2

Definition-Based State-Transition Models in the Abstract

It has been mentioned in the introductory chapter that our system may ultimately comprise many notations. Our current system can already relate six notations: DoNaLD, ARCA, Scout, EDEN, ADM and LSD. DoNaLD is a definitive notation for 2-D line drawings [BABH86]; ARCA is a definitive notation for displaying and manipulating a class of combinatorial diagrams [Beynon86a]; Scout is a definitive notation for describing screen layout [Yung88]; EDEN is a general definitive language for arithmetics, strings and lists [Yung87, Yung89]; ADM is a parallel definitive language [Beynon88b, BSY88, Slade89] and LSD is an agent protocol specification language [Beynon86b, Slade89]. Each of these notations addresses a specific domain. For this reason, each has its own set of data types and syntax. With such diversity of

notations, it is easy to lose focus on what the essence of definitive programming is. Therefore, this chapter will abstractly describe the core of the definitive paradigm – the Definition-based State-Transition (DST) model – before we discuss particular aspects of these definitive notations or languages in more detail in later chapters. This chapter will also discuss the virtues of definitive programming in relation to exploratory software development.

2.1. The Definition-Based State-Transition (DST) Model

In the early stages of the research on the definitive paradigm, emphasis was laid on generalising the "spreadsheet" principle to more general programming notations [Beynon85, Beynon88a, Beynon89]. Until the development of the LSD notation in 1986, the kind of interaction involved was still confined to "redefinitions by user". The LSD notation described a system in terms of processes (and later agents [Slade89]) interacting with each other. Since then, research on definitive programming has been widened to address general-purpose programming. Of particular interest in our research is the programming principle embedded in what we have called *definitive programming* [BNS88, Beynon88b, BRSYY89, BNRSYY89, BSY90], viz the use of sets of definitions to represent computational states. The understanding of definitive programming in terms of states and transitions evolved gradually; the phrase "definition-based state-transition model" first appeared in our papers as recently as 1989. The major work on studying the DST model started then.

A definition-based state-transition (DST) model is a state-transition model in which a state is represented by a set of definitions – a *definitive script* – and a transition is represented by a redefinition. A redefinition has essentially the same significance as a definition. The term *redefinition* is appropriate because a definition will overwrite the previous definition of the variable concerned whilst, if the variable has no current definition, the new definition will be added.

If a definition has to be discarded from a state, it is equivalent to redefining the variable by a special undefined value. This is because we can imagine that the state is a universal set of variables in which the variables are defined with undefined values by default.

2.2. Comparison of the Concept of State and Transition in Different Programming Paradigms

Petre and Winder categorise programming languages along a continuum between two extremes of computational models – the imperative model and the declarative model [PW88]. The imperative model is the computational model of the von Neumann machine. This is a model of "computation by effect". Under this model, algorithms are expressed as a sequence of changes of states. An imperative program contains explicit instructions for controlling the flow of execution. The declarative model, on the other hand, is a model of "computation by value". There is no sense of instruction in a declarative model, instead there is a "script"[1] which defines what is to be computed. Petre and Winder argue that there is a *continuum* of languages associated with the shift from an imperative to a declarative style that involves the transference first of explicit control and then of algorithmic information from the program description to the implementation.

It is obvious that the imperative languages are state-based languages. The declarative languages are arguably stateless in the sense that they describe an abstract input/output relationship rather than any computational state. From another perspective, we might reason that a script is concerned with the description of only one

---

[1] David Turner introduced the term "script" for the programs written in his functional languages to emphasise that such programs were qualitatively different from their imperative counterparts [Turner85].

state – the encapsulated behaviour of the required program (cf Chapter 7). In either way, the concept of state is not significant in declarative languages.

It is generally accepted that procedural programs are hard to verify and are more difficult to adapt onto parallel machines [Baldwin87]. However, we believe that some kinds of activities in the real world are most conveniently described by procedures. For example, a person may like to pick up a book from a bookshelf, walk to a desk and put the book on the desk. The behaviour of a person naturally comprises a sequence of actions which is most appropriately represented by a procedure.

