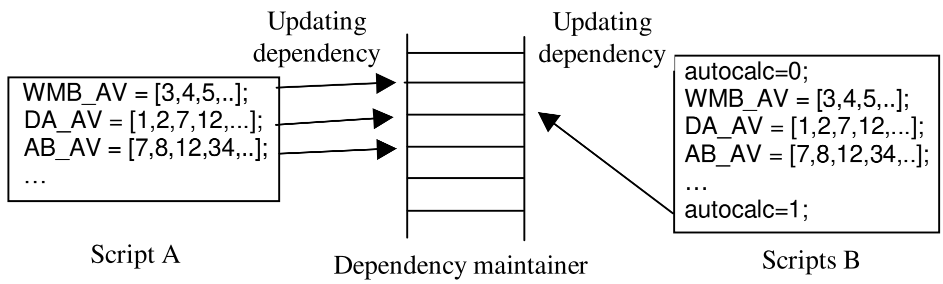

Figure 7-3: Delaying dependency maintenance

Delaying the update of dependency is another essential concern in MWDS. The dependencies are maintained when all required values are given. For instance, in the Timetable model when feeding a set of definitive variables into the model, we do not want to let the system update the dependency every time each new variable is entered. Instead, we only want the dependency to be updated after all variables are included. By way of illustration, Figure 7-3 demonstrates how a built-in variable autocalc (cf. Section 2.1.1) is used in the model to delay the updating of dependency. In Script B, the system will maintain the dependency when autocalc is set to 1 (or after all needed variables are given). In contrast, in Script A, the system will update the dependency every time each variable is read. However, we have to manually insert autocalc in the appropriate place. It is beyond the capacity of the system to determine automatically where to delay the updating dependency.

Eliminating definitions is sometimes essential in MWDS (e.g.) because the dependency in real-world state is not rigid. With the current tools, definitive variables are hard to eliminate once they are defined into the system. They cannot be automatically got rid of, but they can be manually eliminated by a built-in command forget("*string*"). Before we can use this command, we have to ensure that the variable has no dependees before deleting it. For instance, in order to delete B, in the context of the definition:

```
A is B+C;
```

we first have to eliminate the dependence of A on B by redefining A, for example by introducing the definition:

```
A = B+C;
```

or the definition

```
A is C;
```

and then using forget() to eliminate the variable B. This can only be conveniently done manually. This means that, in practice, when a complex model is developed, there are several unnecessary variables that are kept in the system but are assigned a dummy value. For instance, in the MBF model there are many DoNaLD point variables whose values are assigned to {0, 0} – their visualisations are hidden but they still exist in the system.

The concept of dependency and triggered action

Triggered actions appear in many definitive models as automatic agents invoked by the redefinition of their triggering variables (cf. Section 2.1.2). The concept of a triggered action is very rich. It combines the use of a procedural programming technique with dependency. The procedure is activated by its triggering variables instead of by a procedure call. By way of illustration, Figure 7-4 shows an action (viz. report) that is invoked when its triggering variables are redefined. Such triggering occurs in many contexts: where a triggering variable is redefined explicitly (e.g. first=4;); is implicitly changed by a function (e.g. exc(1, ixgtch1);); or is partially redefined (e.g. val[1]=3;).

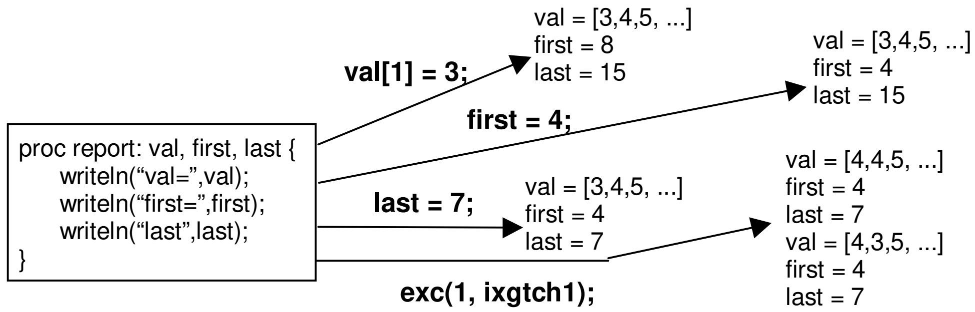

Figure 7-4: An example of a triggered action

As it is easy to imagine, if the script consists of many complex triggered actions, we may have unexpected side-effects when redefining variables. This can cause difficulty in debugging and understanding the script. The problem of determining the order in which triggered actions are executed has been discussed previously. There is also a possibility of cyclic triggering. The modeller should be aware of these problems when interactively developing a model.

