7

Generalising Single-Agent Definitive Systems

In the last three chapters, the Definitive State-Transition model has been applied in the simplest way – all transitions to a definitive state are directly manipulated by the user. We have called this kind of system a "single-agent definitive system". The user is the only agent who initiates transitions between definitive states. The role of the computer is simply to maintain the definitive state, i.e. to maintain the values of the variables to be consistent with their formulae.

The integration scheme described in the last chapter indicates that a definitive state can be specified using an extensible underlying algebra. If the existing definitive notations are not suitable for a particular application, we may consider designing and implementing a new definitive notation. For instance, although we can draw real-

valued function graphs using DoNaLD (an example can be found in [CB92]), it would be better to have a separate definitive notation for plotting graphs. Therefore, conceptually, although a definitive script describes only one state, this state can be very complex.

This discussion leads us to consider whether single-agent definitive programming can be an appropriate general-purpose programming paradigm. Functional programming has already demonstrated that it is possible to represent an executable program using a single script that superficially resembles a definitive script [Research87]. Therefore, we are tempted to ask: "what are the practical differences between a definitive script and a functional program?". In responding to the question, Admira – A Definitive MIRAnda – is prototyped to assist the comparison of definitive notation and the typical functional language Miranda.

By using "programming a stack-based integer desk calculator" as a simple case study in §7.1.3, we find that a definitive script cannot specify interaction satisfactorily. To develop a general-purpose definitive programming paradigm, we must introduce mechanisms for state transition that is independent of the user. This is what we mean by "generalising single-agent definitive programming".

In §7.2 and §7.3, we consider what supplementary information is required to relate single-agent interaction with a definitive script to general-purpose programming. The Trident software briefly introduced in Chapter 5 shows that the gap between a definitive script and an executable program can be very narrow. In §7.2, we consider some techniques for the indirect construction of definitions that require the computer to take on a more active role than maintenance of variable consistency. In §7.3, we describe an extension to the Scout system to allow interaction with a definitive script to be event-driven. The work of this chapter leads us to consider a more general way of describing transitions to complement a definitive script. This will be the subject of the

next chapter, where the *agent* is introduced as a formal concept for specifying state transitions.

7.1. Definitive Programming vs Functional Programming

7.1.1. Design and Implementation of Admira

There are times when we want to define functions and higher order functions. For example, the script for the visualisation of line arrangements described in the last chapter can be written down more neatly if there are sorting and counting functions. The EDEN (imperative) way of defining functions and procedures involves a different paradigm for state representation; improper use of the procedural elements of EDEN will introduce the traditional problems of procedural programming into the definitive paradigm. On the other hand, functional programming does not provide referential information which is significant in specifying state [Beynon89]. To clarify these issues and to understand more about the relationship between functional programming and definitive programming, we have carried out a project to prototype a definitive notation for functions. The definitive notation is called Admira – A Definitive MIRAnda. Admira allows interactive definition and redefinition of functions and on-line function value queries. All these are done in a definitive fashion[1].

For the purpose of comparison, the underlying algebra of Admira is deliberately chosen to be as close to Miranda's as possible. Admira accepts almost the same grammar as Miranda does. The discrepancy between the two grammars is minor; the reason for altering the grammar is to ease the implementation of the prototype.

---

[1] Some functional programming environments like SML [Harper86] and KRC [Turner81] do support interactive redefinition of functions, but in a significantly different manner. In these systems, functions are evaluated at their point of entry. A redefinition of a function will not affect other functions even though their definitions make use of the function being redefined.

7.1.1.1. What Is an Admira Definition?

Each variable in Admira is of the function type; an Admira definition is a definition of a function. Since Admira adopts the syntax of Miranda, an Admira definition is essentially the definition of a Miranda function. It is obvious that what is called an equation in Miranda [Research87] can be treated as an Admira definition. Some functions in Miranda consist of more than one part. For example, for functions defined by cases, for functions with where-clauses, and for functions defined by pattern matching, the functional definitions can be naturally divided into parts, some of which may themselves be function definitions. However, these parts collectively form an Admira definition. In these cases, the name of the function is considered to be the definitive variable to be defined, and the whole definition of the function is considered as a definition. To assist the determination of the end of a definition, Admira requests a full stop at the end of each definition.

7.1.1.2. Storage of Definitions

While checking the syntax of the definitions, Admira keeps a list of all the dependent functions. For example the following defines 5 definitions (square is not counted because it is local to the definition of variance):

```
variance X = mean(square(X)) - sqr(mean(X))
                  where
                        square []  = [];
                        square a:X = (sqr a):(square X);
                  end.
sqr X = X * X.
mean X = (summation X) / (nitems X).
summation []  = 0;
summation a:X = a + summation X.
nitems []  = 0;
nitems a:X = 1 + nitems X.
```

Listing 7.1: An Admira Script for Calculating Variance

The dependency of the functions can be summarised in the following table:

Function Name	Dependent Functions
variance	mean, sqr
sqr	
mean	summation, nitems
summation	
nitems	

When the definition is put in the definition store, the name of the function, the function definition and the list of dependent functions are recorded.

7.1.1.3. Evaluation of Expression

The implementation of Admira does not follow the pattern described in §6.3. Definitive scripts in other definitive notations are translated into EDEN and the EDEN interpreter evaluates the definitions. Because the grammars of Admira and Miranda are so close, it is much easier to use the Miranda interpreter – mira – as the evaluation engine of Admira. When an expression is going to be evaluated, Admira will identify which variables have to be evaluated. These variables are those free variables in the expression together with their dependent variables. The functional definitions of these variables are then collected into a Miranda script. This Miranda script is sent to mira to provide an environment for the evaluation of the required expression. Finally, the expression is sent to mira for evaluation and the result is displayed through mira.