In definitive programming, a state is prescribed by a definitive script. The maintenance of values of variables in a definitive script is similar in spirit to declarative programming in that it involves implicit evaluation of expressions. In contrast to declarative programming, definitive programming does not presume a declarative style of specifying transitions. (Indeed, later in the thesis, we advocate the use of an agent-oriented style for specifying state transitions.) There is, therefore, scope in definitive programming to explore the virtues of both declarative and procedural programming.

Applying Petre and Winder's classification method, the position of the DST model in the continuum from "imperative" to "declarative" is somewhere in between the two extreme models but its bias may vary depending upon the way of specifying state transitions. On one hand, the DST model has variables and a concept of state. In this way, definitive programming is similar to the imperative model. On the other hand, the values of the variables are not necessarily directly modified by a program instruction (there may not be any). This is because a definitive variable is fundamentally defined by a formula instead of an explicit value. That is, its value is determined by what it is asserted to be rather than by direct assignment. Therefore, if the language that governs the state transitions is procedural, the DST model would be biased towards the imperative model. Otherwise, the DST model would be biased towards the declarative model.

In view of our freedom to choose the specification style for state transitions, we can identify the main characteristic in the DST model to be its novel approach to state representation. Therefore, in the rest of this section, we will focus on comparing the concept of state in different programming paradigms.

A definitive script specifies the following information pertaining to a state:

1) a collection of values (values of variables),
2) references to the components of the state (variable names),
3) data dependency information between components of the state,
4) methods of maintaining the state (formulae).

In the following sub-sections, the state representation methods of conventional imperative programming, functional programming and object-oriented programming are compared with that of definitive programming.

2.2.1. Conventional Imperative Programming

In conventional imperative programming, a state is a collection of variables containing explicit values. The machine changes the state by assigning new values to the variables. The connection between the values of the variables cannot be observed by looking at one state only. The meaning of a variable cannot be understood without referring to the program; the meaning of a variable may even change from one state to the other during program execution. For example, in the following program fragment

```
1  sum := 0;
2  for I := 1 to N do
3       sum := sum + a[I];
4  mean := sum / N;
5
6  sum := 0;
7  for I := 1 to N do
8       sum := sum + (a[I] - mean) * (a[I] - mean);
9  sum := sum / N;
```

the meaning of sum in line 4 is the summation of N numbers but the meaning of sum has changed to the variance of the N numbers after line 9, and at other points, sum is a storage of intermediate results.

Definitive programming, to certain extent, gives meaning to the variables. A definitive variable is defined by a formula. This formula is the 'value' of the variable. The formula prevails until the variable is redefined by another formula. However, the value of the formula may change over time as the variables in the formula are redefined. Therefore, there are two levels of understanding a definitive variable: knowing its interpretation (associated with the formula itself) and knowing its current value (the value calculated from the formula).

Backus in his much referenced paper "Can Programming be Liberated from the von Neumann Style?" [Backus78] points out two main problems with conventional languages: word-at-a-time bottleneck and splitting programming into an orderly world of expressions and a disorderly world of statements.

Word-at-a-time bottleneck is the input/output limitation of the von Neumann machine model. This is reflected by the basic operation in a conventional procedural language – each atomic state transition allows a change to the value of just one variable (a single assignment). Because of advances in computer architecture, this word-at-a-time bottleneck no longer applies to computer hardware. It is now the conventional procedural language that imposes this bottleneck. Definitive programming breaks this bottleneck by allowing indivisible changes of values of many variables in a single transition of state. When the definition of a variable is changed, not only the value of this variable will be updated but also the values of those variables defined in terms of this variable.

Backus uses the phrase "the orderly world of expressions" to refer to the expressions on the right hand side of assignment statements. He claims that an

expression has useful algebraic properties whilst a statement has few useful mathematical properties. Using expressions in the context of assignments destroys the usefulness of the algebraic properties of expressions by side effects. In contrast, the definitive state representation preserves the usefulness of the algebraic properties of expressions by persistently associating the expressions with variables. A change of value induced by the change of other variables does not alter the expression associated with that variable.

