# 1 Selected dependency-based applications

In this chapter, the concept of open-development dependency that underlies MWDS is identified and compared and contrasted with closed-world dependency. An early example of the implicit use of dependency can be found in Ross's APT[1] language [ITT67], developed in 1967 for automatic programming of numerical machine tools. Several dependency-based applications, such as spreadsheets, Agentsheets, and several graphical modelling tools, are reviewed to illustrate the potential virtues of using dependency. The last section introduces a modelling framework, developed at Warwick over more than ten years, that adopts dependency similar to that reviewed in those applications as a fundamental concept.

## 1.1 Dependency in closed world and open development

The concept of 'dependency' has been familiar in diverse disciplines for many years. Dependency can be interpreted in many ways. In this thesis, we are particularly concerned with the distinction between dependency as it is used in the 'closed world' and in 'open development' in the sense of Brödner.

**Dependency in closed worlds**: Closed-world dependency is associated with declarative constraints. They are typically prescribed by equational relationships or logical predicates and make absolute assertions about relationships between variables that must hold in every acceptable state. Declarative constraint languages have been used in many applications, including general-purpose programming (e.g. Prolog [CM87]), graphics (e.g. SketchPad [Suth63], Juno [Nel85, HN94]) and databases (e.g. CoLan [BG94], RL/1 [Denn91]). In the applications, declarative

[1] APT stands for "Automatically Programmed Tools", and the APT language.

constraints are well-suited to expressing the feasibility constraints imposed by a problem (such as writing a program, drawing a diagram, or configuring data) in a clear and concise way.

In interpreting a declarative constraint as referring to a world, variables correspond to observables whose values are consistent with observation. It is appropriate to call it a 'closed' world because if – in exceptional or unexpected circumstances – observables have values that are inconsistent with the constraints, they can be given no representation. The word 'world' is used in this context to refer to all states within the scope of current interest and possible focus of attention rather than to the universal domain of experience.

Figure 1-1 depicts the way in which a declarative constraint can be used to specify all the possible states of a closed world. A particular solution to the constraint corresponds to a state of the world. For instance, in a logic programming environment, such as Prolog, the variables $x_1$ through $x_t$ are interpreted as inputs $i_1$ through $i_s$ and outputs $o_1$ through $o_t$ for a program, so that

$$(x_1, x_2, \ldots, x_s, x_{s+1, \ldots,} x_{s+t}) \equiv (i_1, i_2, \ldots, i_s, o_1, o_2, \ldots, o_t).$$

The declarative constraint $\Gamma$ frames the program as a relationship between input values (to be specified) and output values (to be determined). In this application, the declarative constraint can have any number of solutions, potentially none. In other applications, a declarative constraint should ideally have a unique solution. For instance, in a constraint-based graphics system, such as Juno, the variables $x_{i1}$ to $x_{in}$ are interpreted as geometric elements such as points and lines in a geometric figure. The declarative constraint $\Gamma$ (usually formulated as a set of constraints) specifies the relationships between these elements needed to determine the geometric figure.

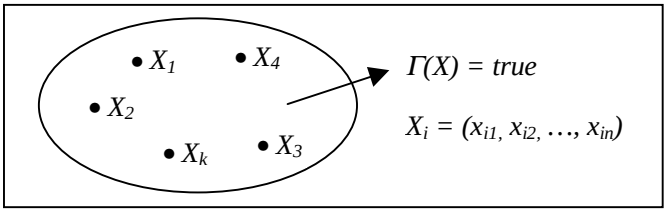


**Figure 1-1: A diagram to illustrate the concept of a declarative constraint**

In a practical application using declarative constraints, *constraint satisfaction techniques* play an essential role. A constraint satisfaction technique is a recipe for finding values that satisfy a declarative constraint. Its primary role is in constructing solutions to a problem. In a logic programming environment, the execution of a program makes use of a constraint satisfaction technique that searches for output values that match a given set of input values and satisfy a specified input-output relation.