The complete implementation of Admira and the grammar accepted by Admira can be found in Appendix B and C.

7.1.2. Recursive Definition and Circular Definition

Superficially, Admira simply integrates the editing, compilation and the evaluation processes of functional programming, but by analogy with the use of other definitive notations such as DoNaLD, we wish to view an Admira script as a representation of a

state. Just as a line in DoNaLD may represent the state of a door, we would like to interpret a functional value in an Admira script as representing the state of an object[2]. We can then change the state (as represented by a set of functional definitions) by defining a new variable (introducing a new functional definition) or redefining an existing variable (changing the definition of the variable associated with an existing function name).

The existence of the Admira prototype suggests that everything that functional languages can do can also be done in a definitive style. This false impression arises because Admira is not executing in exactly the same way as other definitive systems do – Admira does not check for circular dependency of variables. Even with circular dependency, Admira will try its best to evaluate the variables. For example, the script:

```
ones = 1 : ones.

f 0 = 0;
f n = (g n) + 1.

g n = f (n-1).
```

recursively defines a value ones and two functions f and g (f is an identity function and g(n) returns n–1). These values and functions can be evaluated by Admira but the definitions cannot pass the dependency checking scheme applied in other definitive notations.

One reason for avoiding such rigorous dependency checking is the difficulty of distinguishing between circular definitions and recursive definitions from their form alone. The above examples clearly show that recursive definition is a concise way of

---

[2] In practice, it has been proven difficult to make use of an Admira script to represent a system state. This may be connected with the fact that Miranda data types were designed with abstract computation in mind, and are not directly associated with observable attributes of real world objects.

defining a complex data value. Circular definition, which also has self-referencing, is an attempt to define a data value in terms of itself, which is incomprehensible[3].

It is certain that circular definitions are not acceptable in definitive programming. Since recursive definitions bear the same form as circular definitions, how we should interpret recursive definitions such that they are compatible with ordinary definitions?

Recursive functions in functional programming have to have an operational interpretation. For example, the definition "ones = 1 : ones" in Miranda defines ones with the value of a list of 1's, but the definition "ones = ones : 1" will cause a *black hole*. The interpretation of a functional program requires the understanding of the evaluation mechanism. On the other hand, definitive notation is aimed at defining relationships between data. The interpretation of a definitive script does not need to refer to the evaluation mechanism, and preferably should not. The relationship between DoNaLD and Scout can illustrate this. With Scout, writing a DoNaLD script is like creating 2-D drawing on a large worksheet, just like working with pen and paper. The drawing is then viewed through a Scout window. Recursive definitions and ordinary definitions are therefore, in principle, defining objects in two levels. Respectively they are an operational level and a real world level. Their relationship has a parallel in the context of numerical analysis.

During evaluation of a function using a computer, we recognise two types of errors – truncation error and round-off error [BF85]. The term "truncation error" generally refers to the error involved in using a truncated or finite summation to approximate the sum of an infinite series. In other words, it is an error due to the use of an approximation function of the actual function. The "round-off error" is the error

---

[3] The name of God in Hebrew literally means "I am who I am", which is analogous to a circular definition I = I. No one can fully comprehend God, nor circular definition.

that results from replacing a number with its floating-point form. The round-off error affects the results in two ways. Firstly, it affects the parameters supplied to the functions and secondly, it affects the accuracy of the calculations. Therefore, when calculating "y = f(x)", y will in fact be given the value F(X) where F is the approximation to the function f and X is the approximate value of x. Although y is not assigned the intended value, the relationship y = f(x) still holds in the conceptual level. Only when we consider the computational level do we need to know what F and X are.

Recursive definition and ordinary definition are therefore addressing the two different kinds of relationships during computation. Whilst ordinary definitions address relationships such as those between y, f and x, recursive definitions address relationships such as those between f and F, and between x and X. Recursive definitions could be a means of specifying the underlying algebra over which ordinary definitions are defined. We prefer to lay down a computer model in terms such as π and sin() which are the ideal real values and functions in the real world; an implementation, on the other hand, is allowed to deviate from the real values and functions. For instance, an electronic calculator has a π key, but internally the π may only be a ten-digit figure.

If we take this view, we should be able to divide definitions of an Admira script into two kinds. One kind is for modelling the relationship between data, and the other for specifying the implementation of the underlying algebra. Unfortunately, because all variables and operators are functions, it is hard to distinguish from their form which definitions are defining the underlying algebra and which are not. In other definitive notations, the problem may seem smaller. This is because the values of the variables in other definitive notations cannot be treated as operators. However, it is still difficult to differentiate recursive definitions of values in the underlying algebra and circular definitions of variables without consulting their roles in interpretation.

Definitions as part of the definitive state model are qualitatively different from other functional definitions of the implementation of underlying algebra. The mixing of the two in a script, as demonstrated in an Admira script, surely makes the script difficult to understand. If part of the implementation of the underlying algebra is likely to be publicly made known, for example if a definitive notation allows user-defined functions, it is advisable to specify these functions in a section dedicated for specifying implementation.

7.1.3. State and Interaction

The difference between definitive programming and functional programming is highlighted by considerations of interactive programming. In definitive programming, the way in which states and possible interactions with states are represented is strongly related to the way in which we choose to observe a system. Take observation of a jar of gas as an example. A physicist wishing to study the jar of gas at a microscopic level might model the state of the gas in terms of the properties of individual gas molecules. These include the mass, the positions and the velocity of the gas molecules. The molecules interact with each other through collision. A chemical engineer, on the other hand, might be interested in macroscopic properties of the state of the gas such as pressure and temperature. Associated with different ways of recording the state are different ways of describing interaction. For instance, the physicist may like to change the velocity of the gas molecules on the wall of the jar whilst the chemical engineer may like to change the temperature of the jar.