2.2.2. Functional Programming

Functional programming and definitive programming are, in principle, not comparable concerning states and transitions because functional programming is stateless. A functional script defines what is to be computed rather than how to compute the target value. Neither is there a concept of procedural variable in functional programming; a mathematical variable is officially not allowed to vary [BR89].

A disadvantage of functional programming arises also from the lack of the concept of state; such a concept is almost indispensable for describing states and transition in interactive programs. Dataflow languages (a branch of functional languages) are more promising in handling interactive programming. Wadge's VISCID program is an attempt to write a vi-like[2] screen editor in LUCID [Wadge85]. Other attempts at writing screen editors in other non-procedural languages like Prolog and Lisp[3] used side-effects and imperative features; Wadge tried to show using VISCID that it is in fact possible to write non-trivial and non-mathematical applications within the constraints of a functional language. However, VISCID relies on the lazy evaluation strategy to control what Wadge has called "internal memory" variables. (Lazy

---

[2] Vi is a standard UNIX full screen text editor.

[3] Because there are imperative features in Lisp, it is seldom considered as a functional language. However, we can extract a functional subset from Lisp [GHT84].

evaluation (call-by-need) is used in preference to eager evaluation, where a function is evaluated as soon as all the arguments are evaluated. If LUCID ran using an eager evaluation strategy, VISCID could only perform batch mode editing rather than interactive editing.) Based on the fact that the execution of VISCID relies on a particular evaluation strategy and that it has a notion of internal memory, we shall argue that although we *can* use a functional language to program interactive applications, programming in a state-based language would be a more satisfactory solution.

Although definitive programming and functional programming adopt significantly different programming models, a definitive script (a set of definitions) and a functional script have a useful mathematical property in common: a script (in either paradigm) will always evaluate to a unique set of values. It is even plausible to argue that a functional script is a definitive script (see §7.1).

Hudak did a survey on functional programming [Hudak89]. The survey includes a discussion on the active research areas in functional programming. It is interesting to note that there are researches going on in the direction of integrating functional and imperative programming [Lucassen87]. It has been indicated in the last sub-section that we are not promoting imperative programming. However, we emphasis the importance of state-based programming. Although definitive scripts and functional scripts are superficially similar, their interpretation is fundamentally different: our systems recognise on-line redefinition of scripts as part of the computation. This provides a basis for interactive programming using definitive representations of states.

2.2.3. Object-Oriented Programming (OOP)

Many of the ideas behind object-oriented programming have roots going back to SIMULA [DN66]. The first substantial interactive, display-based implementation was the SMALLTALK language [GR83]. Associated with the widespread use of the C language, extensions of C – such as Objective C [Cox84, Cox86] and C++ [Stroustrup84, Stroustrup86] – are also widely used. There is one thing in common

with all these languages – they are all procedural. Hence, OOP gives some people a first impression that it is fundamentally procedural. However, there are a considerable number of object-oriented extensions to non-procedural languages. Loops [BS81], CommonLoops [BKKMSZ86], OakLisp [LP86] and CommonObjects [Snyder87] are some object-oriented extensions to Lisp; SCOOP claimed to be object-oriented Prolog [VLM88]. Hence, it is possible to merge OOP with other programming paradigms. Object-oriented programming is more appropriately understood as a design philosophy [Meyer88, Booch91, WP88].

Cox has described object-oriented programming as an evolutionary development from procedural programming [Cox86]. In particular, the concept of data in object-oriented programming has evolved from a procedural framework. An object is more than a simple value (for example a floating point number in Fortran), or a group of values (for example a structure in Pascal); an object has some methods associated with it to specify how it is to be maintained.

Definitive programming can be viewed as a different kind of evolution from data specification in a procedural style. Definitive programming enriches the meaning of data by assigning to the variables formulae instead of plain values. In a way, we may consider that a definition has already provided a method (the formula) for the maintenance of the variable defined, so that simply grouping the related variables together has a flavour of object-oriented programming. This suggests that object-oriented programming and definitive programming can be usefully combined. Yung has given some suggestions for object-oriented EDEN [Yung89] and Chapter 7.2.3 in this thesis includes a proposal for adding inheritance to the DoNaLD notation.