Constraint satisfaction techniques are intimately connected with the notion of *dependency*. Dependency refers to the way in which changes to the values of some variables are linked with changes to the values of others. In the presence of a constraint, a constraint satisfaction technique establishes dependencies. A change to the value of a variable is accompanied by consequent changes to other variables needed to restore the constraints. Dependency maintenance is the mechanism by which changes to values of variables are propagated to dependent variables. Systems based on declarative constraints thus implement a form of dependency maintenance that is determined by the particular technique for constraint satisfaction that is used.

**Dependency in open development**: Closed-world dependency is motivated by trying to use the computer to automate problem solving. The essential idea is that people are good at identifying the global constraints to be satisfied in the solution of the problem, but not efficient in carrying out the computation and searching needed to find a solution.

Open-development dependency is motivated by trying to engage people in the exploration of constraints. This exploration can be carried out prior to any knowledge of global constraints and with no specific problem in mind. The formulation of problems and identification of constraints relies on experiment whereby people interact in situations to identify patterns of dependency. This dependency is much simpler in character than closed-world dependency.

The principal characteristics of open-development dependency will now be informally described. Modelling with definitive scripts (MWDS), which focuses on using a set of definitions to construct a model, is represented in this thesis as the appropriate framework in which to study open-development dependency. The justification for the epithet ‘open-development’ will be further discussed in Section 1.6.

The relationships that the modeller can apprehend in a situation involve recognising how the value of one variable (say $y$) depends upon the values of others (say $a$, $b$ and $c$). Such relationships are typically based on functional dependencies, as when $y$, $a$, $b$ and $c$ are related by the definition $y = f(a, b, c)$, where $f$ is a function that the modeller can identify through changing $a$, $b$ or $c$. The role of functional dependency in this context is similar to the role that it has in

relational database design[2], where it supplies the essential link between the modeller's experience and the way that attributes are grouped in tables.

In closed-world dependency, the relationship between variables is not in general a functional one. For instance, in finding values to satisfy a given input-output relation in Prolog there may be many solutions, or there may be none. Even where there is a functional dependency in a closed-world model, it may not be easy for the modeller to recognise this or to identify the function explicitly. The kind of dependency established by a declarative constraint is in general beyond comprehensive algorithmic analysis: it is undecidable whether there is a unique solution; there are many solutions; or there are none.

We identify open-development dependency with dependency that is established by explicit functional dependencies that define non-cyclic recipes for computing the values of dependent variables. A dependency of this nature is depicted in Figure 1-2. As is illustrated in Figure 1-2, and will be discussed in more detail in Chapter 2, such dependency can be represented in a 'definitive script'.

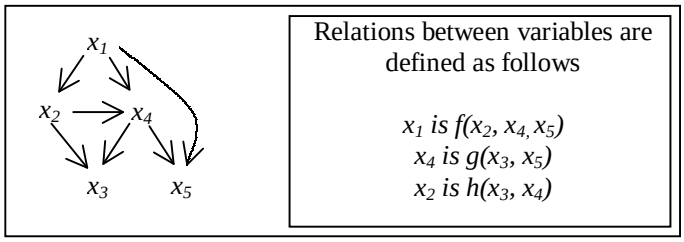


**Figure 1-2: A diagram to illustrate explicit acyclic dependency**

In Figure 1-2, arrows are used to represent the functional dependencies between variables, so that the arrows connecting $x_1$ to $x_2$, $x_4$ and $x_5$ correspond to the functional relationship $x_1 = f(x_2, x_4, x_5)$.

Acyclic dependencies of this kind are, in fact, sometimes used to build constraint satisfaction methods. The principle is that if we wish to change the value of a variable $X$ subject to a declarative constraint, we find an acyclic family of functional dependencies that define the new values of variables dependent on $X$. For instance, Figure 1-3 illustrates how various forms of acyclic dependency could be used to maintain the constraint $a+b+c=0$, when $a$, $b$, and $c$ have the initial values 0, -2, and 2 respectively.