Some models of state are simpler than the others. For example, the pressure of the jar of gas can be related to temperature by a simple formula, but the motion of individual gas molecule is harder to calculate. However, we argue that the choice of a model for representing states and interactions is essentially determined by how we observe. We should ask, "what kind of state description is appropriate under the method of observation in use?" [BR92]. It is inappropriate to obtain the pressure of

gas by an optical measuring instrument. A barometer, on the other hand, cannot indicate the positions of individual gas molecules.

Traditional solutions to the problem of interactive programming in a functional paradigm ignore the issue of choosing an appropriate mode of observation. For example, consider the problem of writing a simple stack-based integer desk calculator in a functional style. The following is a solution taken from [Wadge85]. It is written in pLucid, a functional dataflow language.

```
hd(stack) whenever command eq "p"
   where
      stack = [] fby
            if isnumber(command) then command :: stack else
               case command of
                   "+": (op1 + op2) :: stack2;
                   "-": (op1 - op2) :: stack2;
                   "*": (op1 * op2) :: stack2;
                   "/": (op1 div op2) :: stack2;
                   "p": tl(stack);
                   default: error;
               end
            fi;
      op1 = hd(stack);
      op2 = hd(tl(stack));
      stack2 = tl(tl(stack));
   end
```

Listing 7.2: A Stack-based Integer Desk Calculator in pLucid

The stack program above accepts an input variable command which is a stream of stack operations and output a stream of values. Stack, as well as op1, op2 and stack2, are what Wadge has called *time-varying values*. If we attempt to output stack in the course of calculation, we notice that stack itself is a data stream. If we consider the functional program in Listing 7.2 as a description of a single state, the state under observation is a stream of stacks. This is not ideal because we normally observe a stack at a time rather than a stream of the stacks. It is, therefore, unnatural to write a definitive script that describes a 'stack' which is in fact a collection of many stacks. This example indicates that although we could avoid specifying state transitions in definitive programming by adopting a complex underlying algebra as in Miranda and

pLucid, it is preferable to preserve transitions so that the state described complies with our usual understanding of its real-world counterpart. This is the reasoning that leads to the introduction of agents in specifying state transition.

So far we have no satisfactory solution to specifying a stack within a single-agent definitive paradigm, and we are doubtful if there is such solution without specifying some sort of agent to initiate transitions of states. In the following sections, we will consider a number of proposed and existing programming techniques related to current definitive notations that may be viewed as *agent-like*.

7.2. Agent-like Abstractions and Definitive Scripts

7.2.1. Meta-Definition

In our current systems, there are at least two contexts where meta-definitions (structures that generate definitions) are encountered. The first is found in a proposal of enhancement of the DoNaLD notation given in [Beynon89]. Listing 7.3 is a proposed DoNaLD specification for displaying rungs of a ladder.

```
1. openshape ladder (int n, point a, b)
2. within ladder {
3.     line L = [n.a, n.a + (b-1)]
4.     if n > 1 then {
5.         shape Ladder = ladder(n-1, a, b)
6.     }
7. }

8. shape Ladder = ladder(7, {1,2}, {2,1})
```

Listing 7.3: A Proposed DoNaLD Specification of a Ladder

In the example, lines 1 to 7 defines a general definition of a ladder, ladder, and line 8 specifies an instance of ladder, Ladder.

The shape Ladder contains no self-referencing because the references created in the expression ladder(7, {1,2}, {2,1}) are all distinct from Ladder. The specification of ladder creates the variable Ladder/L which refers to the top rung, the variable

Ladder/Ladder to the remaining set of rungs, of which Ladder/Ladder/L is the topmost, etc. Note that the definition of Ladder in line 8 defines not only the variable Ladder itself but also a set of variables L, Ladder/L, ... , Ladder, Ladder/Ladder and so on.

The second context where meta-definitions are encountered is in the for-loop and the with-loop in the ARCA notation. Taking the with-loop as an example, the following loop:

```
with int 3 : I = 2, 3 do
      D!(2*I) = rot(D!2, I-1, v)
      D!(2*I-1) = rot(D!1, I-1, v)
od
```

is an abbreviation for:

```
D!4 = rot(D!2, 2-1, v)
D!3 = rot(D!1, 2-1, v)
D!6 = rot(D!2, 3-1, v)
D!5 = rot(D!1, 3-1, v)
```

The meta-definitions in both the DoNaLD and ARCA examples above may be interpreted as shorthands for fixed sets of definitions. That is to say, we may substitute sets of definitions for the meta-definitions. This substitution is possible because the parameters of ladder in the DoNaLD script and the values of the index I in the ARCA script are constant values.

The use of meta-definitions with abstractly defined parameters, as in the DoNaLD specification

```
shape Ladder = ladder(m, {x, y}, {2, 1})
```

raises more difficult issue of interpretation. Suppose that a meta-definition M with a parameter P produces f(P) ordinary definitions, then, when P changes, the total number of definitions will change. In a conventional definitive script, redefining P would cause a transition to a destination state differing from the original one only by the definition of P; at most one definition has been changed. Therefore, meta-definitions are not conventional definitions.

A meta-definition which involves an abstract parameter P can be viewed as an agent that effects a state transition when it observes a change in the definition of P. This means that a meta-definition is as much concerned with controlling transitions as with describing states.

