2 Principles of MWDS

This chapter has two parts. The first describes the distinctive characteristics of a definitive script and reviews the computer tools that have been developed at Warwick to support MWDS. The second discusses the principles of MWDS in single-agent and multi-agent scenarios and the use of MWDS as a framework for the modeller's construal of an external situation. The chapter also introduces the concept of an agent-oriented LSD account and its supportive role in MWDS.

2.1 What is a definitive script?

A definitive script consists of a set of *definitions*. Throughout this thesis, the word 'definition' is used in a special sense to refer to a unidirectional functional dependency relation, and the word 'definitive' is to be interpreted as 'definition-based'. A set of dependencies represented in a definitive script is normally acyclic (cf. Figure 1-2). A typical definition takes the form:

$$y \text{ is}^{1} f(x_1, x_2, \ldots, x_n)$$

where y, x_1, x_2, …, x_n are variables, f is a formula involving some algebraic operators and the operands are the variables x_1, x_2, …, x_n or constants. A definition is interpreted as associating the current value of the expression on the right-hand-side (RHS) of the "is" ($f(x_1, x_2, \ldots, x_n)$) with the left-hand-side (LHS) variable (y). The association between values is indivisible in change. The term 'indivisible' is used in this context to convey the idea that a change to the value of a variable of the RHS affects the value of a variable on the LHS in a fashion that permits no interruption. This is similar to the way that a spreadsheet maintains the value in each cell. In fact, if we remove the tabular interface from a spreadsheet, the cell names together with the formulae that define the contents of cells (e.g. *A1 = A2 + A3*) can be regarded as a definitive script.

In MWDS, the variables on the LHS of a definitive script correspond to observables in the external world referent[2]. As explained in Section 1.6, the association between variables and observables is established through the interaction with the model. This association is mediated experientially rather than abstractly specified. This motivates the introduction of variables in definitive scripts that represent observables attached to the computer model rather than the referent. Such observables might include graphical elements, window displays and animations as featured on a computer screen. It is for this reason that a definitive script is to be viewed as representing the state of a computer-based artefact rather than (say) a collection of predicates.

The term 'definitive notation' was first introduced in 1986 by Beynon to refer to the syntactic forms used to describe the different kinds of variables and operators that can be used in formulating definitive scripts. Each definitive notation is associated with particular kinds of variable that can appear on the LHS and RHS of definitions. Each has its own underlying algebra consisting of a family of data values and operators on these data values. Many different definitive notations have been developed at Warwick to serve the representation needs of various kinds of referent (cf. Section 2.1.2). These include definitive notations based on scalars, strings and lists as well as definitive notations to relate data elements on the screen display. For instance, there is a simple definitive notation for line drawing where the data values are points, lines and circles, and the operators on such data values include rotation and translation. Because of the way in which a definitive script is interpreted, the data values and operators featured in a definitive notation must have an experiential significance. For abstract data values such as scalars, strings and lists, this significance stems from the modeller's familiarity with interpreting symbolic data (e.g. the integer 3 or the string "hello") and structure (as e.g. in the database records {1, 'Ann', 165}, {2, 'Betty', 155}, {3, 'Catherine', 172} in a table which records student's height).

To be able to explore characteristics and features of MWDS, it is necessary to understand the fundamental concept of 'definition' and 'dependency', which underlies the use of a definitive script.

---

[1] In practice, e.g. in spreadsheets, the symbol *equals* (=) can be used to denote a definition

[2] The object or situation that is being modelled, whether already existing or only imagined [Russ97]

2.1.1 The characteristics of definition and dependency

In a definition, the relationship between values is a one-way effect – if the value of the RHS is changed, the value on the LHS changes, but not vice versa. The values and operands that occur in a definition can be of many different types, depending on the application. In addition, there are essentially two different kinds of definition: implicit and explicit.

- Implicit definition

A definition is *implicit* if its RHS is a formula that refers to one or more variables. Such a definition establishes dependencies: changing the value of variable(s) on the RHS will automatically affect the value of variable on the LHS. For instance, the following script consists entirely of implicit definitions:

```
1. H is (L+M)*N;
2. M is (J>=0)?2: 4;
3. J is K;
```

In the implicit definition of the form: *y is* $f(x_1,x_2, \ldots ,x_n)$, the value of the variable *y* indivisibly depends on the current value of the expression $f(x_1,x_2, \ldots ,x_n)$. Whenever the value of one of the variables x_i changes or the formula *f* is changed, the value of the variable *y* will be automatically updated to the value that results from the evaluation of the expression $f(x_1,x_2, \ldots ,x_n)$.

- Explicit definition

A definition is *explicit* if its RHS consists of an actual value. The value of the LHS is explicitly given and no dependency is involved. For instance, the following script consists entirely of explicit definitions:

```
1. A is 6;
2. B is “hello”;
3. C is 2.34;
4. D is [12, 29];
```

Because scripts can contain implicit definitions, the current value of a variable can be undefined. Sometimes it is useful to explicitly assign an undefined value to a variable. For this purpose, the symbol ‘@’ is used to denote ‘undefined’.

Explicit definition has significantly different semantics from a conventional assignment in that a change to the actual value of a variable can cause a change to the values of other variables

that depend upon it. For instance, in the following scripts, the assignment to A (at line 7) affects the value of the variable sum:

```
5.  sum is A + 10;
6.  writeln(sum);
    16
7.  A is –10;
8.  writeln(sum);
    0
```

The definitions of A at lines 1 and 7 above illustrate the two possible interpretations of a definition. The first definition of A is part of a definitive script, the *set* of definitions at lines 1 through 4, that is viewed as a representation of an external state. The value of A itself represents the value of an external observable. The second definition of A is part of the *sequence* of definitions at lines 1 through 5 and 7, that can be viewed as a redefinition that represents a transition between external states. It can also be viewed as part of a new definitive script, the set of definitions at lines 2 through 5 and 7. In this way, a sequence of definitions can be interpreted as specifying a sequence of definitive scripts, in general subject to avoiding cyclic definitions. The terms 'definition' and 'redefinition' will be used to refer respectively to the two possible interpretations of a definition, as part of a definitive script and as part of a sequence of definitions.

In MWDS, definitive scripts and sequences of definitions respectively represent states and transitions in the external world (cf. Figure 1-5). The interpretation of state is as depicted in Figure 2-1(a): points correspond to definitive scripts and arrows to (possible) redefinitions. Where two redefinitions can be performed in either order so as to achieve the same transition, as depicted in Figure 2-1(b), it is also possible to interpret their execution in parallel as a single atomic transition. In this way, the model enables us to distinguish between performing two redefinitions in either order, or 'at the same time' and this can be seen as a form of true concurrency [LMRT90].

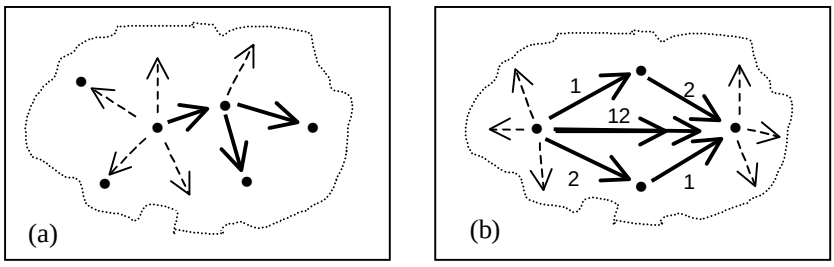


Figure 2-1: A transition of state in definitive scripts

To comprehend how the redefinition of one variable will affect others it is necessary to have an overall picture of the dependencies between all variables. This can be supplied by a dependency network diagram (cf. Figure 1-2). For instance, consider the following sequence of definitions:

1. H is (L+M)*N;
2. L is 9;
3. N is 27;
4. M is (J>=0)?2: 4;
5. J is K;
6. K is 14;
7. H is (J*L) + N;

Figure 2-2(a) gives an overall picture of the dependencies between variables established by the definitions of H, M and J at lines 1 through 6. It also displays hierarchical dependencies between variables, which helps in comprehending the order in which the values of these variables get updated in dependency maintenance. For instance, as illustrated in Figure 2-2(a), the definition of K at line 6 will cause updating of the values of J, then M and then H in sequence. A redefinition can also affect the dependency network diagram. For instance, the dependency network diagram shown in Figure 2-2(a) is transformed into that in Figure 2-2(b) after the redefinition of H at line 7.

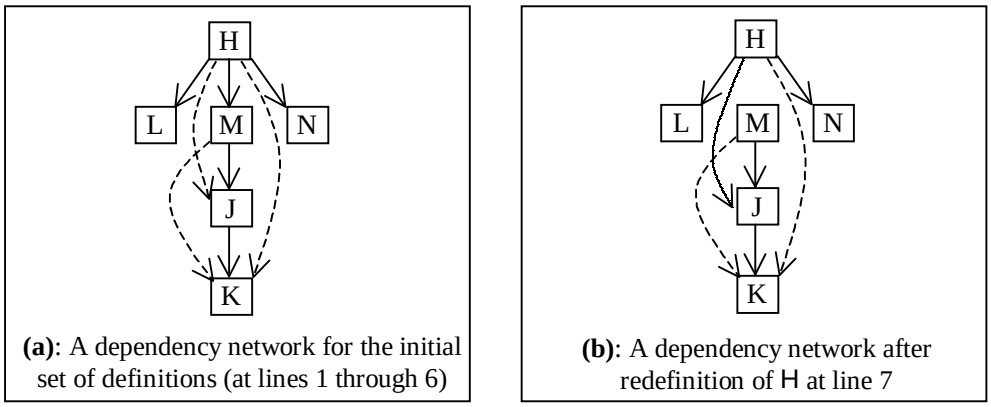
A single definition 'directly' establishes a *direct dependency* between the variable on the LHS and the variables on the RHS (as indicated by the solid arrows in Figure 2-2). *Indirect dependencies* are 'indirectly' established by transitivity through chains of definitions (as indicated by the dashed arrows in Figure 2-2). Both solid and dashed arrows indicate that the variable at the beginning of the arrow *depends* on the variable at the other end.

The open-development dependency concept (cf. Figure 1-2), which underlies the fundamental structure of a definitive script, is significant throughout this thesis. To conveniently explain the dependency between variables, the terms 'dependee'[3] and 'depender'[4] are introduced. A dependee is defined as a variable, which depends on others (e.g. in Figure 2-2 H is a dependee of L) while a depender is defined as a variable on which others are dependent (e.g. L is a depender of H). The concepts of 'direct' dependee/ depender and 'indirect' dependee/ depender can be derived in a similar way to direct and indirect dependency. For instance, in Figure 2-2(a)

[3] Your dependee can be interpreted as "who depends on you"

[4] Your depender can be interpreted as "on whom do you depend"

H is a direct dependee of L, L is a direct depender of H, J is an indirect depender of H via M and H is an indirect dependee of J via M.



(a): A dependency network for the initial set of definitions (at lines 1 through 6)

(b): A dependency network after redefinition of H at line 7

Figure 2-2: The dependency network diagrams

The management of dependency by means of definitive scripts and redefinitions will be referred to as the application of 'definitive principles'. Definitive principles relate both to abstract analysis and to practical model building. The central abstraction is that changing a value of a depender will result in updating the value of all of its dependees (both direct and indirect). To update the values of these variables in practice, a 'dependency maintenance system' is required. Several prototype tools that feature dependency maintenance systems, such as ttyeden and tkeden, have been developed under the EM project. Various definitive notations such as Eden, DoNaLD and Scout have also been implemented along with these tools. At this point, it is necessary to give a brief review of such systems and notations, which serve as the main means to explore and exploit MWDS.

2.1.2 Review of computer support tools

This section gives an overview of the tools and notations that have been developed at Warwick to support modelling with definitive scripts. It discusses the essential features and usages that are needed to understand the illustrative models throughout the thesis. To support MWDS, the implementation of a dependency maintenance system to maintain dependencies and update the values of dependees once the value of a depender is changed is necessary. The system should contain basic features (1) that allow the modeller to define functions and triggered actions, (2) that maintain and update dependencies, and (3) that provide an interactive environment to support open-development modelling. Where (1) is concerned:

- a **function** is a user-defined operator, normally with no side-effect on state, for use in definitions:

- a **triggered action** is a procedure triggered by variables (active values[5]), for instance, to invoke a redefinition or execute a procedure. Actions may be used to synchronise abstract and visual models (e.g. via procedural update of a display), or to simulate agent actions.