[2] A functional dependency in a relational database occurs when the values of a set of attributes in a relation

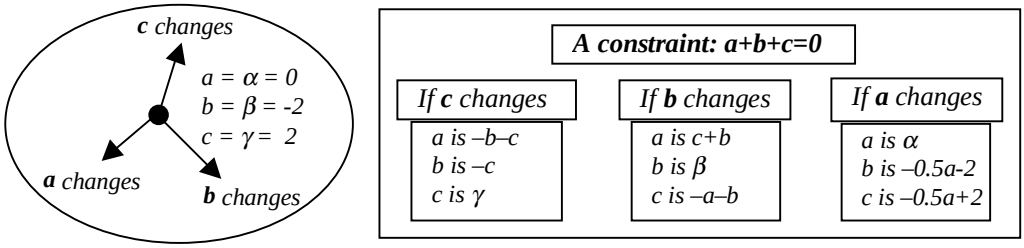


**Figure 1-3: Acyclic dependency to specify a constraint satisfaction method**

As a simple practical illustration of the use of an acyclic dependency in the maintenance of the constraint, consider the application of Ohm's Law ($V = IR$) to a simple electrical circuit comprising a battery and a light bulb. When the battery is changed, the new values of $V$, $R$ and $I$ are determined by the acyclic dependency

$$V \text{ is } V', R \text{ is } R', I \text{ is } V/R$$

where $V'$ is the voltage of the new battery and $R'$ is the current resistance of the light bulb. When the light bulb is changed the same acyclic dependency can be applied in a similar fashion. Far more sophisticated constraint solving techniques in a somewhat similar spirit are applied to the analysis of more complex circuits in Denneheuvel [Denn91, p. 79].

The applications reviewed in this chapter are all applications in which the emphasis is upon dependency in the open development sense. As we shall see, some of these applications try to combine the closed-world and open-development perspectives. This combination is not an easy task, and this is one of our motivations for studying modelling with definitive scripts.

## 1.2 Spreadsheets

The most familiar application of the principles associated with modelling with definitive scripts is the spreadsheet. The term 'spreadsheet' has a long history. Reference to a non-computerised version can be found in the first edition of Eric L. Kohler's Dictionary of Accountants[3]. A non-computerised spreadsheet refers to a worksheet providing a two-way analysis of accounting data [MattWeb]. In this accounting context, it was and is a large sheet of paper with columns and rows that lays out everything about transactions for businessmen to examine. It spreads or shows, for

uniquely determine the value of another attribute (e.g. {StudentID, Module} → Mark).

[3] 1952 – for recent editions see: W. W. Cooper and Y. Ijiri: Kohler's Dictionary for Accountants, Prentice-Hall, Inc.

example, all of the costs, income and taxes on a single sheet of paper for a manager to look at when making a decision [PowWeb].

The 'first electronic spreadsheet' - VisiCalc[4] - was invented by Bricklin and Bob Frankston in 1979 [BrickWeb] to create a program where people could visualise the spreadsheet as they created it. The metaphor of VisiCalc, at the time, was 'an electronic blackboard and electronic chalk in a classroom'. The data presentation in VisiCalc is like the paper spreadsheets but the data can be visualised and interacted with dynamically. The Heapsort model discussed in Chapter 6 has some similar characteristics. The visualisation of the array and tree with a value attached to each cell of the array and node of the tree is displayed on 'an electronic blackboard' where the basic relations between the visual elements are maintained through dependency. VisiCalc organises information into predefined columns and rows. The data can be 'added up' by a formula to give a total or sum. VisiCalc also has many special interactive features that support a WYSIWYG[5] environment such as displaying responsive results, which are instantly automatically recalculated based on formulae stored in the cells referencing other cells.

VisiCalc became an almost instant success and was believed to be a catalyst for the personal computer industry. At that time, only computer specialists could use the computers because of their complicated programming languages and interactions. Many people cite it as the thing that introduced them to the interactive possibilities of computers. It is also a tool to allow others to work out their ideas and reduce the tedium of repeating the same calculations [BrickWeb].