7.2.2. Semi-evaluation

There is a semi-evaluation operator | ... | in the ARCA notation. This operator substitutes the current value of the expression within the two vertical bars to form a persistent definition. The semi-evaluation operator is useful in freezing the value of a variable. For example, in the definition

$$\text{sterling} = \text{foreign} \times |\text{rate}| - \text{charge}$$

the exchange rate can be fixed during a deal and the value of sterling depends on foreign and charge only. As indicated by this example, semi-evaluation can be a convenient way of assisting the construction of a state. It is particular useful within loopings where the current definition of the expression is difficult to obtain without consulting the transition history.

Definition involving the semi-evaluation operator can also be regarded as another form of meta-definition. This is because the definition stored is different from the definition typed in. A significant characteristic of this kind of definition is its context-sensitive nature – the meaning of the definition depends upon the current state. So semi-evaluation also acts like a simple agent to evaluate an expression in the current context, to modify the defining formula, and to effect a transition from the current state.

7.2.3. Inheritance

Since a definitive script does not necessarily specify a complete model, a definitive notation is very suitable for supporting inheritance. For example, we may want to extend DoNaLD so that it can define generic shapes (here, a generic shape does not

mean a class of shapes but a shape bearing certain properties that is generally not realisable due to the lack of some information). Listing 7.4 proposes one way in which we can specify how the properties of objects can be inherited from other objects. The based on syntax instructs the interpreter to copy all the definitions within the basic object into its own scope. So effectively, sqr is defined by the definitions in Listing 7.5.

```
openshape rectangle
within rectangle {
      real width, length
      real area = width * length
      point centre, NE, NW, SE, SW
      line N, E, S, W
      NE = centre + {width/2, length/2}
      NW = centre + {-width/2, length/2}
      SE = centre + {width/2, -length/2}
      SW = centre - {width/2, length/2}
      N = [NW, NE]
      E = [NE, SE]
      S = [SW, SE]
      W = [NW, SW]
}

openshape square based on rectangle
within square {
      real size
      width, length = size, size
}

openshape unitsquare based on square
within unitsquare {
      size = 1.0
}

openshape sqr based on unitsquare
within sqr {
      centre = {500, 500}
}
```

Listing 7.4: A Proposed DoNaLD Specification of Rectangular Objects

```
openshape sqr
within sqr {
	real width, length
	real area = width * length
	point centre, NE, NW, SE, SW
	line N, E, S, W
	NE = centre + {width/2, length/2}
	NW = centre + {-width/2, length/2}
	SE = centre + {width/2, -length/2}
	SW = centre - {width/2, length/2}
	N = [NW, NE]
	E = [NE, SE]
	S = [SW, SE]
	W = [NW, SW]
	real size
	width, length = size, size
	size = 1.0
	centre = {500, 500}
}
```

Listing 7.5: Specification of a Square

This scheme provides a dynamic inheritance method similar to the delegate technique in object-oriented programming [Lieberman86] – the properties of an object are inherited directly from another object instance rather than the properties of a class of objects are derived from another class of objects (cf.: "children inherit properties from their father" vs "babies inherit properties from adults").

At this stage, this inheritance scheme is only a tentative proposal for an extension to the DoNaLD notation. As object-oriented programming is becoming popular, inheritance is surely one of the issues that researches in any programming paradigm cannot overlook. Generic shapes such as the rectangular objects specified in Listing 7.4 are quite different in nature from traditional DoNaLD shapes. They are not generally realisable; they have to be interpreted with respect to observations of a different nature, viz observations of rectangular objects in general rather than any particular rectangular object.

The inheritance scheme we have suggested is a plausible attempt to relate definitive programming and object-oriented programming. This involves some extension of single-agent definitive programming. In particular, definition inheritance

can be regarded as another form of meta-definition. When the base template is changed, its dependent shapes must also be changed. Every generic shape specification can be interpreted as specifying an agent who monitors the base shape and redefines the derived shapes accordingly.

7.3. Input Management

A single-agent definitive system provides an interactive environment for program development. However, there is a distinction between an interactive environment for program development and an environment for developing general interactive programs. The notations and techniques discussed so far only enable us to interact through textual input of definitions; they do not enable us to develop interactive programs that make use of general mechanisms for user-input. To this end, we need to model input devices. Without means to model keyboard, mouse and time, we can hardly write programs with a reasonable user interface.

This section describes an extension to the Scout system so that mouse events and system generating events (such as a clock) can be interpreted. Two worked examples will be referred to in the course of discussion – a room viewer and a vehicle cruise control system. The DoNaLD specification of the room example was written by Edward Yung; the LSD specification of the vehicle cruise control system and its EDEN implementation were originally written by Ian Bridge. The graphical display and the mouse control were written on top of these two implementations using the extended Scout system. The interested reader may refer to Appendix F and G for complete detail.

7.3.1. The Room Example

A sample output of the room example is shown in Figure 7.1. The user may interact through the graphical interface in the following ways:

1. The menu options are self-explanatory.

2. The zoom window (the right-hand-side one) is partitioned into four regions – the four regions divided by the two diagonals; these four regions resembles the four menu options of the zoom position menus.

3. In the normal view (the window on the left), the table can be moved by direct manipulation. If the user presses a mouse button within the table area, drags the mouse and then releases it, the table will move by the same displacement of the mouse position.