Where (2) is concerned, basic dependency maintenance involves scheduling the re-evaluations of variables in definitive scripts. Where (3) is concerned, actions are scheduled in such a way that the user can interact in a 'concurrent' fashion i.e. to intervene in computer activity except where this is indivisible. This gives a special status to computing activity associated with the management of definition, and so can be regarded as 'definition-driven' programming[6].

Many tools that have these basic features have been developed under the EM project. However, throughout this thesis, three Eden tools (ttyeden, tkeden and dtkeden) are mainly used. Several definitive notations have also been developed along with these tools. The following subsections give a short account of the historical development and the characteristics of these three Eden tools. This includes a brief explanation of the syntax and features of several definitive notations that these three Eden tools support.

Dependency maintenance systems

This subsection reviews the specific features of the three Eden tools mentioned above that relate to dependency maintenance in MWDS.

- **Ttyeden interpreter**

The Eden definitive notation and the Eden evaluator *ttyeden* were designed and developed by Edward Yung in 1990 [Yung90]. Ttyeden was the first dependency maintenance system to be developed at Warwick. It has a basic text-based input and output environment that supports interactive user-input. This allows users to write scripts and observe the effect of redefinitions in a stimulus-response way. Ttyeden provides two built-in query features that help users to interact with scripts. One type of query takes the form

?*x*;

[5] Active value (in AI) is a means of associating procedures with a data value so that the procedures will be called when the data is accessed or written [http://www.harcourt.com/dictionary].

[6] A definition causes some change of state, through definitions and actions, until a stable state and then accepting another definitions [Yun96, p.47]

where *x* is a variable name. It returns the defining expression of *x* together with the list of direct dependees of *x* (i.e. the variables that depend on *x* directly). The other type of query takes the form

writeln(*x*);

and returns the current value of variable *x*. The following listing (where script.e consists of the definitions at lines 1 through 4) illustrates the use of these two commands.

```
1.  a is b+c;
2.  k is b*a;
3.  b = 5;
4.  c = 20;
5.  ?b;
    b=5
    b ~> [a, k];
6.  writeln(a);
    25;
7.  ?a;
    a is b+c;
    a ~> [k];
```

Listing 2-1: Ttyeden input and output for the file script.e

- **Tkeden**

The *tkeden* tool is an extension of the Eden evaluator ttyeden. Tkeden was designed by Yung [Yung92] in 1992, and uses a tcl/tk library for windowing and graphical drawing. It does not only support Eden notation, but also other definitive notations such as DoNaLD (for 2D line-drawing) and Scout (for screen display). This enables the tool to support simple graphics and windowing interfaces.

Tkeden has a GUI interface, which provides a user-friendly environment to facilitate MWDS activity. As illustrated in Figure 2-3, users can input sequences of definitions, query and redefine definitions through the 'input window' and view the results of their queries through the 'output window'. The 'history window' keeps a record of the sequences of definitions and queries that have been input by the modeller. By inspecting sequences of definitions and retrieving definitions that have been recorded in the history window, users can easily restore the previous state of the model. Users can also inspect the current status of the Eden definitions stored inside the system through the 'Eden script window'. The features of tkeden that have been illustrated here with reference to the Eden script script.e, also apply to the other definitive notations (as represented by the radio buttons on the Input window in Figure 2-3) that tkeden

supports. In particular, there are counterparts for the Eden script window to display the current DoNaLD and Scout definitions.

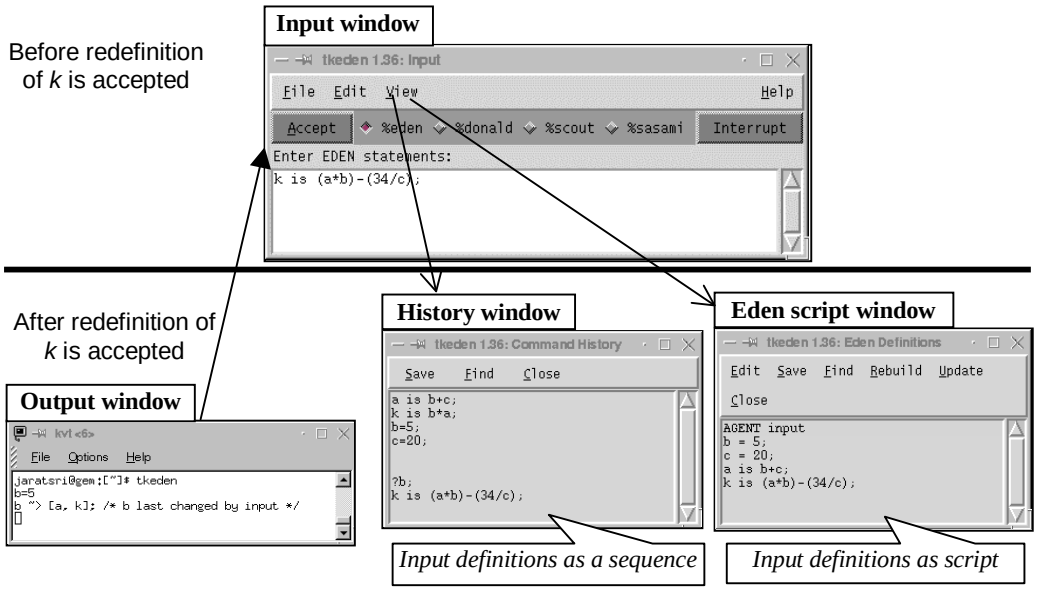


Figure 2-3: MWDS in tkeden

- **Dtkeden**

Dtkeden, developed by Sun [Sun99], is an extension of tkeden that provides a distributed script feature. Within the dtkeden environment, scripts can be passed from one machine to others via different modes of distributing scripts, such as 'Normal', 'Interference', 'Broadcast' and 'Private' mode. Dtkeden inherits all the features of tkeden and has several additional features. One additional feature that has been widely used in many models discussed in this thesis is 'virtual agency'. The virtual agent concept is used to automatically generate a large number of similar definitions.

So far several hundred models have been implemented under these three systems and based on definitive notations that they support – of these, many have been preserved, and in some cases reused. About a dozen of these models are explicitly discussed in this thesis – for convenience, brief descriptions of these models are given in an Appendix to the thesis. The models are also available together with tutorials and additional documentation in the form of a webpage (cf. [ModelWeb]). Eden, DoNaLD and Scout are the three main definitive notations in which most of the models are written. The next subsection gives a brief introduction to the design and implementation of the principal definitive notations used in this thesis.

Definitive notations and their evaluation

So far, we have emphasised the role of the three Eden tools as dependency maintenance system for MWDS. In practice, many other features are needed to give full support for MWDS and more general model-building using definitive principles. The syntax and essential features of several definitive notations developed at Warwick to represent various observations and perspectives on computers are discussed in this section together with the principal programming constructs used to support their evaluation in Eden. The notations considered include the first definitive notation ARCA for displaying and manipulating combinatorial diagrams, Eden for general-purpose modelling, DoNaLD for 2D line-drawing, Scout for window display, Eddi for database modelling and Sasami for geometric modelling.

- **Eden** – an Evaluator for DEfinitive Notations

Eden was first designed and developed by Yung [YY88] in 1987 as a general-purpose definitive notation and interpreter. It was initially developed with the implementation of definitive notations in mind. The Eden interpreter provides a 'hybrid' programming tool that allows definitive and procedural paradigms to be combined.

The Eden syntax and data types are similar to those in C. The basic Eden programming constructs are **`for`**, **`while`** and **`if`**, and the main types for variables are **`float`**, **`integer`**, **`string`** and **`list`**. Eden lists can be nested and non-homogeneous in type. In other words it is not necessary that all variables in the list have the same type. Moreover Eden has dynamic typing. Eden variables do not need to be declared before they are defined; they are dynamically typed according to the type of actual values that are assigned to them through definition. Variables can be defined implicitly by a formula:

e.g. *v is f(a, b, c)*

or defined explicitly to the current value of an expression:

e.g. *v = f(a, b, c)*

There are three abstract programming features in Eden: definitions, functions and actions.

Definitions are specified and implemented according to the principles described in Section 2.1.1. The Eden interpreter maintains the values of definitive variables automatically, and records all the dependency associated with a definitive script.

Functions are used to introduce user-defined operators into definitions. The way in which Eden functions are specified is similar to that used in a procedural program. For instance:

```
func sum {
        para lst;
        auto i, result;
                result=0;
                for(i=1;i<=lst#;i++)
                    result += lst[i];
                return result;
}
```

specifies a list summing function that can appear on the RHS of a definition e.g.

```
total is   sum(data);.
```

Actions are 'triggered procedures' which can be specified via:

```
proc proc_name: <triggering variables(s) as comma separated list>{
        <redefinitions etc to be performed when one or more of the triggering variables is
                                        touched>
}
```

The procedure is invoked to perform its action whenever any one of its triggering variables is redefined or re-evaluated, whether or not the value of these triggering variables is changed.

Apart from the above features, Eden provides some built-in functions that support more advanced modelling techniques. These include, for instance, techniques to delay dependency update, to queue actions, to deal with higher-order dependency and to generate scripts dynamically:

- autocalc :- is a predefined boolean variable that switches the mechanism of automatically triggering actions and updating definitions on and off;
- eager() :- is a procedure to invoke the immediate execution of queued definitions and actions;
- execute("script") :- is a function that turns a "script" (type string) to an executable script and execute it;
- `"string"` :- this will turn a string into a variable name.

- **DoNaLD** – a Definitive Notation for 2D Line-Drawing

The DoNaLD notation is a definitive notation for 2D line drawing. It is designed to support interactive graphics within the framework of a general-purpose programming paradigm based upon definitions [Bey89b]. As a definitive notation [Bey85, BABH86], DoNaLD is based upon an underlying algebra comprising values of the basic data types: **`real`**, **`integer`**, **`point`**, **`line`** and **`shape`**, and numerous operators for combining values of these different types. Integer and real are scalar values. A **`point`** in the plane is represented by a pair of scalar values {x, y}

or can be treated as a position vector. A **`line`** is a line segment that joins two points. A **`shape`** is a line drawing as represented by a union of **`points`**, **`lines`** and **`sub-drawings`**. There are also some special predefined types such as **`arc`**, **`circle`**, **`ellipse`**, **`char`**, **`boolean`** and **`label`**. Unlike Eden, DoNaLD is a strongly typed notation. DoNaLD variables need to be declared before they are defined. DoNaLD is implemented based on the Eden evaluator. All DoNaLD variables are translated into Eden variables so that each has a counterpart Eden variable. In a DoNaLD script the equals sign ('=') is used (in place of 'is') to denote definition. The discussion of how a DoNaLD script is translated into an Eden script will follow the discussion of the Scout notation below.

- **Scout -** a definitive notation for describing SCreen layOUT

The Scout notation is designed to support the displaying of the contents and the laying out of windows on screen. Its purpose is to present the definitive state in a user-specified manner and to supplement the displayed information for other definitive notations. The notation was introduced and implemented by Simon Yung in 1992 [Yung92].

Scout makes provision for presenting data by using definitions to describe the output formats of a variable. With definitions, a persistent link between the internal model and its external representation is achieved. Moreover, the observed changes of variables can be synchronised with internal state changes. Scout allows flexible control over output format [Yung92] based on definitive principles. Like DoNaLD, Scout is implemented based on the Eden evaluator. It is a strongly typed notation. In a Scout script, each variable has a counterpart representation in the translation to Eden. The basic data types in Scout, such as **`windows`**, **`frames`**[7], **`boxes`**[8], **`points`**, and **`strings`**, are related to designing and manipulating the windows displayed on the screen.