In a large script that involves a lot of triggered actions and definitions, the triggered actions can sometimes cause redundant redefinitions and may significantly slow down the updating process. For instance, in the Heapsort model, the triggered action maintainheap1, defined as follows:

```
proc maintainheap1: hc1, next{
        if(!hc1 && next==1){
                exc(1,ixgtch1);
                next=0;
        }
};
```

has hc1, next as triggering variables. Redefining one of these two variables invokes the action. From the script, this action will be activated at least twice if hc1 is false. The first action is activated by the external change that affects the value of hc1 and the second one by the internal effect of the execution of exc(1, ixgtch1). This procedure swaps the values of two elements in the list and subsequently affects the value of hc1 so that the maintainheap1 action is triggered again (albeit with hc1 true).

7.1.2 Problems in the maintenance of large scripts

Since it involves personal experience and human perspectives (cf. Figures 2-11 and 2-17), MWDS arguably helps to facilitate the integration between human thinking and computer representation. Definitive models can be regarded as computer-based artefacts which embody the modeller's construal. Their interpretation is informal and their structure need not follow a specific pattern as that of a procedural program does. The script in the model can represent the modeller's understanding, perception and observation, which may be influenced by the external situation. Studying the script can resemble learning and comprehending the modeller's perception and the representation of the associated external situation. There are many problematic issues concerned with maintaining or debugging the script. These can be listed as follows:

- **Specifying the right variable**

It is characteristic of definitive scripts that they are very open-ended. Each definition can be changed on-the-fly, as can the defining formula in a spreadsheet cell. When we handle a large spreadsheet, it is very hard to grasp the inner mechanism of the sheet. The spreadsheet user may not know which cells it is appropriate to change in order to make sense of the inner mechanism. MWDS also has a similar problem. For instance, the Lines model (cf. {Lines91}, Section 3.2.1) was initially developed with a focus on visualising abstract mathematical concepts. The modeller did not concentrate much on designing the interface. The six key variables that trigger all the dependencies and, as a result, determine how the corresponding visualisation will be displayed, are not easy to identify without any hint or information from the modeller. This makes it hard for the unfamiliar user to make sense of the model. Although he/she can try to interactively investigate and interrogate the script to gain a grasp of the relationships in the script, it will be much quicker if he/she knows these key variables in advance, and can interact with the model progressively more deeply. To overcome this problem, most of the definitive models have README files accompanying them to guide the user and to record information about their development. Alternatively, provided the model is not too large, we can identify the key variables visually by drawing the dependency network diagram for the script (cf. Section 2.2.1 and Figure 2-2).

An analogy for a naïve user interacting with a definitive model is a child first encountering a new toy. Initially he/she hardly knows what the toy is for and how it works. He/she just interacts with it by trial and error. With the supporting tools for MWDS, the user can experimentally interact and explore the model in a similar way that the child tries to get amusement and understanding from the toy.

- **Understanding complex dependency**

Some people can express thoughts and relationships very precisely, some cannot. Definitive scripts can be used broadly to represent the modeller's perspectives on his/her model, which can be unstructured and can involve many complex interdependencies. To be able to understand a definitive script, we may need to memorise the observable names and understand their semantics and the dependencies amongst observables. This can be easy if the script is small, but it will be very hard if the script is large and has very complex interdependencies between observables. There are several ways in which the definitions in the script can be grouped to assist their interpretation. For instance, the definitions in the script of the Lines model can be partitioned according to their similarity or their interdependency (cf. Section 3.2.1), whilst the definitions in the script of the Heapsort model can be grouped according to their functionality (e.g. visualisation or internal mechanism or triggered actions) or according to the definitive notations to which they belong (e.g. Scout, DoNaLD).

When developing definitive models, it is best to construct them based on the three fundamental concepts of observation, agency and dependency (cf. Figure 2-17) so that they work efficiently and that their scripts are sufficiently well-structured to be understood. The Heapsort model and the Lines model are good examples of such models. If the modeller misuses definitive scripts, the model may be designed with unnecessary procedural-style features. In this case, it is quite hard to trace, debug and understand the script in the model. The General-User-Interface model and the Car History model are examples of such models.

Even though the (d)tkeden tools offer many features to help the modeller to interact with the model and inspect the script (cf. Section 2.1.2), it can still be difficult to grasp the relationships amongst variables inside a large script. The built-in command 'symboltable()' returns the symbol table in the form of an Eden list and provides the modeller with the comprehensive information about all the variables in the definitive model. The use of meaningful observable names that correspond to their referents can also be helpful in understanding a script.