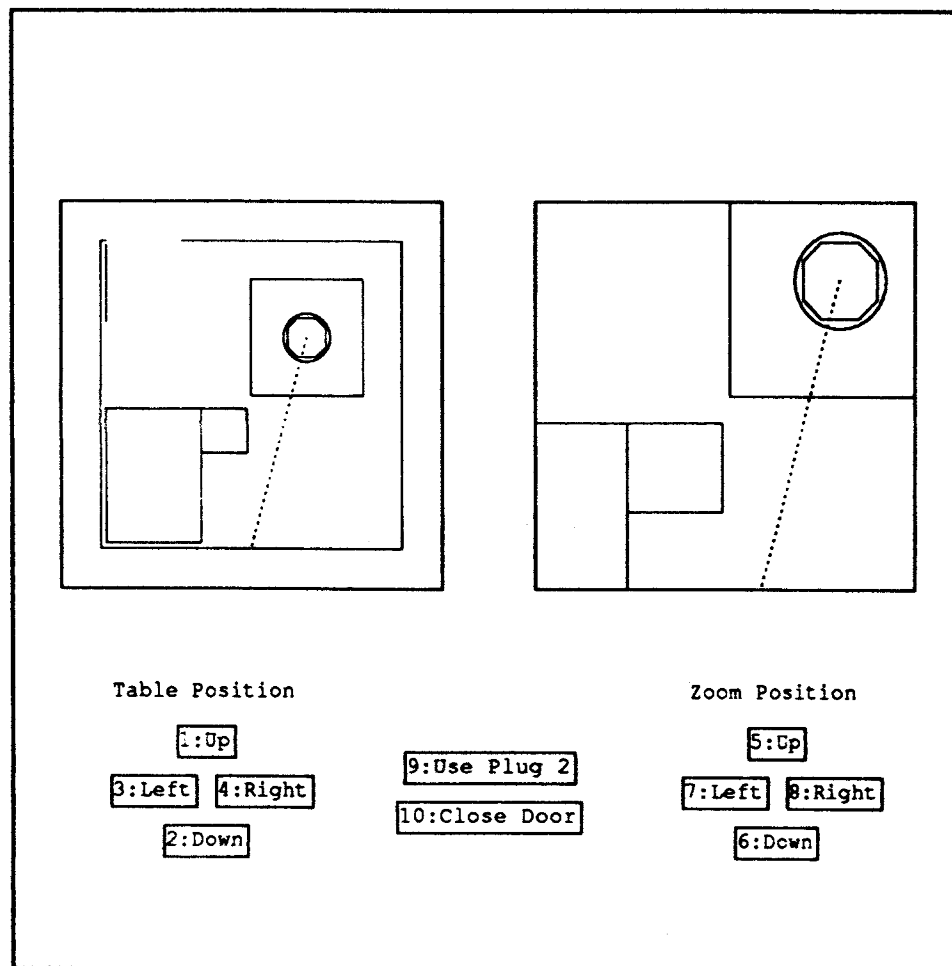

Figure 7.1: A Sample Output of the Room Viewer Example

7.3.2. The Vehicle Cruise Control Example

The vehicle cruise control example has been discussed in [BBY92]. The focus of [BBY92] is on an agent-oriented programming paradigm; the user-interface is discussed in the language of the LSD agent specification language (see next chapter). In this chapter, the EDEN implementations of some of the user-interfacing agents will be used to illustrate some techniques of input management in the Scout system.