The screen in Scout is an imaginary, rather than a physical screen. A mapping from the imaginary to the physical screen is done through definition. A Scout script specifies a single screen that can contain many windows. Windows can be set to a 'sensitive' mode in order to be used in a user-interface (e.g. as interactive icons and buttons). A window can display text, an image or a line drawing from ARCA and DoNaLD.

---

[7] A frame is a screen area associated with a list of boxes

The integration between these three notations (Eden, DoNaLD and Scout)

As mentioned before, DoNaLD and Scout are implemented using the Eden evaluator. Both DoNaLD and Scout scripts are, in fact, translated into Eden scripts before being executed. The integration between these three notations inside (d)tkeden is illustrated in Figure 2-4.

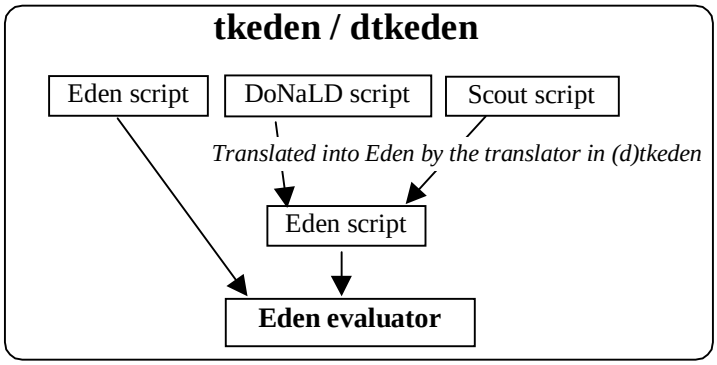


Figure 2-4: The integration between Eden, DoNaLD and Scout in (d)tkeden

As mentioned previously, DoNaLD and Scout variables need to be translated into Eden scripts. The translation conventions are summarised in the following tables.

1. The pattern for Scout and DoNaLD variables translated into Eden

	Example script	**Script translated into Eden**
DoNaLD	table	_table
	table/drawer	_table_drawer
Scout	str1	str1

2. The Eden equivalents for Scout and DoNaLD definitions and valu es

	Example script	**Eden translated into Eden**
DoNaLD	int s1=10	_s1 is 10;
	char k="hello"	_k is "hello";
Scout	string t1="good";	t1 is "good";
	integer re1=2;	re1 is 2.0;

[8] A box is a rectangular block of pixels with specified the top left and bottom right corners. For instance, the

3. The Eden translation of special data types (e.g. **`line`**, **`point`**, **`label`** and **`box`**)

	Example script	Eden translated into Eden

DoNaLD	point k1={0,10}	_k1 is cart(0,10);
		proc P_k1: _k1, A_k1, DoNaLD{
		  plot_point(DoNaLD,&_k1, &A_k1);
		};

	line l1=[k1,{0,20}]	_l1 is line(_k1, cart(0,20));
		proc P_l1: _l1, A_l1, DoNaLD{
		  plot_line(DoNaLD,&_l1, &A_l1);
		};

Scout	point p1={0,0};	p1 is [0,0];
	box bx1=[p1,{20,40}];	bx1 is formbox(p1,[20,40]);

- **ARCA** – a notation for displaying geometric diagrams

ARCA is one of the early definitive notations, developed by Beynon in 1983 [Bey83]. It was originally designed to support the display and manipulation of geometric diagrams with particular emphasis on combinatorial graphs with coloured and directed edges. It was invented as a medium to be used (possibly in conjunction with automated techniques) for constructing computer representations of connected graphs such as 'Cayley diagrams'[9] and uses definitive scripts to express the relationships between nodes, edges and modes. ARCA includes an auxiliary definitive notation, used to declare variables of complex type and to specify their mode of definition. An ARCA script is more abstract than a DoNaLD script in nature. An example of an ARCA script will be considered in the Lines model (cf. {Lines91} in Appendix B) to be discussed in Chapter 3.

The DoNaLD and ARCA notations are complementary. They correspond to different ways of observing geometric structures. ARCA [Bey86] defines the abstract connectivity of a graph (in terms of nodes, edges and vertexes), while DoNaLD scripts explicitly define geometrical elements such as lines or points.

- **Eddi** – a notation for database system based on dependency

Eddi is a definitive notation which integrates the concept of dependency with a relational database model in a similar way to Todd's ISBL [Todd76] discussed in Chapter 1 (cf. Section 1.4). The data type in Eddi is the relational table and the operators are the primitive relational

---

definition box1 = [p1, p2] specifies a box with p1 at the top left corner and p2 at the bottom right corner.

[9] "Groups and their graphs [GM65]"

algebra operators (cf. Section 2.1.3). Eddi supports both definition and assignment of relational variables. Definitions in Eddi correspond to relational database views. The implementation of Eddi is based on the Eden evaluator, and Eddi scripts are translated into Eden for interpretation. An example of an Eddi script is discussed in Section 2.1.3.

- **Sasami** – a notation for geometric modelling

Sasami was designed and implemented by Ben Carter [Carter99]. It is a definitive notation for modelling with boundary representations of geometry based on the OpenGL graphics API. The basic data types in Sasami are the vertex and the polygon. Dependencies in Sasami can connect geometric characteristics, scalar information, colour, texture and lighting attributes. Its implementation is based on the Eden evaluator, and Sasami scripts are translated into Eden for interpretation. A screenshot of a Sasami model is given in Appendix B (cf. Figure B-2).

Extensions to (d)tkeden

An important recent development has been the introduction of a parser generator, written in Eden, by Chris Brown [Brown00]. This parser generator allows observation-oriented parsers to be interactively specified and modified within the (d)tkeden environment. So far several notations have been implemented based on the Eden evaluator and then included into the (d)tkeden tool. For instance, the latest versions of the Eden tools allow the Eddi definitive notation to be incorporated into the interpreter by including a specification of the Eddi parser (cf. the procedure call installeddi() in Figure 2-8). Variants of other programming languages that are not definitive notations, such as SQL and LOGO, have also been implemented in this way.

2.1.3 Illustrative examples of using definitive scripts

To complement the above review of the tools and notations, some models developed using them will now be discussed. These illustrate how we can use definitive scripts in a diverse range of applications, and particularly in those applications – spreadsheets, geometric modelling and database – discussed in Chapter 1.

- The role of definitive scripts in general modelling

A spreadsheet is one of the most successful dependency-based applications. The spreadsheet has a grid initially consisting of ‘blank-input’ boxes (or so-called ‘cells’) that allows users to structure, modify, format and segment their models. It provides a good combination of text and graphics so that a cell value can be defined very compactly in the text-based formula language

and all cell values can be displayed on the screen. A cell name (i.e. *A1, A2*) can be regarded as a definitive variable and its associated formula as a definition.

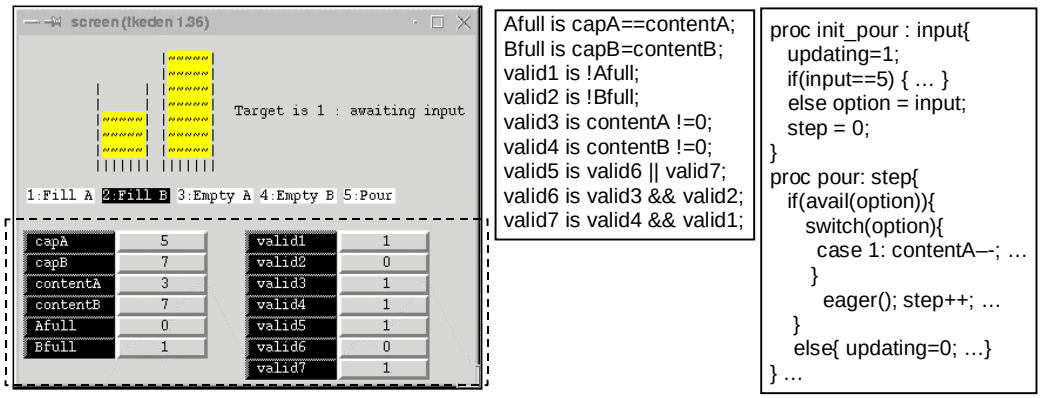


Figure 2-5: The Jugs model with an array of columns to display the current values of key variables

Figure 2-5 depicts a variant of the original Jugs model (cf. {Jugs92} in Appendix B) that is discussed at this stage to illustrate how definitive scripts can be used to construct a model with features similar to a spreadsheet. An array of columns (highlighted by the dashed rectangle) has been added to display the current values of key variables. This array demonstrates that within definitive scripts there is a process of updating dependency similar to that in spreadsheets. For instance, in the model, changing the value of contentA or contentB will affect the value of other variables.

The Jugs model is developed using the tkeden interpreter. In the model, the three mechanisms – definitions, functions and actions – are combined. As illustrated in Figure 2-5, two jugs: jug A (on the right) and jug B (on the left) are visualised, together with the set of permissible menu options. Each jug has its own capacity (capA and capB) and content (contentA and contentB). The definitions – Afull, Bfull and valid1 through valid7 – represent observable states of the model. For instance, Afull defines whether jug A is full or not.

Two types of interactions are involved in using the model. In one mode of interaction, the users can change the state of the model through predefined appropriate actions. For instance, users can select menu options whose validity determined by this set of definitions associated with the variables valid1 through valid7. Each menu selection triggers the Eden action int_pour to make an automated sequence of redefinitions in action pour. Each such definition changes either the value of contentA or contentB, or both, and consequently will affect the values of variables that depend upon them (as listed in the dashed rectangle in Figure 2-5). As a result, the state and visualisation of the model are also changed. In the second mode of interaction, users can interact

through the Eden interface to redefine variables freely to reflect a shift in perspective on the model. For instance, the redefinitions

```
Afull is contentA == capA –1;
Bfull is contentB == capB –1;
```

reflect the idea that a full jug is not filled to the brim. The latter mode of interaction gives more flexible interaction to users since they can arbitrarily redefine a definition in an exploratory fashion.

The two modes of interaction are respectively similar to the redefinition of a cell value (typically) by a spreadsheet user and the redefinition of a formula (typically) by a spreadsheet designer. In MWDS, there is not a clear distinction between the roles of explicit and implicit redefinition. The automated sequences of redefinitions illustrated in the Jugs model are similar to the use of spreadsheet macros.

- The role of definition and dependency in geometric modelling.

As discussed in Section 1.3, many researchers have studied the use of definitive principles as an underlying concept to support geometric modelling. The interactive graphics language [Wyv75] introduced by Wyvill is one example of work in this spirit. In developing his language and system, Wyvill aimed to provide an interactive environment in which the user can easily define, modify and adjust geometric entities.

```
1     %donald[10]
2     real width, doorwidth
3     boolean open
4     line door, n1, n2
5     point hinge, lock, NW, NE, Lframe, Rframe
6     NW = {10,90}
7     NE = {90,90}
8     Lframe = NW+{20,0}
9     Rframe = Lframe+{doorwidth,0}
10    n1 = [NW,Lframe]
11    n2 = [Rframe,NE]
12    open = true
13    width = doorwidth
14    door = [hinge, lock]
15    hinge = Lframe
16    doorwidth=20.0
17    lock = hinge+ if open then {0,-width} else {width,0}
```

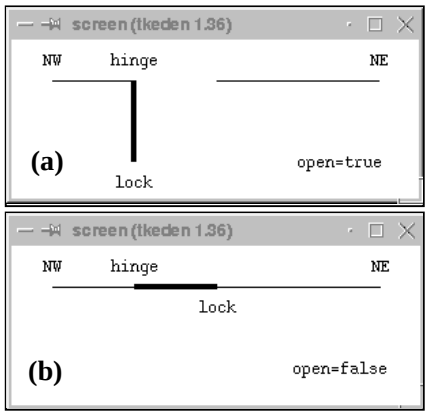


Figure 2-6: Scripts and screenshots of a geometric model representing a 'door'

[10] %donald is used to mark that the script defined after this point is a DoNaLD script

This section will give a simple illustration of how definitive scripts are used to support geometric modelling. The script in Figure 2-6 defines the line drawing to represent an open door as it might appear on an architectural plan (cf. Figure 2-6(a)). Each variable represents **`point`**, **`line`** and **`label`** that can be mapped directly to its referent on the visualisation of the model. Scripts are defined in an easily interpretable way, for instance, a line consists of two points, and changing one of these points affects the position of the line.