- **Identifying the state of the model**

As discussed before, MWDS offers an open-ended model development. The modeller uses definitive variables to represent his/her observations. The model can be started with an unclear goal and then be incrementally redefined, modified and extended in diverse directions with no limitation. Without a record of the model development or some hints about its purpose, it is very difficult for the modeller or the next developer to identify the state in which the model has been left by the last phase of development.

Interacting with a definitive model is in some respects similar to reading a book. A reader may place a bookmark so that when he/she resumes reading, he/she knows where to open the book. However, sometimes he/she may need to revise what was read before. The new developer spends some time experimenting with the model in order to identify the state of model. In the worst scenario, the state of definitive model is such that the reader ends up reading the book from the beginning only to find that the book written by the early modeller is unfinished. MWDS supports the concept of open development. A model can never be complete: it is typically completed to the extent that it satisfies the designer's requirements at some point in time. In MWDS, good documentation can help the user to start engaging with the model, but in order to fully understand the model it is necessary to actually interactively experiment with it.

- **Difficulty when debugging**

In MWDS, when we want to redefine variables or bring new variables into the model, we need to make sure that this does not destroy the essential mechanism of the model. To ensure this, we have to understand the existing definitions by using some features provided by the (d)tkeden tools and investigate the effects of redefinitions. When dealing with a large script, redefining one variable can affect a great number of variables through side-effects and unexpected updating of variables may occur. This is similar to the problem of unexpected changes from triggered actions.

Since MWDS is, in principle, closely related to spreadsheets, it inherits their disadvantage of difficulty in debugging [Nar95]. The unexpected change caused by redefinition is an essential problem for debugging. In traditional programming, the program works sequentially in a specific order and no side-effects from dependency updates appear. In contrast, the interpretation of a definition in a script is not fixed on its introduction but changes whenever the script is modified. This is because redefining dependee variables causes a propagation of changes to maintain dependency in the model and the scope of this propagation is determined dynamically by the script.

- **Reusability**

Reusability is a big issue that has been studied for many years for various programming languages especially for object-oriented languages. The libraries in many programming languages such as C and Pascal can be seen as one form of reuse of a program. The inheritance feature of the object-oriented paradigm is another example. However all these traditional programming languages follow a top-down design technique. The entire domain process is always divided into small sub-problems. Several small procedures are combined and work together in a complete system.

MWDS has a different approach to analysing the problem domain. It attempts to understand and solve the problem at the same time. A script can represent the individual modeller's observation. Each observable has its own counterpart in the referent. It typically needs to exist all the time in the model, since its value can be indivisibly updated by the dependency system. In this respect definitive variables are quite different from the temporary parameters that are passed to a sub-routine in a procedural program. Scripts are rarely reused in the same sense that 'routine' procedures are reused in procedural programming languages. The analogy between definitive variables and spreadsheet cells is helpful in this context. The literal definition associated with the cell is rarely reused, but there may be other cells where a similar pattern of calculation is required. The copy and paste mechanism provided in a spreadsheet illustrates such an example of reusing an existing pattern of definition in a cell.

- **Reproducing similar set of definitions**

As discussed in Chapter 2, definitive scripts are used to represent observation and perception, in much the same way that the experimental scientist uses an artefact as a means for the metaphorical representation of observables. In this context, it is often useful to formulate definitions to represent similar kinds of observations. For instance, the points p1, p2, p3 in the Heapsort model are similar to each other, with only small differences of syntax. It is often necessary for the definitions in such a set to be repeatedly defined manually, even though it represents a generic abstraction perceived by the modeller. For instance, in the Heapsort model and the Lines model, there are generic families of definitions so that the models can be enlarged by generating definitions from the generic pattern for each family.

7.1.3 Evaluation efficiency

In small definitive models, the issue of efficiency is not highly significant, since the dependency maintenance system is capable of storing and updating all dependencies at a reasonable speed. However, when dealing with large definitive models (e.g. such as contain more than a thousand definitions), the system consumes a large amount of memory to store a script and takes a significant time to complete all the updates once there is a change. For instance, when running the timetabling model, it takes a significant time to update and set up all definitions (windows, trigger procedures and definitions) at the beginning. The system has to keep all dependencies up-to-date.

As discussed in Chapter 5, data representation in MWDS differs from traditional data representation. In a traditional approach, data modelling serves the purpose of helping in organising and handling data so that they can be referred to easily in practical applications [Mull99]. It is the initial abstraction that hides the complexity of the system. A data structure is designed to serve a particular application and to reduce the complexity to a level that a programmer can grasp and manipulate with programs. In contrast, in MWDS, representing data is similar to representing physical observables. Each variable exists atomically in the model (cf. Figure 5-9) but may be linked by chains of dependency to a large number of other variables. For instance, in the Lines model for 9 lines, we have a network of almost 1000 definitions indivisibly interrelated with each other and the value of one variable, viz. numcovedge, is determined by the values of more than 500 variables. This makes the data representation in MWDS very rich, but potentially expensive to maintain.