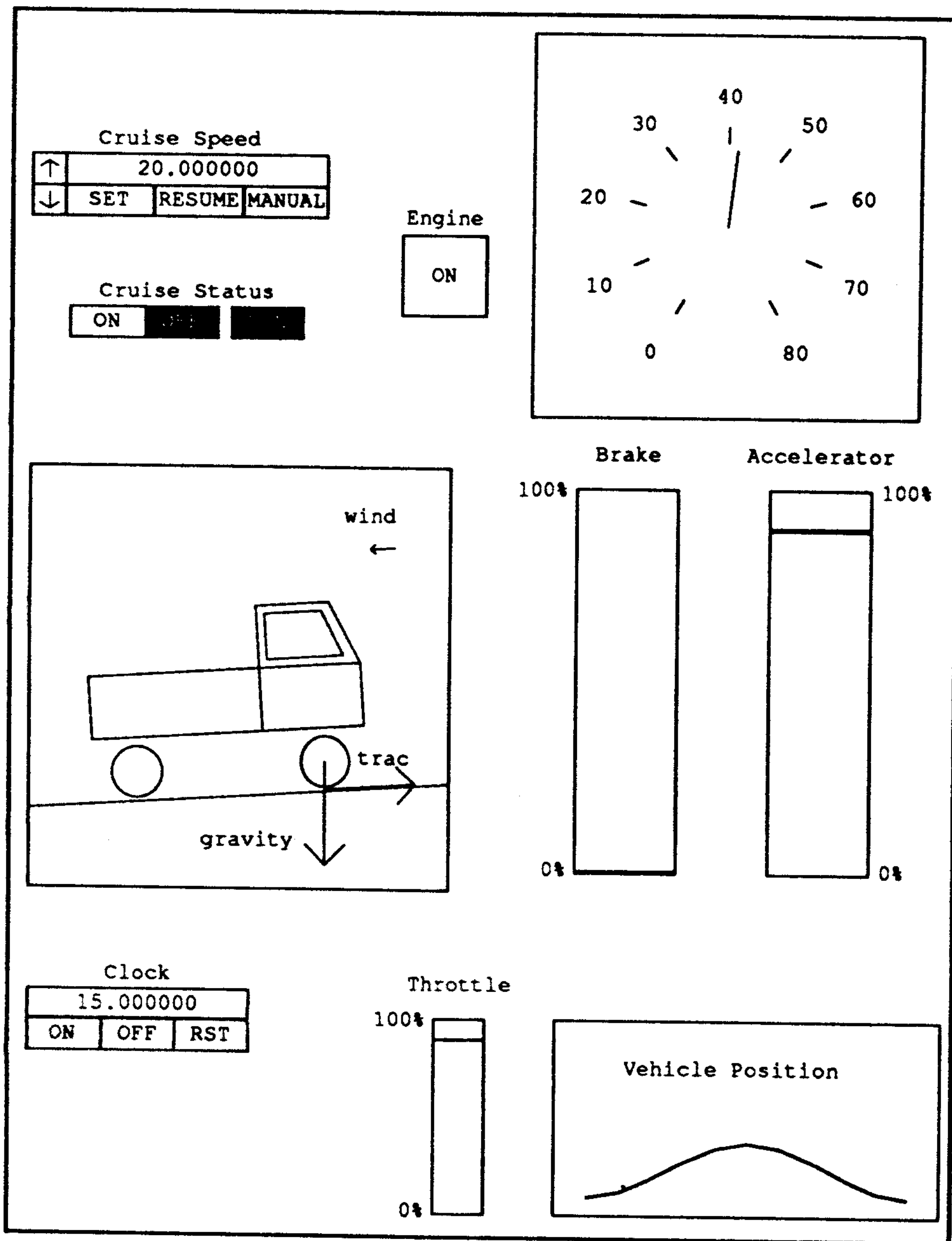

Figure 7.2: A Sample Output of the Vehicle Cruise Control Simulation

A sample output of the room example is shown in Figure 7.2. The user may interact through the graphical interface in the following ways:

1. Switch on or off the engine by pressing the ignition button. The ignition button is an example of toggle switch (see §7.3.4.2.).

2. Switch on or off the cruise controller by pressing the "ON" or "OFF" button on the cruise controller panel. The cruise controller switch is an example of radio buttons (see §7.3.4.4.).

3. Set and resume the cruise speed by pressing the "SET" and "RESUME" buttons respectively; switch to manual speed control by pressing the "MANUAL" button. The set of buttons – "SET", "RESUME" and "MANUAL" – also illustrates radio buttons.

4. Increase or decrease the cruise speed by pressing the buttons with the up arrow and the down arrow respectively. These two buttons are examples of duration-sensitive buttons (see §7.3.4.5.).

5. Change the position of the accelerator by pressing the mouse in the accelerator window. The nearer to the "100%" mark, the further the accelerator is depressed. The accelerator window is a variant of a menu button (see §7.3.4.3.).

6. Start, stop or reset the simulation clock by pressing the "ON", "OFF" and "RST" button in the clock panel respectively. They are implemented in the example as menu buttons (see §7.3.4.3.).

7.3.3. Extension to the Scout System

7.3.3.1. Considerations

The extension to the Scout system regarding input management takes into account the following points:

1) Limitations of the Original EDEN System

In the originally EDEN interpreter, the only user interaction is via typing in EDEN statements (including definitions). Upon received a definition or an action call initiated by the user, the interpreter will store the definition in its definition store and execute the action. All the actions triggered by the definition or invoked by the original action will then proceed. Not until all the triggered actions have been terminated will the system accept another statement. Under this scheme, there is no chance of processing any external input in the midst of a non-terminating loop. A problematic case is when a system clock has to be simulated. In a clocked system, such as the cruise control animation, there is no easy way to change the parameters, such as the accelerator position, while the clock is running[4].

2) Modes of Input

The aim of input is to initiate state transitions. In definitive programming, transitions are modelled by redefinitions. For this reason, we devise mechanisms to treat all modes of input as ordinary definitions. That is to say, a mouse pressing action, for example, will cause a variable to be (re)defined. Three modes of input are identified: user-generated events, e.g. mouse-pressed; system-generated events, e.g. clock updating; 'normal' channel of input, i.e. type-in EDEN statements.

3) Modes of Response

Ideally, we would like the system to be capable of different modes of response. For instance, an input may demand an immediate response (as in an interrupt, when the system suspends activities to service a user request) or may cause a change to the

---

[4] In the original EDEN cruise control simulation, as developed by Ian Bridge, the clock stops every 10 seconds to allow any possible change of definitions before the simulation continues (manually).

system without an immediate effect (as in polling activity, where the system monitors the values of input variables intermittently).

Redefinition of variables is a method of communicating state changes to the system that can be used for both interrupt and polling activities. Interrupt and polling activities correspond to different ways of associating actions with the redefinition of input variables. Since a definition may cause indivisible value changes and invoke EDEN actions in the same conceptual transition of state, implementing an input event as a definition can simulate the effect of an interrupt. In polling, actions are performed with reference to the current values of the input variables as and when appropriate.

4) Interrelationship between input management and Scout

It has been argued that separating input devices from application programs is inappropriate for modern user interfaces [Meads87]. It has also been argued that the Smalltalk "Model-View-Controller" (MVC) paradigm of an application may not take full advantage of the close relationship between output and input handler [Myers90]. In MVC, a program is separated into three parts: the *model* which embodies the application semantics, the *view* which handles the output graphics that show the model, and the *controller* which handles input and interaction. Unfortunately, programmers have found that the code for the controller and view are often tightly interlinked; creating a new view usually requires creating a corresponding new controller. In fact, both are often entwined with the model, so all three need to be rewritten.