Spreadsheets are an offspring of VisiCalc. They inherited most of VisiCalc's features. They were first developed mainly to support financial, accounting and business users and are widely used in business for financial and related modelling [Bod86]. This is because spreadsheets provide features that enable even unsophisticated users to write programs by specifying formulas that establish numerical relations between data values [NM91]. The user's task is then to write a series of small formulas rather than the more difficult task of specifying the full control loop of a program as a set of procedures. Therefore the users can concentrate more fully on understanding and solving their problems.

---

[4] The name 'VisiCalc' is a compressed form of the phrase 'visible calculation'

[5] What You See Is What You Get

The use of spreadsheets as a modelling tool was restricted to financial applications in the early days. However because of their underlying principles and features such as definition, instant automatic recalculation and the 'program-by-example' interface, spreadsheets have become popular and familiar in other disciplines. For instance, in engineering, a spreadsheet has been used as the development environment for a dynamic force model of a manoeuvring ship [MS95].

Alan Kay, the originator of many notable ideas in computing[6], was one of the first people to point out that spreadsheets might be used for programming applications [Kay84]. Bell and Parr [BP93] take up Kay's agenda concerning spreadsheets for programming to demonstrate some hitherto unexplored capacities of spreadsheets by illustrating how to use spreadsheets to program 'Conway's game of Life'[7].

The extension of the spreadsheet paradigm to other computer science disciplines has been studied by a number of researchers. For instance, a system that allows both simple and complex graphical objects to be programmed directly using direct manipulation and gestures, in a manner that fits within the spreadsheet paradigm, has been developed by Burnett and Gottfried [BG98]. Penguims [Hud94], which allows interactive user interfaces to be created with little or no explicit programming, is an environment based on the spreadsheet model for specifying user interfaces. Davis and Kanet in [DK94] have developed an application-specific interactive system based on spreadsheets.

Spreadsheets have been extensively studied and used by many researchers because they offer distinguishing characteristics such as 'open modelling' which represents situations, allows meanings to evolve, and offers 'what-if' experiment. These illustrate the use of open-development dependency. Some spreadsheets also exploit closed-world dependency features (e.g. Excel includes Equation-Solver to find a solution for a particular user-defined constraint).

Researches concerned with exploiting the principles and potential applications of spreadsheets that relate to the open-modelling theme include [BG87, BP93, DK94, Nar95, Davis96]. The principal advantages and disadvantages of a spreadsheet, as identified by these researchers, are set out in Table 1-1.

---

[6] He is one of the inventers of Smalltalk and the architect of the modern windowing GUI

| Advantages | Disadvantages |
|---|---|
| Being an interpreter rather than a compiler | The flexibility of an interpreter over a compiler is offset by the slower speed of execution |
| It provides a necessary flexibility for the development of the model | Can encourage development without sufficient foresight, debugging can be just experimented hacking. |
| Spreadsheet's integrated working environment such as print, save, report. | Its non-procedural nature, which does not, for one thing, enforce structural discipline. For instance, unless care has been given to layout design and naming of cells a listing of cell formulas need not follow the order of their execution. This obfuscates the logic for third persons. |

**Table 1-1: The advantages and disadvantages of spreadsheets**

Like programming languages, spreadsheets are sophisticated tools with strengths and weaknesses and they are open to misuse. Many different researchers [Nar95, ISL95, Pan98, Green00] have reviewed the problems and difficulties encountered with spreadsheets from the user perspective. For instance, adding more formulae to an existing spreadsheet is very easy, but what Green [Green00] has identified as "the absence of any abstraction mechanisms, the poor role expressiveness and the pervasive hidden dependencies" encourage undetected errors and can make the inner workings of a large spreadsheet hard to grasp.

The concept of open-development dependency, and the use of definitions in particular, plays a crucial role in the success of the spreadsheets since it enables many kinds of real-world referent to be straightforwardly captured in a computer so that the user can interactively experiment with its representation (the 'what-if' feature).