2.3. Virtues of the DST Model

2.3.1. Data Dependency, Concurrency and Consistency

An important area of concern in studying different programming paradigms is the support for recording and retrieving data dependency information. Data dependency information is useful in two areas: concurrent programming and program development.

With data dependency information, the compiler can automatically distinguish when two operations must be done sequentially because one produces or destroys a value that the other needs. Therefore, the more easily the data dependency information can be obtained, the better the program is suited for parallel processing.

In definitive programming, an acyclic graph of dependency can be drawn from a set of definitions and concurrent updating of the variables can be performed in each layer of the graph. Such a scheme for concurrent maintenance of definitions is discussed in depth in [Yung89]. This way of parallelisation is a kind of data parallelisation (i.e. parallel evaluation of data) which can be determined implicitly by the system. Since it is hard to prove the correctness of those parallelisation schemes given explicitly in a program, implicit parallelisation is more reliable. Moreover, data parallelism is highly effective [Baldwin87, HS78]. Definitive state representations seem to be a good foundation for concurrent programming.

Turner [Turner83] and Landin [Landin66] predicted the future trend for the development of programming languages would be non-procedural languages. One reservation they have about the development of non-procedural languages is that "on conventional von Neumann computers, non-procedural language runs two to three orders of magnitude slower than traditional imperative languages" [Turner83]. Turner suggested that non-procedural languages can be used as effective tools for software prototyping while waiting for the development of systems suitable for non-procedural languages. In fact, many non-von Neumann system architectures are developing: for

example dataflow machine architecture [Veen86, Sowa87], dataflow / von Neumann hybrid architecture [Iannucci88], parallel logic inference machine [Clocksin87, Jorrand87] and object-oriented computer architecture [Harland88].

Speed of execution is significant, but the efficiency of developing a program is of equal importance. The identification of data dependency provides useful information in maintaining a program during program development. Consider the following scenario. Suppose that the value of a in an imperative language, at a stage of program development, had to be maintained to the same value as $2 \times b$, but some time later the programmer determined to alter this assertion to "a equals to $3 \times c$". Then what the programmer has to do is to remove all the assignment statements of the form "$a = 2 \times b$" and insert assignments "$a = 3 \times c$" after each modification of the variable c. If some of the "$a = 3 \times c$" statements were, by mistake, not inserted, the old value of c would be retained in a at certain points of the program. Or if some of the "$a = 2 \times b$" statements were not deleted, then a might obtain a value totally unrelated to c. Understanding data dependency assists the programmer to know exactly what actions have to be done to the program when the specification is altered.

Definitive programming not only makes use of the data dependency information to maintain the consistency of data, it actually prevents inconsistency. Because there is only one persistent definition of a variable stored inside a state, no redundant information and hence no potential inconsistency of data will be found in a definitive script[4].

---

[4] Relational database theory also acknowledges that redundancy leads to potential inconsistency. The relational database designer prevents update anomalies (potential inconsistency) by decomposing a large database into normal form [Ullman82]. Basically, the decomposition schemes that are employed group the fields of the database into sub-databases according to the dependency of the fields. A definition is similar to a single relation within a relational scheme in that only related variables are linked together.

2.3.2. Support for Incomplete Models

It is clear that, in certain contexts, such as during the construction or modification of a model, some variables are not evaluable because the dependent variables are not defined. This does not affect other parts of the model that do not depend on these undefined variables. The partially completed model may still have a meaning: a room without furniture is still a room. Even when a definition depends on undefined variables, its defining formula is also meaningful, not least for the purposes of analysis. A definition limits the possible values of the variable. This point will be elaborated in the next section.

2.3.3. Possible Transformation

Just looking at a set of values gives us little information for reasoning about what these values mean and how they should be manipulated. The following two DoNaLD specifications[5] both produce the shape shown in Figure 2.1.