In definitive programming, we also admit the close relationship between model, view and controller. A model is constructed by a definitive script; the realisation of the definitive script forms a display. The relationship between the model and the display is analogous to a mechanical linkage: a change to the model causes a simultaneous change to the display. Input handling also has a close relationship with both the display and the model. The meaning of pressing a button depends upon where the mouse is located; the input changes the model and in turn changes the display. In the Jugs

example mentioned in Chapter 5, the region inscribed in a menu window denotes the existence of a menu option. The button pressing action within that area should cause a change of the water levels in the jugs model and hence affect the jugs display. The information described in the Scout notation is useful for input management.

7.3.3.2. The Extension Plan

Based on the considerations above, the extension plan to the Scout system involves the following:

1. Since the meaning of an activity of an input device depends on the location of the pointer device, Scout window has a new attribute *sensitivity*. In the current design, this field is only a boolean value indicating whether input is accepted inside the window. Ideally, *sensitivity* may also be used to specify which kind of input is acceptable.

2. In view of 1), the EDEN/X Window interface EX has to be able to generate a definition upon an input action within a sensitive Scout window. In order to assist the interpretation of the input, the variable name to be defined must be related to the Scout window name. In our current implementation, the kinds of input EX manages are pressing and releasing of mouse button and key-press on keyboard. Mouse movement leads to such frequent generation of definitions that system performance becomes unacceptably slow, and mouse movement is not currently managed.

   The variable name to be defined is determined by which region the mouse is in. Consider a button pressing or releasing action. If the mouse is within a DoNaLD or ARCA window, the variable name would be the Scout window name concatenates with "_mouse"; if the mouse is within a text window, it would be the Scout

window name concatenated with "_mouse_" followed by the box number[5]. When the mouse action occurs, the value assigned to the appropriate variable records the nature and the location of the mouse action. Currently, the value is a 5-tuple of (*button*, *type*, *state*, *x*, *y*),

where *button* = the button number pressed or released;

type = the button action (4 = pressed, 5 = released);

state = the state before the button action occurred (shift (+1), caplock (+2), control (+4), meta (+8) and was pressed (+256)). For example, if a button is released while the shift and control keys are depressing, *state* will be 1 + 4 + 256 = 261;

x, *y* = the x- and y- coordinates of the mouse in the coordinate system of the window in which the mouse action occurred.

As with mouse events, a stroke on the keyboard will generate a definition. Instead of "_mouse" or "_mouse_" followed by a box number, the variable name will end with "_key" or "_key_" followed by a box number. The value defined will also be a 5-tuple: (*key*, *type*, *state*, *x*, *y*), where *key* is the ascii code of the key pressed.

As an example (taken from the vehicle cruise control simulation in Figure 7.2), consider the Scout window is defined as follows:

---

[5] A text window may consist more than one boxes whereas exactly one box constitutes a graphic window.

```
point     brakeOrg = {225, 250};
integer   BAwidth = 50;
integer   BAlength = 200;
window    brakePedal = {
          type:       DONALD,
          box:        [brakeOrg, brakeOrg + {BAwidth, BAlength}],
          pict:       "BRAKE",
          xmax:       100,
          ymax:       100,
          border:     1,
          sensitive:  ON
};
```

A mouse click (press and release) in this window will typically generate the following definitions:

brakePedal_mouse = [1, 4, 0, 50, 70];

brakePedal_mouse = [1, 5, 256, 50, 70];

3. The EDEN interpreter has to be able to manage definitions coming from different sources. The definitions generated by EX are sent to EDEN via a *message queue*, a UNIX system V inter-process communication method, whereas the type-in definitions come in through a pipeline. An EDEN action may also generate definitions (system-generated events) and send these to the EDEN interpreter via the message queue. This definition will then be processed in the next time slot[6] of the interpreter. The input management for the EDEN interpreter has to be modified so that input accepted from the pipeline and the message queue is interleaved.

---

[6] In a *time slot*, the EDEN interpreter will process an EDEN statement and all its consequent actions.

7.3.4. Input Handling Techniques

The following sub-sections describe how the definitions generated by input events can be combined with different patterns of EDEN actions to animate a timer and to implement different kinds of buttons.

7.3.4.1. Push Button

A push button is one which is logically *true* when it is pressed and is *false* when it is released. This is the simplest form of button. It can be animated by a definition as simple as:

```
ButtonStatus is PB_mouse[2] == 4;
```

This defines ButtonStatus to be true when a mouse button is pressed in the Scout PB window, and false otherwise.

7.3.4.2. Toggle Switch

A toggle switch is one which has an initial state. Every time a button is pressed, the state of the toggle switch reverses; releasing a button has no effect on the toggle switch's state. The previous state of a toggle switch has to be remembered and an initial state has to be defined. Typically, a toggle switch is animated as follows:

```
engineStts = esOn;
proc chgEngineStts : ignition_mouse_1 {
        if (ignition_mouse_1[2] == 4) {
                engineStts = (engineStts == esOn) ? esOff : esOn;
        }
}
```

In this example, the engine status (engineStts) is initially esOn. The engine status will alternatively change to esOff and esOn whenever a button is pressed in the ignition window.

7.3.4.3. Menu Buttons

A menu button is a button that invokes an action when it is pressed (or released depending on the design). It is like a door bell which will start a melody when the button is pressed; the melody will continue even if the button is released (releasing the button has no effect on the door bell). This kind of button is often used in menu selection. For example in the room viewer example (in Figure 7.1), pressing the "zoom up" menu option (the zoomUp window) once will move the zooming area up by 100 units. The implementation of the "zoom up" option is as follow:

```
proc zoomUp_action : zoomUp_mouse_1 {
        if (zoomUp_mouse_1[2] == 4) {
                zoomPos = pt_add7(zoomPos, [100, 0]);
        }
}
```

7.3.4.4. Radio Buttons

Radio buttons are defined by a set of buttons amongst which exactly one of them is depressed at any time. The pressing of one button will cause another button which is currently selected to be released. The following shows an example of a set of three radio buttons (RB1, RB2 and RB3) with the initial condition of button RB1 is on. This scheme requires the knowledge of the current values of the buttons but minimises the updating of variables[8].