Maintaining large complex dependencies consumes a lot of resources to store both the variables and their dependencies. For instance, when running the Lines model for 10 lines, we receive a 'stack overflow' error. This is because the value of variable numcovedge is dependent on over 700 variables and the system does not have enough stack space to evaluate this variable.

7.2 Possible solutions

As discussed in the last section, MWDS raises many problematic issues. Most of them are related to understanding, organising and retrieving scripts. When dealing with a small script, these problems are easy to overcome. But when a large script is involved, these problems are prominent. Although the EM tools provide efficient features to help in tracing dependencies and values of variables, it is still hard to manually inspect the variables in a large script one by one.

Understanding a script involves both comprehending the internal semantic relation, and the external semantic relation. The support that can be given by the EM tools themselves and by the auxiliary techniques to be described in this section is mainly useful in respect of the internal semantic relation. Scripts can be treated as written text. They can be reorganised to give various interpretations. They can be generated with reference to their generic and abstract patterns. This section will discuss possible techniques that can be used, in conjunction with supporting practices and tools, to help in comprehending scripts and interaction with the model, and in generating scripts automatically.

7.2.1 Techniques to help in comprehending definitive models

- **Using the Awk-based tools to help in extracting and organising scripts**

Definitive scripts have a distinctive and relatively simple syntactic structure: the relationship between variables is expressed by different forms of definition. There are many ways to organise scripts in order to interpret them. In this section, scripts are treated as a plain text that can be manipulated and transformed by tools written in the Awk programming language [AKW88]. Awk is a powerful language to handle data manipulation, change the format of data and generate new sets of data with reference to existing ones by using pattern-matching techniques [DR90].

The Awk tools to be described in this section demonstrate how scripts can be automatically reorganised in different forms to assist their interpretation. They can be classified into three categories according to their primary objectives in respect of manipulating scripts:

1. Partitioning tools – used to organise scripts according to their underlying definitive notations;
2. Reporting tools – used to analyse characteristics of definitive variables defined using specific notations or in specific ways;
3. Script extension tools – used to automatically generate scripts, based on existing ones, that can be later added into the model to help in debugging and comprehension.

In this discussion, we do not give details of how the tools are written, but concentrate primarily on how these tools are used and help us to comprehend scripts and models. The Car History model is revisited and used as a case study. The Car History model was developed using the single file approach. Its script consists of several hundred definitive variables from different definitive notations. This makes it hard for the modeller or a new developer to understand and debug the script. The following discussion introduces tools of each of the above three types and illustrate how they can be applied in reorganising, analysing and generating scripts.

Partitioning tools

Definitive scripts can be used to represent various kinds of observations. The current EM tools feature three main definitive notations: Eden, DoNaLD and Scout, respectively used to represent observables for general-purpose use, observables associated with simple geometric entities and observables associated with display of windows and user interface devices. Each notation has different characteristics and syntax. Although DoNaLD and Scout scripts are translated into Eden script before their evaluation, it is easier to understand the relationships between definitions from their native syntax rather than from their Eden translation. By first understanding the relationships between the Scout and DoNaLD variables in the model, we can gain a rough grasp of the internal mechanism and the structure of the model, and can then proceed to examine the patterns of state transition in the model.

A partitioning tool has been written in Awk to extract the definitions in each of the three principal notations Eden, DoNaLD and Scout from a given script. This tool is invoked by the command *split-tkeden* , and runs on the Unix platform so that it can be used in conjunction with other Unix commands. It has three options ('e', 's' and 'd') to extract the Eden, Scout and DoNaLD definitive scripts respectively.

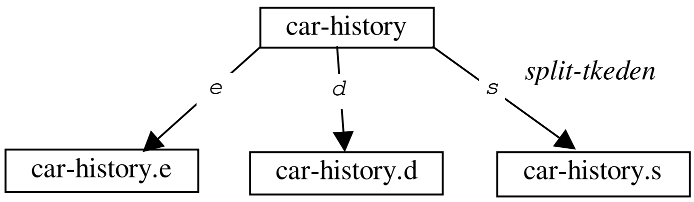

Figure 7-5: Illustrating of using *split-tkeden*

```
%eden
cars is ["Ford_Escort",
Ford_Escort, ...];

Ford_Escort is ["model",
fe_model, "exhaust", ...];
...
```

```
%donald
int Carx, clipsx, Enginx, ...
viewport Car1
openshape car
within cart{
outerref = if ~cartype==4 then {430,
660} else {390,700}
...
```

```
%scout
window title = {
        box: [tmin, tmax]
        pict: "Title"
        type: DONALD
        fgcolor: "blue"
        bgcolor: "white"
}; ...
```