By looking at the shape alone, it is not possible to guess which specification is the one that generates this shape. This shape may represent a file cabinet with its drawer opened, or it may represent a LED display which is showing the digit 8. A correct interpretation of the shape can only be made with reference to the underlying model in mind, which means a state of an object is more than a set of values (say pixel values). Definitions relate the state and the model in such a way that changes in the model reflect changes of external state; the possible transformations to the object are described in a set of definitions.

---

[5] The example is taken from [BCRY90].

```
openshape cabinet
within cabinet {
  int     width, length
  point   NW, NE, SW, SE
  line    N, S, E, W

  N = [NW, NE]
  S = [SW, SE]
  E = [NE, SE]
  W = [NW, SW]

  width, length = 300, 300

  SW = {100, 200}
  SE = SW + {width, 0}
  NW = SW + {0, length}
  NE = NW + {width, 0}

  openshape drawer
  within drawer {
    boolean  open
    int      length
    line     N, S, E, W

    length = if open then ~/length else 0
    open = true

    N = [~/NW + {0, length},
         ~/NE + {0, length}]
    S = [~/NW, ~/NE]
    W = [~/NW + {0, length}, ~/NW]
    E = [~/NE + {0, length} , ~/NE]
  }
}
```

```
openshape led
within led {
  int      digit
  point    p1, p2, p3, p4, p5, p6
  line     L1, L2, L3, L4, L5, L6, L7
  boolean  on1, on2, on3, on4, on5, on6, on7

  digit = 8

  p1 = {100, 800}
  p2 = {100, 500}
  p3 = {100, 200}
  p4 = {400, 800}
  p5 = {400, 500}
  p6 = {400, 200}

  on1 = digit != 1 and digit != 4
  on2 = digit != 0 and digit != 1 and digit != 7
  on3 = digit != 1 and digit != 4 and digit != 7
  on4 = (digit == 0 or digit >= 4)
        and digit != 7
  on5 = digit == 0 or digit == 2 or digit == 6
        or digit == 8
  on6 = digit != 5 and digit != 6
  on7 = digit != 2

  L1 = if on1 then [p1, p4] else [p1, p1]
  L2 = if on2 then [p2, p5] else [p2, p2]
  L3 = if on3 then [p3, p6] else [p3, p3]
  L4 = if on4 then [p1, p2] else [p1, p1]
  L5 = if on5 then [p2, p3] else [p2, p2]
  L6 = if on6 then [p4, p5] else [p4, p4]
  L7 = if on7 then [p5, p6] else [p5, p5]
}
```

Listing 2.1: Two DoNaLD Specification for Describing the ⊟ Shape

Figure 2.1: An ⊟ Shape

If we are interested in the ⊟ shape only, so that no more change to the shape is needed, this information about possible transformation becomes redundant. Confusion may arise here as to whether the transformation information should be classified as part of a state. The answer can be established using the following illustration. In a

procedural graphics drawing package, it is possible to transform a geometric object to another geometric object via addition and deletion of line segments or other operations. A transformation from a  shape to an  shape may take the following sequence:  ,  and  . Does it mean "3 + two lines = 8"? Of course not. The interpretation of this  shape as the number 8 cannot be justified. Using definitions to represent a state of an object models the object more faithfully because, on top of a set of values, the possible transformations about the object are described as well.

2.3.4. Exploratory Software Development

To assist in exploratory software development, definitive programming provides:

- a modelling principle: a set of definitions *models* a state. This helps in the understanding phase of software development.
- data consistency. By means of definitions, values of variables will be maintained to their associated formulae. This implies fewer errors during editing a definitive program and easier for debugging.
- good prospects for efficient execution. The potential for data parallelism in evaluating definitive scripts is an advantage for allowing more explorations in shorter time.

These lay a solid foundation for developing definitive programming into an exploratory programming paradigm.

The possibility for software exploration makes reasoning about the properties of definitive programs difficult. This reasoning issue has to be addressed in the future, it is probably associated with the issue of specifying the intended use of definitive programs and the privileges of the users.