In Figure 2-6, n1 and n2, as defined at line 10 and 11, represent the sections of wall on each side of the door. The actual door is represented by the variable door at line 14, which is dependent on its two end points: hinge and lock. The dependency network diagram to display the dependencies between variables can be drawn as depicted in Figure 2-7.

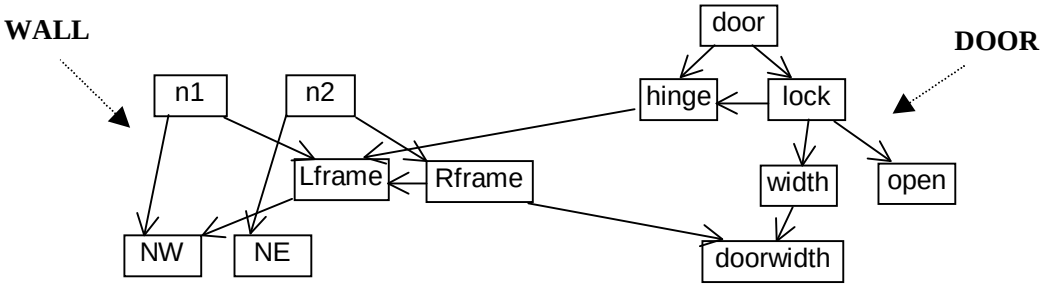


Figure 2-7: The dependency network diagram for the script in Figure 2-6

Redefining variables will cause the re-evaluation and update of the variables that depend upon these variables. From the diagram above, it can be seen that redefining the position of NW will affect the position of all points and lines with the exception of the point NE. Changing the size of doorwidth results in repositioning of Rframe, lock and hence changes the size of the door. Changing the value of open to 'false' or 'true' will reposition lock so that the visualisation of the door becomes 'open' and 'closed' as shown in Figure 2-6 (a) or (b) respectively.

- The role of definitive scripts in a database model

As discussed in Section 1-4, Codd's relational model allows attributes to be organised in a systematic way according to the relationship between them. Todd [Todd76] implemented a system to maintain relationships between data based on Codd's relational model by using dependency (cf. ISBL discussed in Section 1-4).

The Eddi notation was developed as an extension to tkeden to illustrate how we can use definitive scripts to set up and query a relational database. Eddi is implemented using the Eden evaluator and can run on the (d)tkeden systems. Eddi implements the five basic operators of

relational algebra – union(+), difference(-), natural join(*), intersection(.) and selection(:) – identified by Codd[Codd79]. The syntax of the Eddi data definition language (DDL) and data manipulation language (DML) can be seen in Figure 2-8. In the panel on the left in Figure 2-8, the first two Eddi DDL commands respectively create a table ALLFRUITS and insert values into the table.

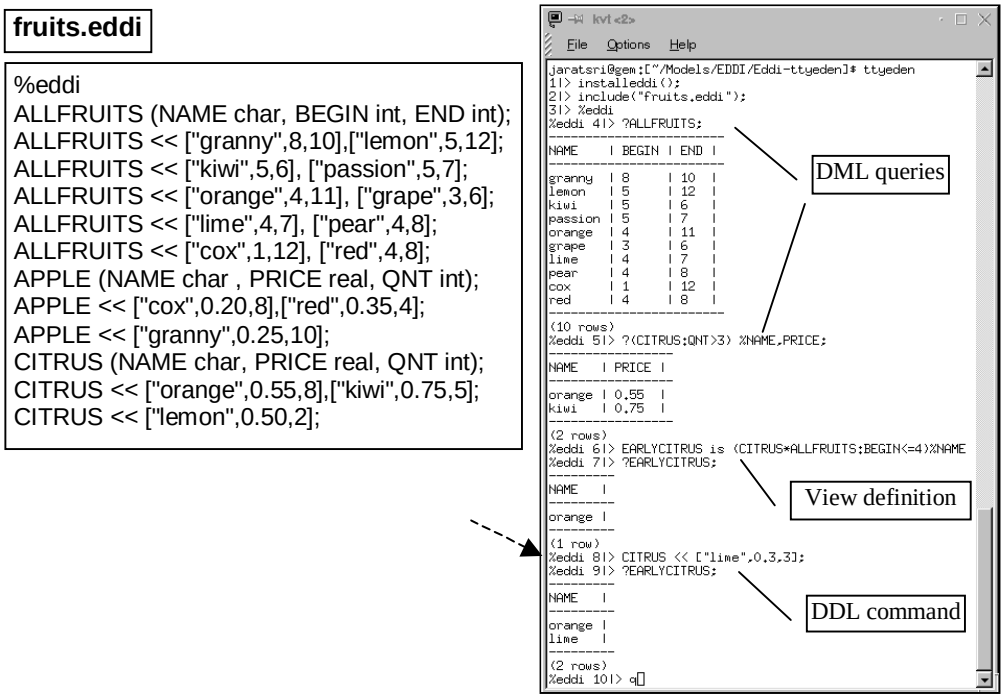


Figure 2-8: Illustrating the use of Eddi

The panel on the right in Figure 2-8 is a snapshot to illustrate the use of the Eddi DML. The definition of EARLYCITRUS corresponds to a relational database view, rather than a query. The SQL [Harr98] equivalent of this definition is:

```
CREATE VIEW EARLYCITRUS AS
(SELECT name FROM CITRUS, ALLFRUITS
WHERE CITRUS.name = ALLFRUITS.name AND begin<=4)
```

The use of dependency in Eddi is illustrated by adding a new variety of CITRUS named lime (as indicated by the dashed arrow). As a result of maintaining dependencies, the values of EARLYCITRUS are automatically updated with an additional value lime.

Definitive scripts can be used to represent diverse kinds of referent as shown through previous illustrative models. Different kinds of representation can also be used in combination in one environment, as shown in Figure 2-9. Figure 2-9(c) is extracted from a script in which the ALLFRUITS table (Figure 2-9(a)) is defined in Eddi and the display (Figure 2-9(b)) of the data associated with the fruit of current interest (viz. 'granny') is represented using DoNaLD and

Scout. This illustrates the use of diverse definitive notations each of which has a particular role in representing the external referent.

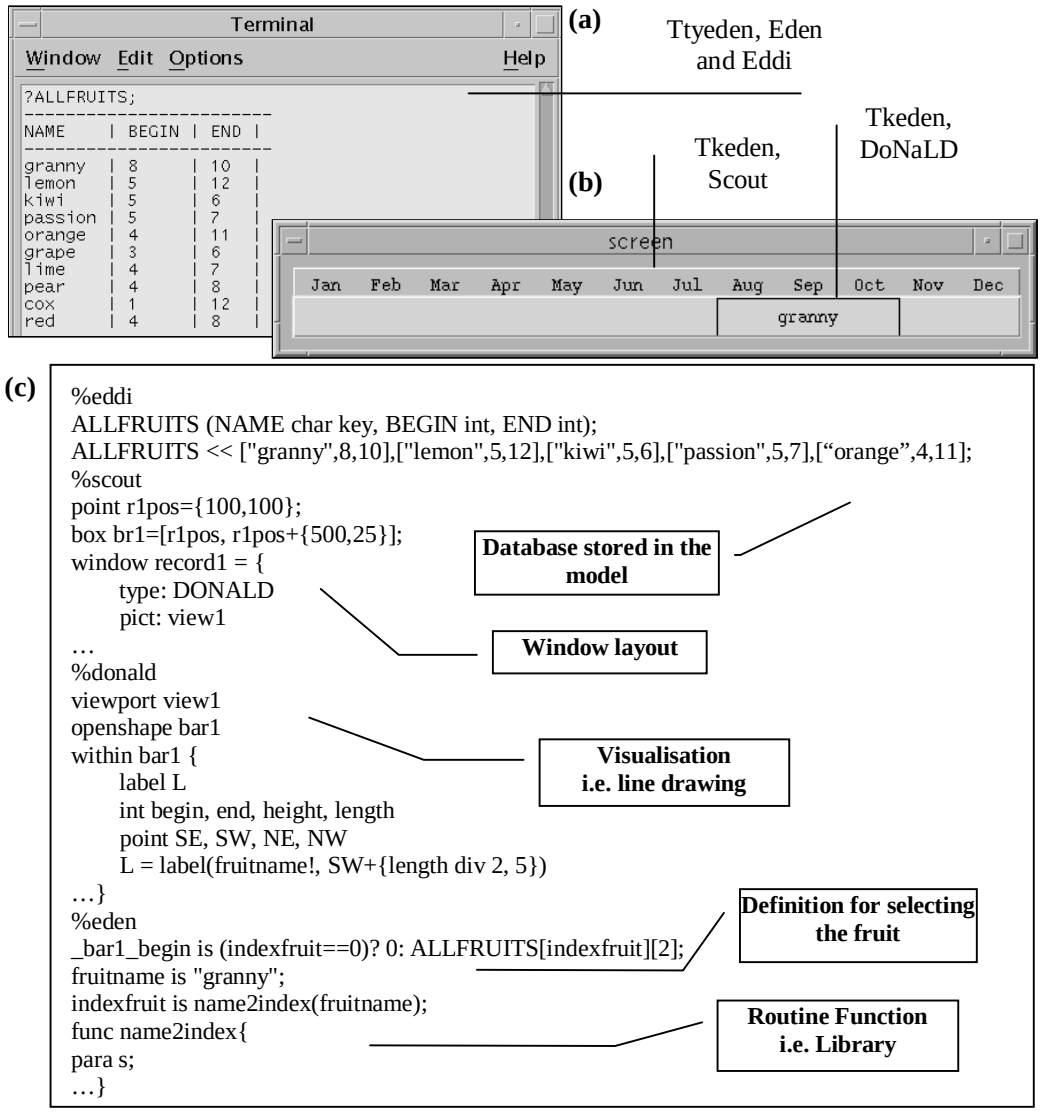


Figure 2-9: (a) the tabular view of the ALLFRUITS relation (b) a graphical presentation of the fruit of current interest (i.e. ‘granny’) (c) the various extracts from types of definitive script in the model

2.2 Single-agent modelling

As discussed in Section 1.6, the correspondence between a definitive script and its referent is established on a state-by-state basis (cf. Figure 1-5). The observables in the referent, and the modeller’s atomic interactions with the referent, have direct counterparts in the computer model. Whereas the relation between the computer model and the external world is normally established through the preconceived – and typically publicly agreed – conventions of a closed-world model

(cf. Figure 1-6), the relationship between a definitive script and its referent is experiential and essentially private and subjective.

The models discussed in the last section have demonstrated the use of definitive scripts to directly represent various kinds of observable. MWDS is in the first instance concerned with 'single-agent modelling', that is, modelling passive situations in which there is no change of state without the intervention of the modeller. Definitive scripts, in this context, represent observables from the modeller's perspective. Such observables can reflect the modeller's imagination, experience and observation on external real-world entities. The current state of a referent, as construed by the modeller, is determined by the current values of the observables and the dependencies that hold between them. Observable can represent an explicit value or an implicit value in which the value of the observable is functionally dependent on the values of other observables. The modeller can change the state by redefining those observables. The interpretation of this change may reflect a shift in the modeller's viewpoint in response to new experience, or a change in the external referent. The model can be incrementally developed through representing the state with a set of observables, interactively changing the state, interpreting the corresponding state and refining the state until stable patterns of state-change that satisfy the modeller emerge. Satisfaction in this context may take many forms: realising a functional input-output or stimulus-response relationship, achieving a pleasing screen layout or tracing a target behaviour. Model building of this nature focuses on dealing with knowledge that is embodied in, and gained through, experience of interaction with computer artefacts.

2.2.1 General characteristics of single-agent MWDS

This section describes and illustrates how MWDS can serve the needs of single-agent modelling that entails private activity, personal experience and subjective construal; how situated modelling activity is blended with the emergence of a conceptual model; and how this applies to two different semantic relations that we identify as 'internal' and 'external'.