---

[7] pt_add() performs a vector sum of the two argument points.

[8] Alternatively we can define
proc update_buttons1 : rb1_mouse { if (rb1_mouse[2] == 4) { RB1 = 1; RB2 = 0; RB3 = 0; } }
and so on. The method shown in the main text is preferred because it minimises the number of variables to be redefined.

```
RB1 = 1; RB2 = 0; RB3 = 0;

proc set_RB1 : rb1_mouse { if (rb1_mouse[2] == 4 && !RB1) RB1 = 1; }
proc unset_RB1 : RB2, RB3 { if ((RB2 || RB3) && RB1) { RB1 = 0; } }

proc set_RB2 : rb2_mouse { if (rb2_mouse[2] == 4 && !RB2) RB2 = 1; }
proc unset_RB2 : RB1, RB3 { if ((RB1 || RB3) && RB2) RB2 = 0; } }

proc set_RB3 : rb3_mouse { if (rb3_mouse[2] == 4 && !RB3) RB3 = 1; }
proc unset_RB3 : RB1, RB2 { if ((RB1 || RB2) && RB3) RB3 = 0; }
```

7.3.4.5. Duration-Sensitive Button

A duration-sensitive button is essentially a push button. The reason for putting the duration-sensitive button in a separate category is that its duration of pressing, rather than its logical state, is significant. The implementation of a duration-sensitive button is therefore different from that of a push button.

```
incrBtn = pbUp;
proc incCrSpeed : incrBtn, crUpWin_mouse {
	if (crUpWin_mouse[2] == 4) {	/* button pressed */
		SendToEden("incrBtn = pbDown;\n");
		if ((cruiseStts != csOff)  && (cruiseSpeed_mph < maxCruiseSpeed_mph))
			cruiseSpeed_mph = cruiseSpeed_mph + 1;
	} else { /* button released */
		if (incrBtn == pbDown) incrBtn = pbUp;
	}
}
```

In the above example (cf Figure 7.2), when a mouse button is pressed in the crUpWin window ("increase cruise speed" button), the setting for the cruise speed will be increased by 1 mph repeatedly so long as the button remains in a down position. SendToEden() is an EDEN function which will send the argument string to the EDEN interpreter itself via the message queue. Since the definition denoted by the argument string redefines incrBtn, a triggering variable of the incCrSpeed action itself, the variable cruiseSpeed_mph will keep on increasing in every time slot until the button is released.

7.3.4.6. Clocking

The technique used to implement the duration sensitive buttons is also applicable to the simulation of a system clock. The following shows how a chime may be implemented. This chime will print a line "Bell!" every 5 seconds.

```
chime = 0;
proc clock_watcher : clock {
        if (clock – clock_init) >= 5) {
                clock_init = clock;
                chime++;
        }
        SendToEden("clock = "//str(time())//";\n");9 /* update clock */
}
proc bell : chime { if (chime) writeln("Bell!"); }
clock = clock_init = time(); /* set clock to current time; start the clock */
```

The above techniques are sufficient for most, but not all, kinds of interaction in the room viewer and cruise control examples. These techniques only use the button pressing or releasing status. Other information, such as the position of the mouse in the window, is not used. There are cases in the room viewer and cruise control examples where the positional information is used. For example, movement of the table in the room (see Figure 7.1) is related to the displacement between the mouse button pressing and button releasing positions; the position of the mouse when pressing a button in the accelerator window (see Figure 7.2) determines how far the accelerator is depressed.

There are many techniques for interaction that are currently in use or are desirable. The two worked examples illustrate basic principles upon which a large

---

[9] SendToEden() makes use of message queue (one of the System V IPC methods) to send a definition to EDEN itself rather than directly executing the definition. This is because direct execution will block the execution of other definitions and actions whereas messages will be processed later when other actions are done.

number of techniques can be built up, and demonstrate that definitive programming can easily produce elegant user-interfaces.

7.4. Summary and Conclusion

There are two possible directions in which to generalise single-agent definitive systems. One is to develop more complex underlying algebras; the other is to introduce agents. This chapter starts by exploring the first possibility. We have investigated the Admira prototype. An Admira script can be interpreted as part pure definitive script and part functional specification of the algebra underneath the definitive script. Admira shows that, by having an appropriate underlying algebra, a definitive script can describe a complex state which encapsulates many states and transitions. However, when we increase the computational power of the operators in the underlying algebra in this way, we sacrifice clarity in state-based interpretation of the associated scripts. For example, Miranda has complex operators, but a Miranda script has an obscure state-based interpretation. In contrast, DoNaLD has simple operators but a DoNaLD script has a clear state-based interpretation.

In the Definitive State-Transition model, the state description should be determined by the mode of observation. This allows us to gain maximum understanding of the real-world system. Definitive programming focuses on the description of the relationship between observable properties of the real world; developing powerful underlying algebra is of secondary importance. This means that developing powerful underlying algebra is no substitute for the introduction of agents.

Some existing and proposed features of definitive notations like loops, semi-evaluation and inheritance are agent-like. In particular, we can view EDEN actions used for input management as programmable agents. Agents for governing the state transitions are important, but have to be used in a disciplined way. Improper use of EDEN actions, for example, can make the difference between principled definitive

programming and an anarchic form of procedural programming. In the next chapter, we shall discuss an agent-oriented definitive system formally.