Listing 7-1: Extracts from the scripts in the car-history (.e, .d, .s) files

By way of illustration, the script for the Car History model is written in one file. The *split-tkeden* command can be used to extract the Eden script as follows:

```
bash$ split-tkeden "e" car-history > car-history.e
```

Figure 7-5 illustrates how the *split-tkeden* command can be used to divide the entire script into three separate scripts that respectively contain Scout definitions, DoNaLD definitions and Eden definitions, functions and actions. The three scripts are stored in the files: car-history.s, car-history.d and car-history.e (cf. Listing 7-1).

After grouping the script according to its notation as shown in Listing 7-1, we can try to understand each script in isolation. Splitting a large script in this way can be helpful in locating errors. As was discussed in connection with the Lines model in Chapter 3, there are in fact many ways in which a large script can be organised. Other partitioning tools can in principle be developed to support alternative organisations of scripts.

Reporting tools

The Awk-based reporting tools introduced in this section are mainly used to help in analysing definitions relating to variables of specific types. Eden is dynamically typed, and there is no need to declare the type of an Eden variable. Scout and DoNaLD, on the other hand, are strongly typed and each type has a different abstract meaning. When the type of a variable has been ascertained, it is easy to anticipate what kind of structure the variable should have. For instance, a line is defined with a pair of points. The report generating programs described in this section are particularly useful for scanning DoNaLD and Scout scripts. They extract a list of variable names together with their types. Though the report generated is purely textual, it can be used in conjunction with the executing script. For instance, if the user wants to inspect any particular variable in detail (e.g. to determine its current value and direct dependees), he/she can use the (d)tkeden interrogating features described in Section 2.1.2. The reporting tools that have been constructed by the author are listed in Table 7-1.

Reporting commands	Descriptions
type.donald	type.donald <*filename.d*> List DoNaLD variable names together with their types.
type.scout	type.scout <*filename.s*> List Scout variable names together with their types.
proc.eden	proc.eden <*filename.e*> List Eden function and procedure names.
window.pict	window.pict <*filename.s*> List Scout window names of type DoNaLD and their associated DoNaLD picture names.
view.donald	view.donald <*DoNaLD picture name*> <*filename.d*> Extract the DoNaLD script associated with a DoNaLD picture name.

Table 7-1: A list of reporting tools

Figure 7-6 illustrates how the reporting tools: *type.donald, type.scout* and *proc.eden* can be used to generate reports for the Car History script. These commands can be combined using Unix pipe as indicated in the dashed rectangles in Figure 7-6. For instance, in Figure 7-6, the *split.tkeden* command is used to extract the Scout script and passes its output to the *type.scout* command, which in turn passes its output to the Unix **`more`** command.