A key idea in MWDS is that the computer model serves to represent a situation and transformations associated with the contemplation of this situation. The model is being used, not to compute a result, or describe a behaviour, as in conventional programming, but to represent a state metaphorically, in such the same way that a physical artefact can be used as a prototype. The emphasis is upon representing the state so that it can be changed and interpreted by the modeller to reflect changes both in his perspective and in the external referent (cf. Figure 1-5).

Personal everyday exploration

In MWDS, the modeller's viewpoint on interaction with the computer model closely resembles his/her everyday interaction with an external situation. There are two components in the modeller's understanding of the situation: the immediate perception of current state and the implicit knowledge of potential interaction (cf. Figure 2-10). The relationship between the two is continuously evolving in response to practical activities, such as experimental interaction in the situation and exploration of the environment surrounding the situation, and conceptual activities associated with interpreting this experience, such as learning skills and recalling the consequences of patterns of behaviour.

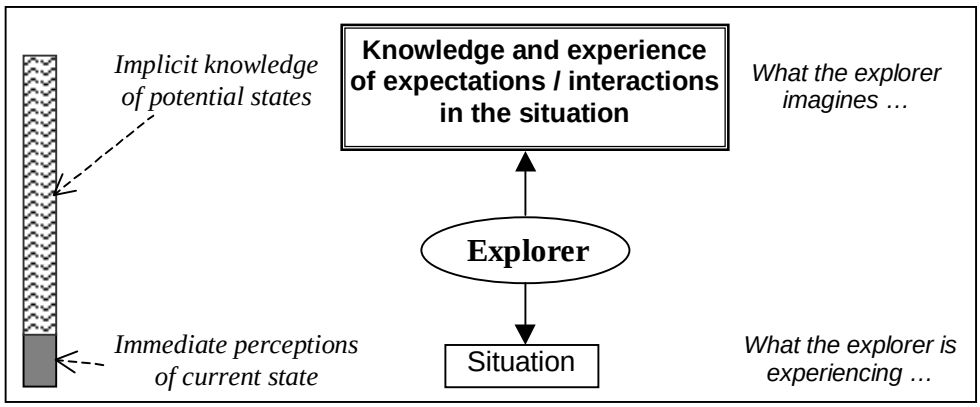


Figure 2-10: The explorer's understanding of a situation

The richness of the relationship between immediate perception of state and implicit knowledge of potential states depicted in Figure 2-10 is made clear by considering how different people comprehend and construe 'the same' external world or situation differently based upon their experience and knowledge. For instance, when I take my foreign friend to my country, we are apparently in exactly the same situation but we comprehend surrounding things differently. As far as implicit knowledge is concerned, I know the way to the local supermarket and know the sorts of food that I can find there. I also have a richer perception of immediate state, because I can speak and read the language and can construe the situation and interact in it more effectively. My friend and I can hardly be viewed as 'being in the same situation' since I have so much more background knowledge and experience of the environment than my friend has. However, my friend can enhance both his implicit knowledge and immediate perception of the situation by experimenting within the situation and communicating and interacting with people. Through such activity, there is a gradual change in understanding the situation.

MWDS activity and everyday exploration

Figure 2-10 is drawn with everyday interaction with a situation in mind, but applies equally to exploring the products of MWDS. When we interact for the first time with a model based on a complex definitive script, our situation is similar to that of the foreign visitor to an unfamiliar country. The modeller – in contrast to the naïve observer – typically knows far more about possible interaction with the model (in terms of potential states, activities and interpretations), and can also interpret the visible state of the model directly. This motivates the use of definitive scripts to build artefacts to represent our understanding of situations.

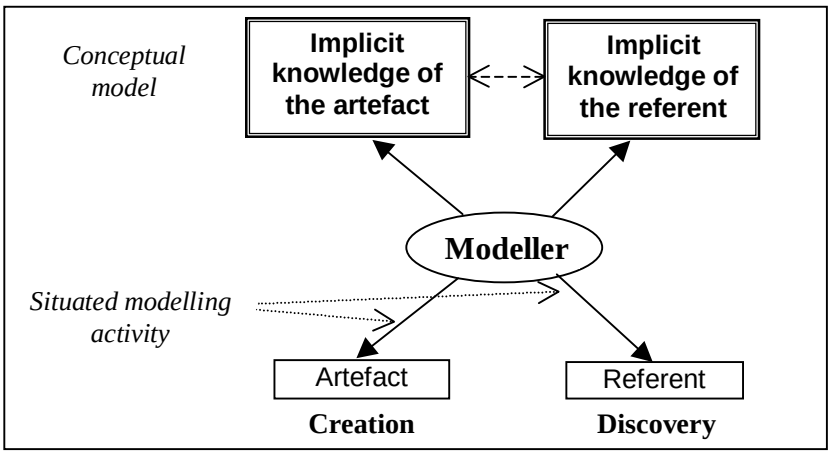


Figure 2-11: Modelling the modeller's understanding

Figure 2-11 shows how the activities associated with exploring a computer-based artefact and exploring an external-world referent are integrated in MWDS. Two activities, whose features are similar to everyday exploratory activities such as are described in Figure 2-10, go on in parallel. They are linked: there is a correspondence between observables in the artefact and the referent and also a correspondence between expectations and interactions at the conceptual level. They are also different: one activity involves exploring the situation and identifying the referent and the other involves experimenting with computer-based technology to construct an artefact. These two activities are labelled 'Creation' and 'Discovery' in Figure 2-11. MWDS involves situated modelling activity that combines experimentation with the referent and incremental construction of the artefact. Successful modelling leads to a blending between the implicit knowledge of the artefact and of the referent in the mind of the modeller. The significance of the artefact cannot be assessed through the immediate perception of its current state alone, but only through the potential interactions that are associated with implicit knowledge (cf. Figure 2-10).

Figure 2-11 helps to explain the personal nature of building an artefact using MWDS. It is commonly the case that each modeller has a different degree of knowledge of a particular situation. The modeller typically identifies observables and represents understanding based on his/her background knowledge and experience. The artefact is first designed based on the modeller's initial viewpoint on the external-world state, but both the artefact and viewpoint may change during interaction. The artefact can embody the user's observables and personal experience as perceived and imagined. Building an artefact in this sense is a kind of creative activity since it involves understanding situations, modifying the artefact and gaining new knowledge through interaction.

Artefact-referent relationship in MWDS

There are two ways in which the artefact-referent relationship depicted in Figure 2-11 can be related to MWDS. A definitive script can itself be viewed as an artefact whose referent is the state of the computer as experienced by the modeller. For instance, in Figure 2-6 the definitive script comprises DoNaLD definitions that refer to an associated visual representation on the computer display. Alternatively, a definitive script together with the associated visible computer state can be viewed as an artefact whose referent is a situation in the external world. For instance, the DoNaLD script and the associated line drawing in Figure 2-6 represents a conventional door.

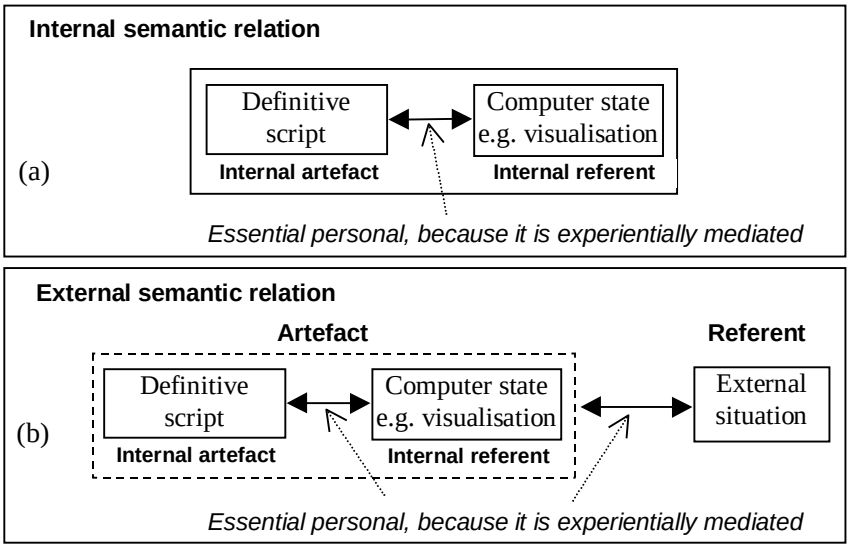


Figure 2-12: Two views of the semantic relation (a) within the computer state, (b) between the model and the external situation

These two views of the artefact-referent relationship in MWDS are illustrated in Figure 2-12. The first view, depicted in Figure 2-12(a), will be described as the *internal semantic relation* and the second, depicted in Figure 2-12(b), as the *external semantic relation*. These two semantic

relations loosely correspond to the two views of the concept of 'program' described by Smith in [Smith87], namely the specification view and the ingredient view respectively. In the former view, the semantics of the program refers to the computational activity that it prescribes. In the latter view, the semantics of the program refers to the interpretation of this computational activity in the external world.

Where the internal semantic relation is concerned, the treatment of a definitive script as an artefact requires justification. As discussed in Section 1.6, definitive scripts are used to capture the direct apprehension of open dependencies as experienced by the modeller. Observables in the external-world referent are directly reflected in 'the computer model' and there are direct counterparts in the computer model for changes to observables in 'the external world'. On this basis, a definitive script itself can be treated as an artefact whose current state can be observed by the modeller through querying variable values even in the absence of an explicit visualisation (cf. Listing 2-1). Where there is an explicit visualisation (as in Scout and DoNaLD scripts), the definitive script can be regarded as an artefact for which the external-world referent is the computer display. For instance, the script (a set of definitions) defines the display of the Door model as depicted in Figure 2-6. The significance of the script as a source of experience for the modeller is highlighted by combining the script in Figure 2-6 with the dependency graph in Figure 2-7. The implicit links in the text established by variable names define a physical structure for the script that is made explicit in the dependency graph. A key idea is the physical organisation of variables in ways that correspond to the modeller's view of the door and the wall as features of the line drawing on the computer display in Figure 2-6. This view of a definitive script as an artefact reinforces the link with the spreadsheet concept: compare the way in which the rows and columns of an examination spreadsheet are associated with students and subjects.

The appropriate interpretation for the internal semantic relation is as a 'successful blend' between the script and the external computer-state. Its success depends upon facts about the implementation (e.g. that changing the script indivisibly affects the screen and respects dependency) and the supporting technology (e.g. that there are 'enough pixels' for practical purposes). This blending allows the modeller to gain implicit knowledge of how 'changing the script' and 'changing the display' are related. The blending between an artefact and its referent in Figure 2-11 (as it applies to Figure 2-12(a)) also enables us to blur the distinction between 'the artefact' and 'the referent' as is necessary in embedding Figure 2-12(a) into Figure 2-12(b). Such combining of an artefact and a referent is common in everyday experience. For instance, consider

the way in which a lift system is designed so that there are panels on each floor to indicate the current position of the lift. The panel acts as an artefact whose state is indivisibly linked to the position of the lift. This means that in practice the panel is regarded as an intrinsic part of the lift system and the lift user make no distinction between the actual position of the lift and the position of the lift as indicated on the panel.

The internal and external semantic relations are respectively linked to two aspects of the agenda for MWDS: technical and conceptual. The technical agenda is concerned with the full exploitation of the computer as an artefact through the development of tools, notations and techniques. The conceptual agenda is concerned with general principles and potential applications. These two agendas interact. In making a new discovery or exploring a new domain, we may encounter new kinds of experience. In order to deal with the new experience we may need to build a new kind of artefact to imitate the external-world situation. Creating new kinds of artefact involves realising internal semantic relations between scripts and their experiential counterparts, and for this purpose we may need to design a new tool or notation.

Internal semantic relation

In MWDS, the internal representations are very significant because they are artefacts, embodying personal observables and perceptions, for the modeller to experience. This is in contrast to traditional program semantics, where (e.g.) any two programs that have the same input-output behaviour are regarded as equivalent. Smith in [Smith87] describes this indiscriminate identification of programs that have the same behaviour as 'promiscuous modelling', and asserts that "although, promiscuous modelling may be helpful in answering large-scale and hence rather coarse-grained questions, …, it can be pernicious when one asks fine-grained questions about control, intentional identity, and the use of finite resources." MWDS enables the modeller to identify and control the relationship between the script and the computer display. The blending of the script with the computer display that results is associated with using the computer as an instrument, as discussed in [BCH+01]. This is illustrated in [BCH+01] by considering the computer as an instrument for clock design: the variables in the script represent observables (e.g. the display of hour and minute hands) in the clock and a complementary set of definitions represents dependencies that connect the positions of the hour and minute hands to the current time.

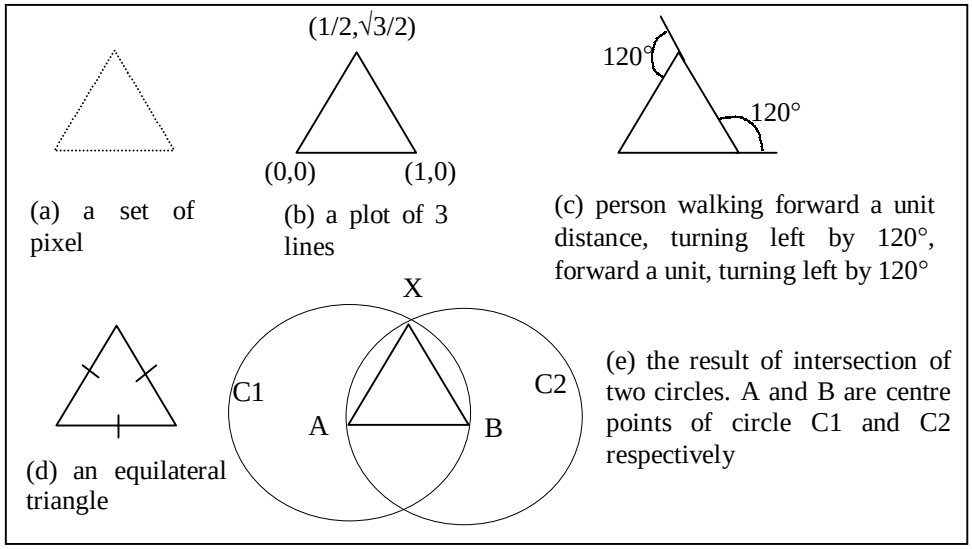


Figure 2-13: Various representations of a triangle taken from [FP89]

People perceive and represent things differently according to their different perspectives and objectives. There is no single representation that can service all that people require. Several representations for a triangle, as shown in Figure 2-13, illustrate diverse possible views on representing a triangle. Even though the external representation looks the same in each case, the internal representation may be different. For instance, Figure 2-13 (a) is made up with a set of pixels and Figure 2-13 (b) is defined by three lines that link coordinate points on the XY plane. In MWDS, the internal representations of these triangles will make use of different sets of observables.

The different internal semantic relations associated with Figure 2-13(c) and Figure 2-13(e) are illustrated by the example scripts and snapshots corresponding to each script shown in Figure 2-14. MWDS is essentially concerned with framing the internal representation of an artefact so as to reflect the role it has to play in representing its referent. In Figure 2-14, the two DoNaLD scripts represent two distinct styles of triangle. One is a triangle that results from the intersection of two circles, so that relevant observables include circles and the radii of the circles (represented by the variable DIST1). The other triangle is designed to represent a person walking (cf. Figure 2-12(c)), so that relevant observables include variables, such as step, that represent the movement of a person.

Each script comprises a set of observables that is required to display its corresponding triangle. Each observable has a geometric counterpart on the display; for instance, CA and CB refer to two circles on the figure. We can change the values of observables and this will affect the

shape of the geometric referent accordingly. For instance, if the scripts are combined, the size of the two triangles can be changed so as to be consistent with the value of the variable DIST by defining DIST1 = DIST. Changing the value of the variable DIST will then affect the size of both triangles.

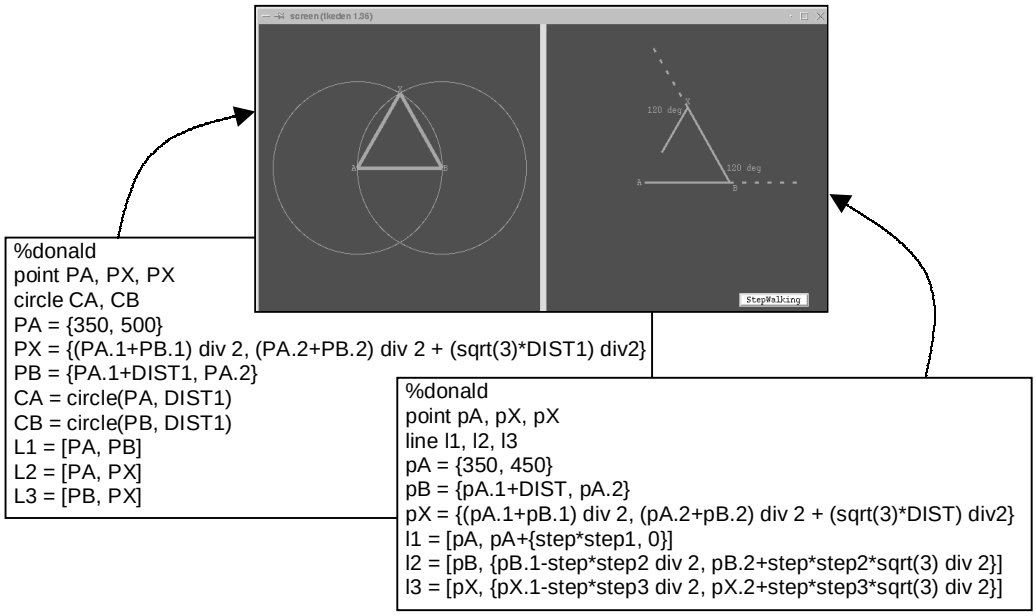


Figure 2-14: A screenshot of two different perspectives on representing a triangle

External semantic relation

In MWDS, the external semantic relation is at all time potentially the subject of negotiation. The interaction between the modeller's state of mind and the artefact he/she is creating is dynamic, and the meaning of the artefact is shaped as it is being developed. The Number model {Number02} and its variants have been developed to illustrate how the model can be reshaped to serve different external situations as they occur to the modeller. Figure 2-15 shows several of these variants, such as might be used for example:

(1) By children to learn to read and to count numbers;
(2) For diagnosing problems in number and word recognition;
(3) For writing a cheque (either for a person or as part of a supermarket checkout);
(4) For learning to read numbers in different languages;
(5) For exploring the structure of number systems in different cultures.

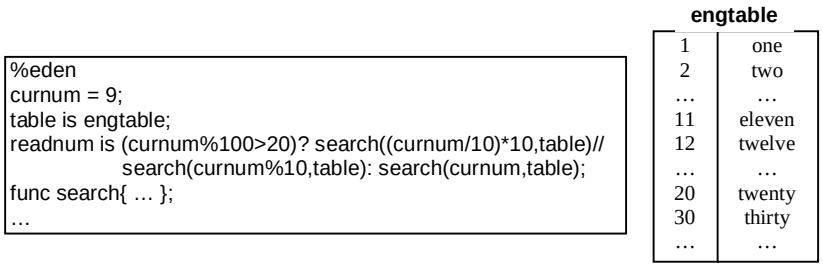


Figure 2-15: The seed for the Number model

All the models in Figure 2-16 are derived from the simple seed model in Figure 2-15. In its initial form, the seed model establishes a dependency between a number (the variable curnum) in the range 1 to 99 and the number as expressed in words (the variable readnum). The table, as shown in Figure 2-15, records the association between the digit and its English reading; in this case we are concerned with reading numbers in English. The first application of the Number model can be used in the situation (3): the model is a component of a device that reads a number and transforms it into words so that no visualisation is required. In the script in Figure 2-15, the assumption is that the number is given abstractly (e.g. taken from a computer memory) but in other contexts it may be more appropriate to observe numbers as strings of digits (cf. the cashier's view of the number to be entered through a keypad).

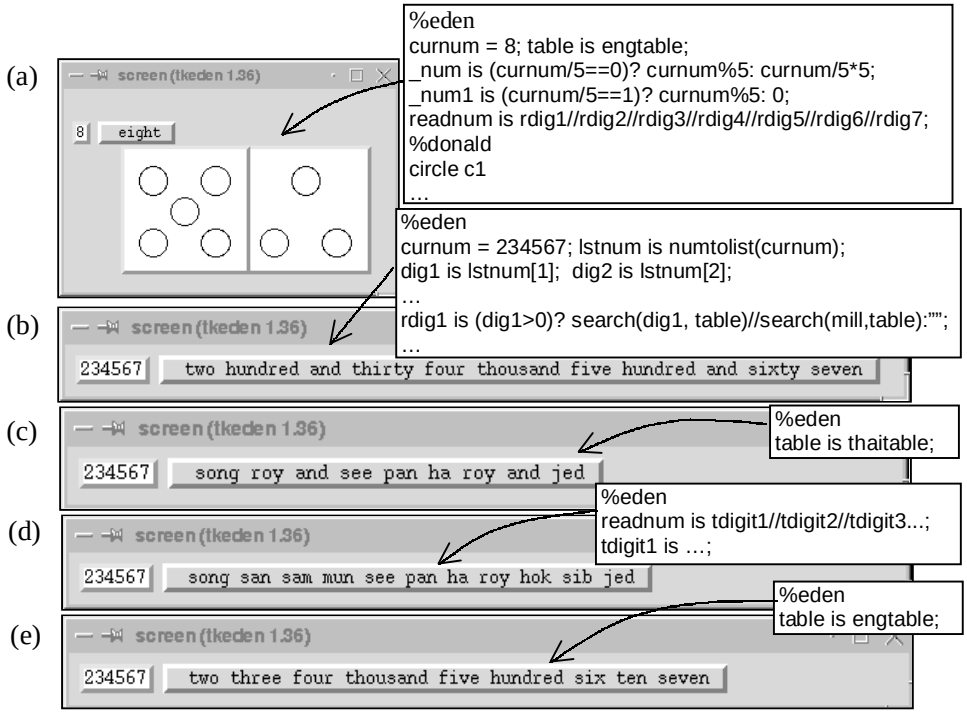


Figure 2-16: Several example variants of the Number model

The next development was to extend the model to deal with larger numbers. With the addition of a simple interface (consisting of Scout definitions for displaying two windows, one to get an

input number and the other to display the reading of that number), this model is as depicted in Figure 2-16(b). This application can be used for both educational purposes (1) and (2) described above – in the appropriate situations. For instance, the model might be useful to children with basic numeracy and literacy skills and also to the people who are learning English. A major feature of MWDS is that it can take account of the subtle effect of situation on the external semantic relation. By way of illustration, a brain-damaged banker is recorded as being unable to read the simple and commonest words but having no trouble at all in reading number words [Fron02]. Thinking about the variety of ways in which humans might observe and interpret numbers motivated extensions of the seed model (including all those depicted in Figure 2-16) that take digits rather than numbers as the basic observables. It also indicates the potential role for presentations of numbers that are based on artefacts rather than symbols. Figure 2-16(a) is a model for teaching children to read, to write and to count numbers that combines symbolic representation with an artefact in a way that is characteristic of MWDS. The modification of the model to read numbers in different language (e.g. Thai) is depicted in Figure 2-16(d). To do this, we needed to define a table (analogous to the engtable in Figure 2-15) for mapping between the digit and its Thai reading and to include a set of definitions to specify the structure of the Thai number system.