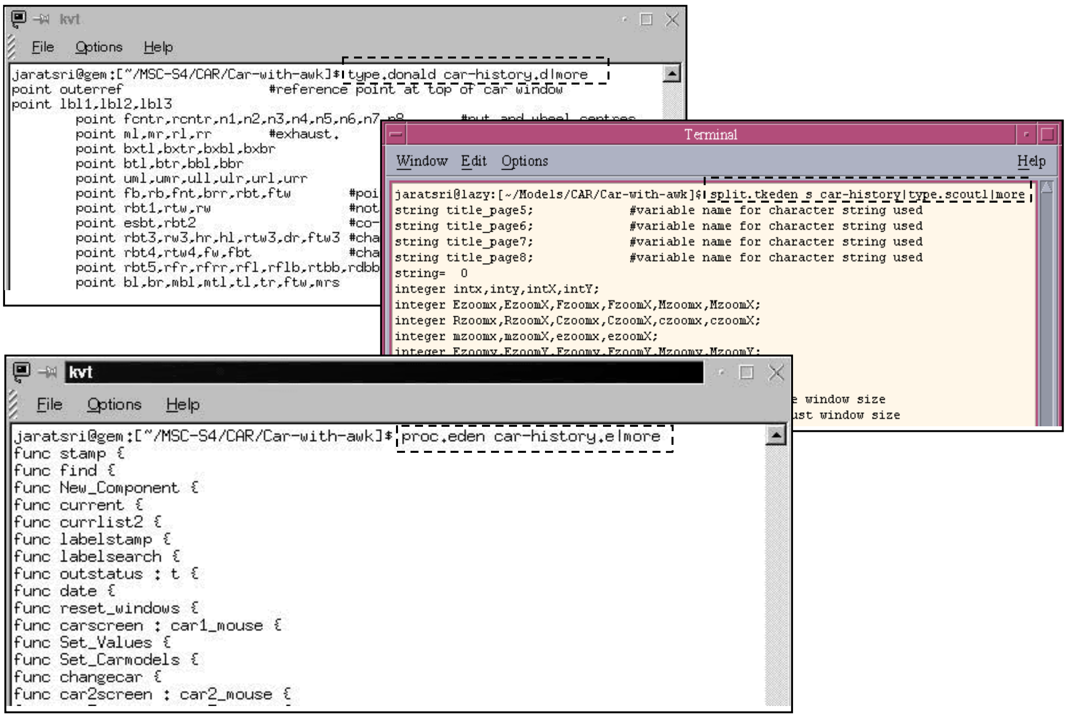

Figure 7-6: Screenshots of running the reporting tools on the Car History script

In some respects, the reporting tools are analogous to database queries. The scripts can be viewed as a representation of persistent data stored files, and the tools as query commands. The tools take advantage of the distinctive character of definitive scripts to provide special-purpose support for debugging and comprehending a script. For instance, the *window.pict* command scans through the script and identifies the names of Scout windows of type DoNaLD together with their associated DoNaLD picture names. Another program, *view.donald,* can then be used to extract the DoNaLD script for a particular picture name. For instance, the DoNaLD script for car as displayed in Figure 3-12 can be extracted by the *view.donald* command as follows:

```
bash$ view.donald car1 car-history.d
```

Script extension tools

This section introduces another technique to help in comprehending a definitive model that involves adding extra scripts to the model to provide interactive querying features. The script extension tools are used to read the original script so as to make a list of references and to generate scripts based on these references. When such generated scripts are interpreted in conjunction with the original script, they provide extra features to help in debugging and understanding the model. This technique is more interactive and can be used more readily in dynamic analysis of the model that the previous techniques discussed in early sections.

There are two phases in the use of a script extension tool. Phase 1 is concerned with generating a script. For instance, the command:

bash$ show.win car-history.s > car-history.show.win.

is used to transform the car-history.s script into a new script where there is new functionality for the middle-button mouse click[1]. In phase 2, the script generated in phase 1 (e.g. car-history.show.win) is introduced into the executing Car History model. With the additional script generated by show.win command, the definition of the window will be displayed as output if the user clicks on the middle mouse button when it is pointing at a particular window. This extra functionality is illustrated in Figure 7-7, where the user clicks on the middle mouse button when it is pointing at the central window, and the definition of that window is displayed as output.

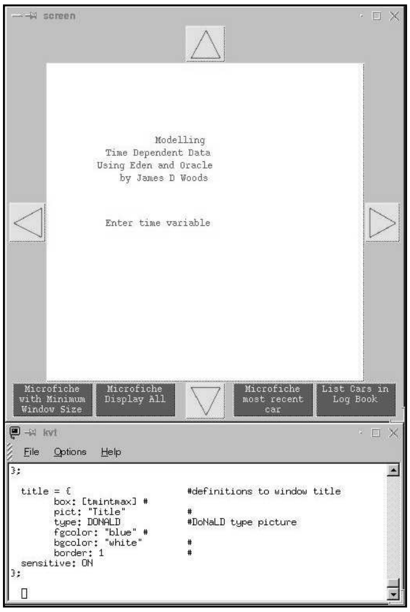

Figure 7-7: The Car History model with additional script generated from the tools

After adding this script, the essential functionality of the model remains the same. It is like extending the model by including a layer of interpretation on top of the original model. This is

[1] This functionality will be supplementary to the original functionality of the middle mouse button (if any), when both scripts are interpreted together.

one example of how we can automatically generate scripts and such scripts can provide extra features to the model, in this case to help in understanding the internal semantic relation.

The techniques discussed in this section demonstrate how definitive scripts can be reorganised, analysed and automatically generated. This is because scripts are very open in character. The script in the model can be redefined and a new set of definitions can be added on top of the existing one. Scripts can also be treated as plain text so that a programming language for manipulating text such as Awk can be used to extract a particular set of definitions.

The tools and techniques discussed above illustrate how report generation can assist script comprehension. However, the best way to understand the script in the model is by experimenting with them. The more one interacts with the model, the more one understands and absorbs the concept behind its construction and representation. This motivates special-purpose script extension techniques that can be used to enrich specific models.

- **Special-purpose script extension**

This section describes another way to extend definitive models. Instead of using automatic tools to generate scripts, the developer can write scripts as an extended feature to help new users to interpret the original scripts. The developer may want to make the model more comprehensible and easily accessible. In this section, some examples are developed to show how the developer can extend his/her model in order to help the user understand or interact with it.