In practice, the various modifications of the Number model led to a series of scripts, each referring to different situations. One of these – the 'final-version' of the Number model – is the basis for all the models shown in Figure 2-16. The script extracts in Figure 2-16(a) through Figure 2-16(e) illustrate how the variants of the Number model can be successively derived from the final version by redefining variables or including a set of definitions. Note that, by redefining the table in the model depicted in Figure 2-16(b) to thaitable, the number is read in Thai, but the transcription is still based on the structure of the English number system, as depicted in Figure 2-16(c). A set of definitions that specifies the Thai number system is included so that the model can correctly read numbers in Thai culture, as depicted in Figure 2-16(d). Imitating the shift from Figure 2-16(b) to Figure 2-16(c), if we redefine the table from Thai to English at this stage, the number will be read in English, but the transcription will be based on the Thai number system, as depicted in Figure 2-16(d). By observing the change made by a series of redefinitions from Figure 2-16(b) through Figure 2-16(e), we can explore the structure of number systems in different cultures (cf. (5)). It is characteristic of MWDS that the transition from the model in

Figure 2-16(b) to that in Figure 2-16(c) was discovered by accident, but turns out to be unexpectedly useful in explaining the relation between Thai and English number systems.

The modifications described above are not exhaustive. Other simplifications could be done: the numbers required in teaching children are very simple (e.g. in the range 1 to 10) so that the script can be less complicated than the one required in the other variants of the model (e.g. Figure 2-16(b) through Figure 2-16(e)). Throughout the development of the Number model, the important issue is not so much the complexity of the modelling activity and redefinitions, but the rich interpretations of the situations and perceptions of the modeller. By studying the script, we can understand how the script is changed to reflect the external situation and the modeller's perspective. For instance, the representations of the number are first based on its abstract value (cf. Figure 2-15), then later based on the list of digits (cf. Figure 2-16). This change of representation may be motivated by a change in situation (e.g. a number is keyed in digit by digit) or a change in the modeller's perspective (e.g. reading a digit in association with its position).

In MWDS, many other instances of potential redefinition are motivated by the different roles that the modeller can play. Acting as a designer, he/she may change attributes such as the colour, font and size of the number in the Number model for children (cf. Figure 2-16(a)). Acting as a user, he/she may consider such issues as how to improve the way we key in the input number and the presentation of the word-based output. The modeller can also act in a role that is outside the scope of either the designer or the user, as when reconfiguring the display to convenient size for demonstration, or introducing alternative non-standard definitions that express 1200 as 'twelve hundred'. The openness of the Number model is further illustrated by the fact that it can be directly used in conjunction with other models that make use of integer data values.

2.2.2 The role of MWDS in construal

It is part of human nature to seek to understand, to be able to predict and to exercise some control over the world we live in. Kelly [Kelly55] pictured this by saying that we operate as 'personal scientists', developing implicit 'theories' about our experience. Once 'theories' have been proved, tested and accepted, they supply the rules that can be applied to relate and understand behaviours of interesting domains. Before this stage is reached, the process of observing, expressing and understanding problem domains is involved.

In building an artefact to represent our understanding of a situation, there are two types of situations to consider. In one type of situation (the 'single-agent' scenario), the modeller regards him/herself as the sole instigator of change. In the other (the 'multi-agent' scenario), the modeller observes that there are apparently changes beyond his/her control and attributes these to other agents. This informal classification of situations is based on the modeller's interpretation and forms part of his/her 'construal', in a sense to be elaborated below.

So far our discussion of MWDS has focused on the single-agent scenario. In this scenario, an artefact is typically being used interactively to stimulate thought or to capture observations. For instance, when using a spreadsheet as a single-user application in financial planning, the user is the sole instigator of change. In the multi-agent scenario, the modeller stands in the role of an external observer of concurrent interaction involving many agents. Building an artefact to represent such a situation is more complicated: the modeller is not only concerned with how he/she interacts, but also with understanding how other agents interact.

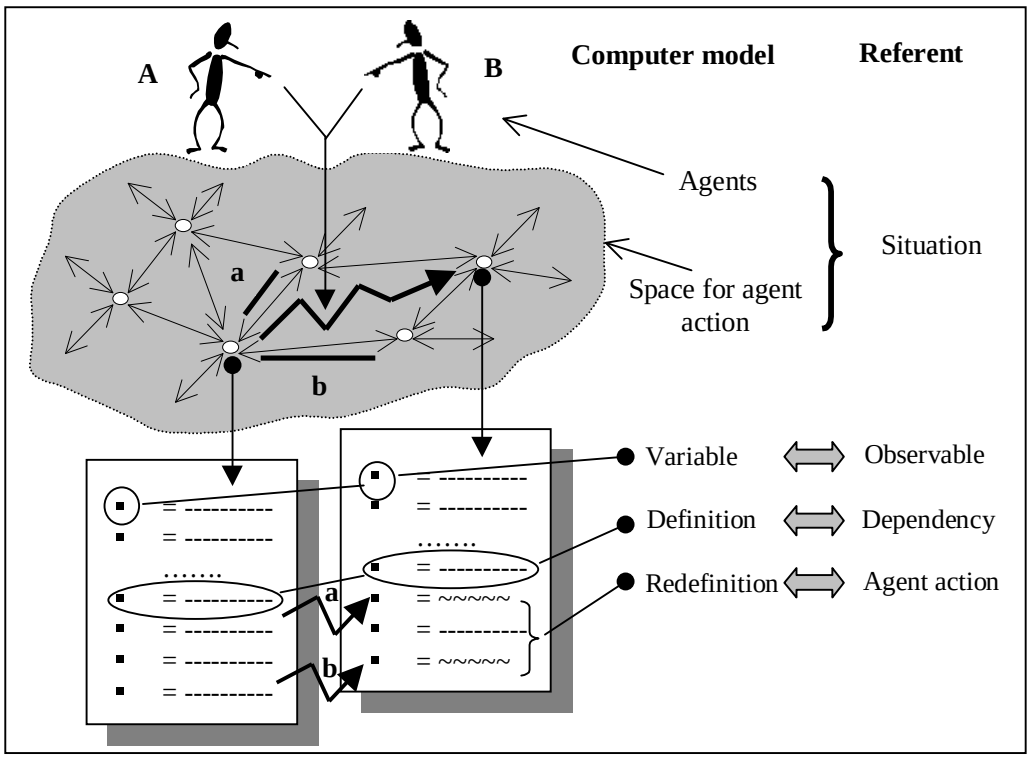


Figure 2-17: MWDS for computer-based construal

MWDS provides a framework for building such artefacts. Within the artefact, it is necessary to represent concurrent actions of several agents. Figure 2-17 illustrates how MWDS can be applied in this context. In the figure, the space for agent action corresponds to the state space in Figure 2-1(b) and the redefinitions **a** and **b** to the redefinitions labelled 1 and 2 in Figure

2-1(b). Two agents **A** and **B** are acting concurrently in this space to perform the redefinitions **a** and **b** respectively. In Figure 2-17, the definitive script represents the modeller's view of state in his/her role as an external observer.

The framework for MWDS depicted in Figure 2-17 is a generic pattern for the representation of the modeller's understanding, or *construal* of a situation. The term construal here refers to 'sense-making' in a very general context that embraces both single-agent[11] and multi-agent scenarios. People can construe situations differently based on their roles and objectives. For instance, for the artist, construal is associated with exploring and making sense of personal experience and imagination through the creation of an artefact. This is typically a single-agent activity. The work of the playwright or novelist has a multi-agent aspect; he/she construes fictional situations in terms of the roles of characters and how they interact. The engineer engages with construal in connection with the design of real-world products to perform a specific function. The scientist tries to understand a complex phenomenon by devising construals through observation and experiment.

MWDS is most closely related to the way in which the experimental scientist uses an artefact as a means for the metaphorical representation of observables. For instance, as Gooding describes in [Good90], Faraday constructed artefacts to represent observables such as electrical currents, magnetic fields and the relationships between the polarity of a magnetic field and the direction of current. Gooding introduces the term 'construal' to refer to a concrete artefact to embody understanding the experimental interactions such as Faraday favoured when trying to understand complex phenomena. In emphasising the role of physical artefact in understanding, Gooding is stressing the crucial contribution of knowledge of the physical world to scientific theory. In a similar spirit, Feynman points out the essential non-mathematical complement to a theory-based perspective on physics [FLS64]:

> "A physical understanding is a completely unmathematical, imprecise, and inexact thing, but absolutely necessary for a physicist".

MWDS as depicted in Figure 2-17 offers a framework for constructing construals as physical artefacts in the sense advocated by Gooding. It is also associated with a shift perspective on computer science similar in spirit to what Gooding and Feynman promote for physics. In computer science, most people tend to interpret computing in terms of a mathematical theory of

[11] When applying Figure 2-17 in a single-agent scenario, the modeller replaces agents A and B

computation, whilst in MWDS we emphasise the embodiment of computation within the external situation. In the context of Figure 2-17, the concepts of agent, dependency and observable are the key concepts in construal. Using definitive scripts as a means to construct computer-based construals of situations assists the cognitive process of identifying agency, dependency and observables. This activity relates to aspects of computing for which there is no theory, that can only be explored within a pragmatic framework, such as are significant in respect of pre-articulate activities, pre-formalisation, situated modelling and personal viewpoints.

In Figure 2-17, the use of humanoid icons to depict agents is not intended to exclude impersonal or inanimate forms of agency, but to stress a key principle of definitive scripts. All agencies are construed as similar to human agency. All state-changing agents are construed as operating through changing observables and, in their turn, responding to changes in observables. When construing complex phenomenon, we will need to postulate observables that we cannot directly observe (e.g. electric currents). The strategy for construing such interactions in a multi-agent scenario is described in [Bey97]:

> "For inanimate agents, the stimuli and responses typically involve observables that cannot be directly sensed and manipulated by a human agent. Knowledge about the protocols for interaction of such agents has then to be represented in ways that are intelligible to a human agent."

Figure 2-17 is to be interpreted in the implicit context of the modeller's exploratory interaction with the computer model and its referent. The aim of this interaction is to create a model embodying relationships between observables, dependencies and agents congruent to those that the modeller projects onto the referent. The computer model provides perceptible counterparts for relationships that typically cannot be directly observed in the referent.

Figure 2-17 illustrates the application of what we identify as 'definitive principles for the representation of state' [Slade89]. Throughout this thesis, a model that exploits definitive principles will be referred to as a 'definitive model'[12]. Figures 2-11 and 2-12 illustrate the distinctive – and, to our knowledge, otherwise unremarked – way in which the internal and external semantic relations are treated in MWDS. On this basis, throughout the thesis other kinds of computer-based model will be described as 'traditional'. In practice, many of the definitive models that have been implemented using the (d)tkeden tool have features other than definition.

[12] The term 'model' is being used here in the sense associated with open development (cf. Section 1.6) and does not refer to a mathematical model.

In MWDS, we are concerned with definitive models that – apart possibly from user-defined operators to be used in defining formulae and some forms of construct to automate redefinition – purely comprise definitions ('pure definitive models').

LSD Analysis

In interacting with an unfamiliar definitive model, the explorer (cf. Figures 2-10 and 2-11) lacks the original modeller's knowledge and experience of expectations and interactions. This model might be a construal of a multi-agent scenario (cf. Figure 2-17). To understand such a model, the explorer needs to grasp the original modeller's conceptual model (cf. Figure 2-11), identify the agents within the model and account for their interaction. In the particular case when the explorer is the modeller, for instance, throughout the development of an artefact, it is also important to be able to record and document how to construe and interact with the artefact.

LSD is a special-purpose notation designed for specifying and documenting our experiences, expectations and possible interactions in the context of MWDS. It supports a form of observation-oriented and agent-oriented analysis originally developed by Beynon in collaboration with Mark Norris of British Telecom in 1986 [BN88]. Mike Slade [Slade90] further elaborated on its design and characteristics as an agent-oriented notation. LSD is interpreted in several ways according to context: for construal (an LSD 'account'), for description and for specification.