Dynamic annotation

A definitive script can represent diverse kinds of real world attributes. The DoNaLD and Scout definitions that specify a GUI interface (which consist, for example, of lines, points, windows and boxes) may be easy to comprehend, but some definitions represent abstract meanings conceived by the designer. Such definitions may be complicated for the new developer to understand. Furthermore, their current interpretation depends on their value and state, so that their full significance is best appreciated through interaction. For instance, consider the definition:

```
lock = hinge + if open then {0,-width} else {width,0}
```

taken from Figure 2-6. In Figure 2-6, the point which represents the lock in the DoNaLD line drawing is labelled using auxiliary definitions that are not listed in the script. This provides a simple form of dynamic annotation, whereby the position of the label is linked to the position of the point. An alternative application of the same idea might involve displaying strings of the form:

the door is open: lock is currently at hinge + {0,-width}
the door is closed: lock is currently at hinge + {width, 0}

according to the current value of the boolean variable open. Such annotation helps to translate between the coded computer representation of the door and its commonsense external interpretation.

As the above example illustrates, a definitive script has very great expressive power where the interpretation of state is concerned. It can be used to represent the current status of an observation or an abstract concept at a particular time. To each dependency, there correspond many different specific values for the observables it involves, and these stand in different relations to each other at different points in time.

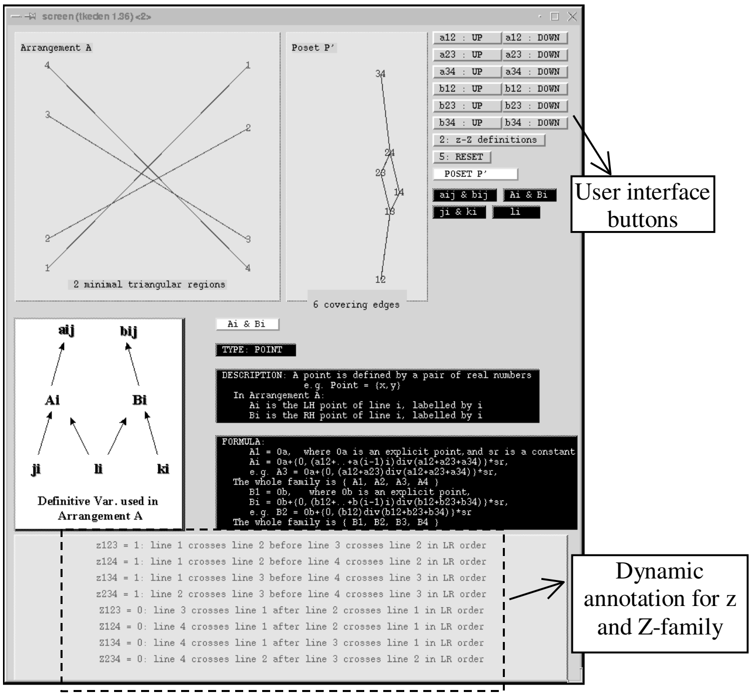

Figure 7-8: Special-purpose script extension of the Lines model

The purpose of dynamic annotation is to provide a richer interpretation of a particular script with reference to its current state. This is further illustrated in an extended version of the Lines model studied by the author in [Rung97]. As discussed in Chapter 3, the Lines model (cf. Figure 3-8) includes generic families of variables {z123, z124, z134, z234} and {Z123, Z124, Z134, Z234} and the value of each of the variables in these families is 1 or 0 depending on the relative positions of the points of intersection between the lines. Each variable can be dynamically interpreted depending on the state of the model (cf. Figure 7-8).

This modification of the Lines model demonstrates how the designer can include the dynamic explanation of definitions in the model through an extension that is itself a definitive script. Further extensions of the Lines model that assist the user in comprehension are discussed in detail in [BRSW98].

User interface

MWDS offers an open-ended modelling framework in which the model is being developed in parallel with the ongoing search for its requirements and specifications. The requirements for the model are gradually cultivated by the designer through open interaction in which any variable can be redefined at any time. Because interaction with the model is so open, it is hard for a new developer to know how to interact with it. To overcome this problem, the modeller can give access via a graphical user-interface to reliable and interesting patterns of interaction that have been found. Such an interface helps the human interpreter to interact with the model more systematically. It circumscribes the openness of the model but also provides guidance for the user concerning how to interact with the model. In this way, the modeller determines in advance how broad the user's interactions should be. The sequence of interactions can be devised by the modeller to prevent the user from destroying the model by mistake.

The introduction of interface may be unhelpful as far as understanding the qualities of a definitive model are concerned, since the user's interaction is unrepresentatively restricted. On the other hand, it may be quite helpful for the novice user to have limited scope to revise a model. The user does not need to spend much time on studying how to run the model. The model can be closed for the novice user and can be exposed for the expert user.

There are at least two motivations for developing interfaces to definitive models: to assist comprehension of the model and to adapt the model for conventional use as a device (cf. Chapter 5). An interface designed to assist the comprehension of the Lines model is depicted in Figure 7-8.

In MWDS, the user-interface can be configured dynamically and there is a possibility of close involvement of the user in the model development (cf. Chapter 5). Because of this, MWDS offers a way to develop interfaces to a definitive model that are 'user-centred' [NB99] in that their design is guided by how domain knowledge is to be processed so as to best meet specific user needs. The potential for 'user-centred' interface design supported by MWDS is illustrated in the Timetable model discussed in Chapter 4. The user interface of this model was added after the main data model had been developed.

7.2.2 Techniques to help in automatically generating definitive scripts

A definitive script can have a generic form, and in this case it is easy in principle to generate a new script from the existing one by generalisation. The scripts for the Heapsort and Lines models are both examples of scripts with a generic form. The script for the 15-element Heapsort model was originally derived from a 7-element Heapsort model by generalisation. An alternative to explicit manual generation of such generalised scripts is required. Two distinct techniques for automatic script generation are proposed in this section. One uses a 'template'; the other uses 'virtual agents'. Both these techniques generate scripts automatically, so that the developer does not need to enter scripts for a large model manually.

- **Using a template**

This method of script generation uses a definition as a template. Scripts are repeatedly generated with reference to the template. An example of this can be seen in the Heapsort and Lines models. Both models are similar in the way that their scripts are generic and can be enlarged. The concept of using a definition as a template to generate scripts is illustrated in Figure 7-9.

```
func makestr {
        para listl; auto l, result;
        result = ""; l = listl;
        while (l#!=0) {
                result = result // l[1]; shift l;
        }
}
a = 1;
templatej is ["j",ja," = label(\" ", ja, " \" ", "A", ja, ")\n"];
jdef is makestr(templatej);
ja is str(a);
func alljdef{
        auto i,result;
        result = "";
        for(i=1;i<=N;i++){
                autocalc = 0; a = i; autocalc = 1;
                result1 = result1 // jdef;
        }
        return result1;
} …
```

A function to convert a list to a string

The definition of a template

A function to generate a script to a particular pattern

Figure 7-9: Generating scripts using a template

The script extracted shown in Figure 7-9 generates a family of definitions for the following DoNaLD variables of type string:

```
j1 = label("1", A1)
j2 = label("2", A1)
j3 = label("3", A3)
j4 = label("4", A4)
```

The variable templatej, shown in Figure 7-9, is a template definition that records the essential structure of a family of *j* definitions. It is of type list. The function makestr will convert this list to a string and the function alljdef serves to convert the string generated from the template templatej to a string that represents a sequence of definitions. The variable a in the function alljdef is the key value on which templatej depends. The scripts that are generated in this way can either be written to a file or passed directly to the built-in Eden function execute() for evaluation (cf. Section 2.1.2).

- **Using a virtual agent**

The concept of virtual agent was proposed by Sun in his thesis [Sun99] and implemented as a feature of dtkeden (cf. Section 2.1.2). A virtual agent offers an alternative technique for auto-producing scripts which are similar to each other. Instead of using a template, as discussed in the previous section, a virtual agent offers a built-in syntax and command to serve this task. This way of generating scripts is more straightforward and is more closely integrated with the dependency evaluation in dtkeden. With this method, there is no need to generate scripts as strings and pass them to execute().

The use of virtual agents offers an easy and systematic method of generating a set of definitions. The definition of the Scout windows in the Timetabling model is a particularly good example of such use. As illustrated in Figure 4-2, the script for the display of staff buttons, as listed in Listing 4-4, is automatically generated by using virtual agents. The role of the template is played by a file in which the structure of the family of definitions to be reproduced is stored (cf. the listbutton_x2.s file in Figure 4-2). This file is included into a main file (cf. the listbutton.s in Figure 4-2) and the number of definitions based on this structure to be generated is determined by the for-loop specified in the listbutton.s file.

The use of virtual agents is subject to several limitations:

- The method generates variable names automatically. This means that there is a risk that name clashes will arise.
- Unlike templates, virtual agents can only generate scripts of certain limited forms. For instance, it is not possible to use virtual agents to generate the definition:

hcl is [hc1, hc2, hc3, hc4, hc5, hc6, hc7].

- Scripts generated using virtual agents may be difficult to understand and maintain since they are generated and evaluated at the same time rather than written to a file.
- The maintenance utilities suggested in section 7.2.1 cannot be applied to scripts generated using virtual agents, since they only act on explicit definitions.