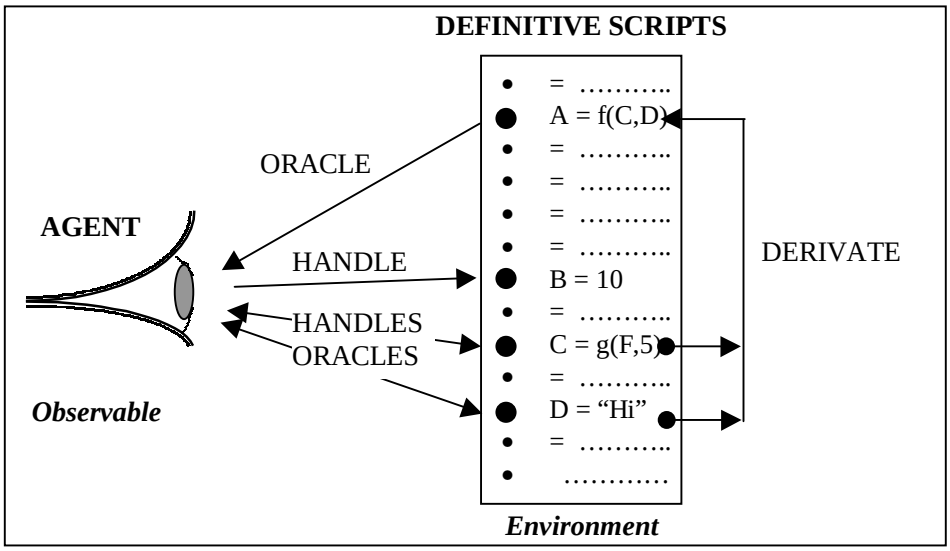


Figure 2-18: Definitive script as observer's model of state ('one-agent' modelling)

The basic concepts of LSD will be introduced with reference to single-agent MWDS. In this context, definitive scripts are used to document the interactions that describe and/or prescribe

the modeller's view of a situation. Definitive variables[13] implicitly serve as parameters for change. By interacting with the script, the modeller can infer the current status of the artefact and see the effect of changing a parameter. Different classes of variables can be characterised from the modeller's point of view. There are those variables – such as readnum in the Number model (cf. Figure 2-16(a)) – that are implicitly defined and are subject only to indirect changes of value; these will be called **derivates**. Some variables – such as curnum whose value is defined by the user – are conditionally under an agent's control; these will be called **handles**. Other variables, such as the choice of the table, reflect the external environment; they can be observed by the modeller, but are subject to change beyond the user's direct control; these will be called **oracles**.

The relationship between handles, oracles and derivates is depicted in Figure 2-18 – note that the classifications are not exclusive. Some variables are handles for the user, some are oracles and the derivates indicate how these variables are indivisibly coupled in change.

In a multi-agent scenario, an additional function of LSD is to describe the roles that can be played by the agents participating in the particular situation. LSD agents can represent both human participants and inanimate components. In keeping with the principle that all agencies are construed as similar to human agency, these roles are specified with reference to those aspects of the situated state to which the agent can respond and those which it can conditionally change [BNOS90]. In a multi-agent scenario, LSD then supports the systematic analysis and metaphorical representation of observables through which stimulus and response are mediated. Exceptionally, LSD can be used for specification purposes, but in general it admits many different operational interpretations, corresponding to different presumptions about the environment in which agents interact, and the nature and reliability of their stimulus-response

Figure 2-19 illustrates how the modeller, in the role of an external observer, establishes an LSD classification of observables in multi-agent MWDS. The blocks of definitions in the right hand column of the figure correspond to observables that are bound to different agents (Agent 1, Agent 2 etc.) internal to the model ('internal' agents). The variables bound to an agent are classified as **state** variables. Such variables reflect the properties and features that are attached to the agent. (These variables are not normally considered in single-agent modelling since it is unusual to take account of the state of the modeller.)

[13] The term 'definitive variable' is used to refer to a variable in a definitive script. Unlike mathematical and programming variables, definitive variables can be interpreted as observables (cf. Figures 2-12 and 2-17).

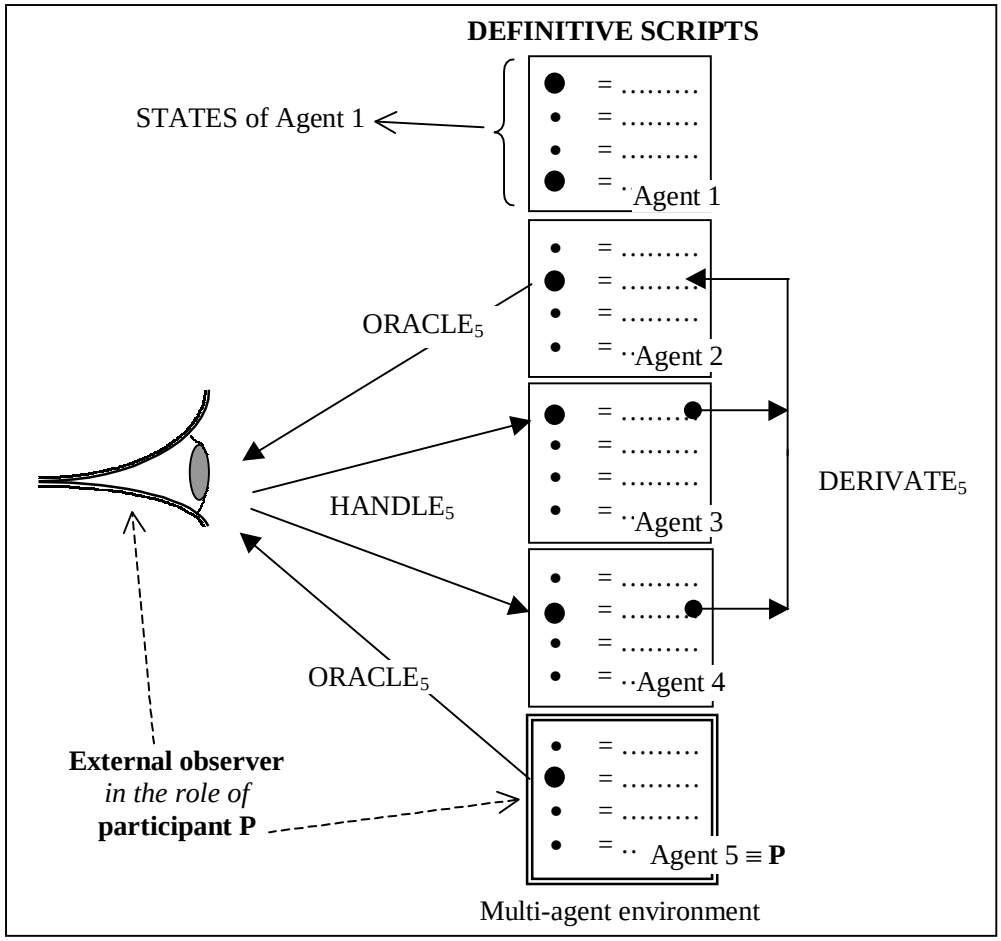


Figure 2-19: Definitive script as observer's model of state ('multi-agent' modelling)

The presence of agents other than the modeller in Figure 2-19, in contrast to Figure 2-18, reflects the different perceptions of the modeller about possible instigators of change. Whereas in Figure 2-18 the assumption is that the effect of any action of the modeller is predictable, and can be modelled via dependencies between observables, in Figure 2-19, there are changes to observables that cannot be attributed to the modeller's actions. In this way, agents can be viewed as complementary to dependencies.

In the multi-agent scenario, Figure 2-18 has a simple counterpart which shows how the observables associated with the internal agents are classified as oracles, handles and derivates from the perspective of the external observer. Such a classification is shown in Figure 2-20(a) for the particular case of the electrical circuit previously used to illustrate Agentsheets in Chapter 1 (cf. Figure 1-4). Figure 2-20(b) illustrates the way in which LSD can be used to classify observables from the perspective of each of the internal agents (B, S, C and L). Figure 2-19 depicts the way in which the external observer develops a 'personal construal' for an internal agent (Agent 5) in the multi-agent environment. To achieve this, the external modeller

participates, or imagines participating, in the role of each internal agent to identify the oracles, handles and derivates pertaining to that agent (cf. $ORACLE_5$, $HANDLE_5$, etc).

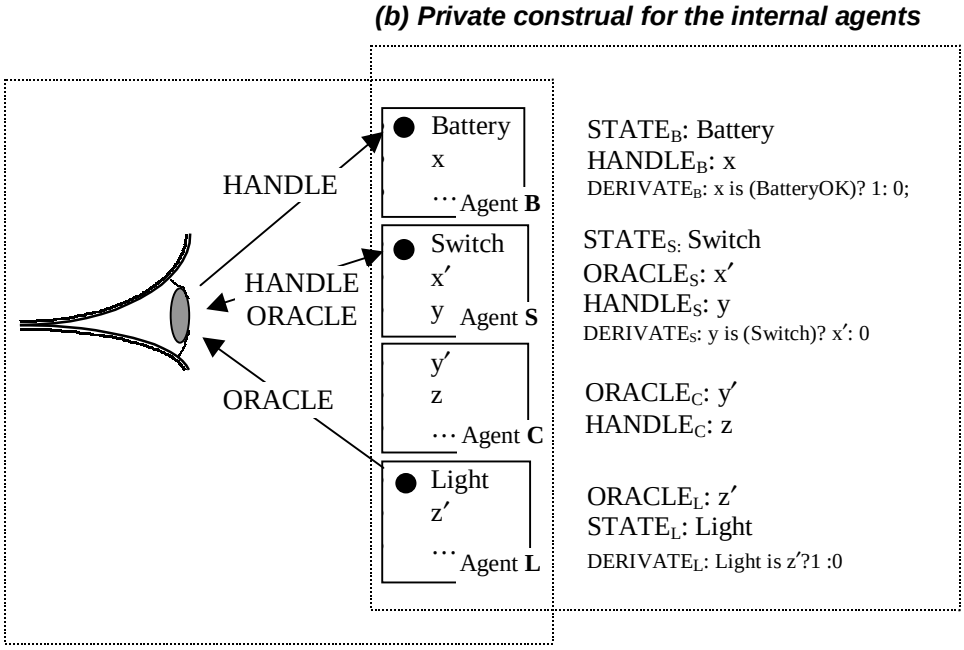


Figure 2-20: An LSD account of the electrical circuit (cf. Figure 1-4)

In building a definitive model, an LSD account is typically used as a design-sketch – not as a complete design. The behaviour of the model to which the account refers is very open. For instance, in the case of the electrical circuit in Figure 2-20, we can introduce the derivates

x′ is x, y′ is y

to describe the normal and typically reliable behaviour, but the light can fail, the battery may be flat, the switch can be stuck and the cable can be cut. The interpretation of oracles in this context is sufficiently open to correspond to unexpected situations. For instance, an oracle might only be observed intermittently or only provide an approximate or out-of-date value. In addition, there are other potential behaviours outside the scope of any given LSD account. For instance, in the electrical circuit, the battery life depends on time.

An LSD account also makes it possible to attribute state changes to agents. For instance, in the electrical circuit, the switch is normally operated by the user. This can be recorded in LSD by associating a **privilege** to the user agent. This takes the form of a guarded action:

condition $\rightarrow$ *action*

where the action involves redefinitions of the values of observables and instantiation or deletion of agents. The user's privilege to operate the switch can be expressed:

switch==on → switch=off

or – if we wish to consider the position of the user relative to the switch:

(switch==on) && (distance_between_user_and_switch<=…) → switch=off.

The set of privileges for an agent is described as its **protocol**. As is explored at length in [Slade90], it is impossible to give a precise specification of the assumptions that are needed in order to give an operational behaviour to an LSD account. By way of illustrating some of the issues, in the context of the electrical circuit, the switch may be attached to a timer so that it has a privilege:

(switch==on) && (time>$time_0$)→ switch=off

where $time_0$ refers to the time at which the switch was switched on. This privilege is quite different in character from the user's privilege to operate the switch, in that it represents a potentially reliable stimulus-response pattern associated with the agent that is linked to the observation of time. There is also a possibility of conflict in any behaviour associated with the LSD account in this case, since more than one agent can change the same observable. The character of an LSD account as a way of documenting interactions involving real-world observables and agents is similar to that of Faraday's informal – but essential – record of the interactions with his construals, as described by Gooding in [Good90].