## 1.3 Graphical modelling

Many graphical and geometrical languages and systems based on dependency have been devised. These include PIC[8] [Kern82] and PDL[9] [Wyv75]. Precursors for pure definition-based notations

[7] The game of life is essentially based upon a two-dimensional array, together with a whole collection of rules for how life is to develop. This means that situations in life can be represented well on a spreadsheet and that the program is much more concise than a solution written in a procedural language such as Pascal [BP93].

[8] A graphics language for describing simple diagrams or pictures

[9] Pictorial Description Language designed by G Wyvill in 1974

for computer graphics are to be found in the early research work of mathematicians Brian and Geoff Wyvill. The PDL language, developed by G Wyvill [Wyvil74, Wyvil75], is an early example of a definition-based notation for graphics (see Appendix A for a sample PDL script). In his doctoral thesis 'An Interactive Graphics Language' [Wyv75], B. Wyvill subsequently exploited PDL as the basis of an environment for interactive graphics. This thesis draws explicit attention to the advantages of using open-development dependency, for instance, in reducing the complexity of interaction in design and in integrating the roles of designer and user.

The application of dependency concepts is a recurring theme in the subsequent research of B and G Wyvill. By way of illustration, one important objective when designing a graphics system is to develop a good 'testbed', where experiments with new modelling and motion control techniques can be easily facilitated [CW89]. This requires an 'open' software architecture, so that programmers can modify or add code to a model in a dynamic interactive fashion. Such an architecture encourages the modeller to think of the scene not as fixed, but as changing over time. The model looks different at different times depending upon the bias of the interaction. A possible software architecture for integrating modelling and animation that implicitly exploits definitive principles is proposed by Chmilar and B Wyvill in [CW89]. Their system supports extensibility and the substitution of alternative representations for the same geometric object. It also allows multiple modelling primitives to be used in one scene or model since different modelling techniques have different strengths and weakness.

Open-development dependencies are also represented in other software for graphics. Jean [Jean87] invented an interactive graphical diagram editor that allows users to incorporate knowledge about the diagram through relationships or constraints between graphical objects. As a result, it can reduce the amount of work that the user has to do when changing a diagram. Once the user alters an object, the editor automatically alters any related object to maintain the pre-designed relationship. The applications discussed previously focus on defining the relationship between each graphical element explicitly. This helps a user to comprehend corresponding changes on a graphical figure made by his/her modification, and hence assists in further modification and design.

The use of declarative constraints in graphics, as pioneered by Sutherland in SketchPad [Suth63] and further developed in Juno [Nel85, HN94], offers a different perspective on a graphical model. In such a system, the positions of geometric elements are specified by equational constraints, and the response to modifying one geometric element is to invoke a

constraint satisfaction procedure to update the positions of all other elements, which is beyond a user's control. Modifying a graphical model, in this sense, may be limited by characteristics of a constraint satisfaction technique used.

The L.E.G.O. system designed by Norma Fuller is another example of an interactive graphics system in which object definitions are expressed in terms of geometric relations between object elements [FPR85, FP88, FP89]. The L.E.G.O system shows that using geometric constructions can eliminate the need for solving large systems of non-linear equations inherent in declarative constraint-based systems [FP88].

Definitive scripts discussed in this thesis offer a more explicit form of dependency maintenance, whereby the way in which a change to one geometric element affects other elements is explicitly prescribed – a theme explored by Richard Cartwright in his thesis [Car98]. Definitive scripts can be used to maintain simple equational geometric constraints (cf. Figure 1-3). For instance, both Jean and Cartwright discuss ways in which – in the framework of a definitive script – two points can be constrained to be a fixed distance apart.

## 1.4 Relational query languages

Edgar Codd, who is famous for his contributions to the theory and practice of database management systems, first introduced the 'relational model' in 1970 [Codd70]. Codd's conception is the basis for relational database management systems such as Oracle, Ingres, DB2, Access, Foxpro and Paradox. It provides an abstract theory of data that is based on mathematical foundations in set theory and predicate logic. Codd first proposed *tuple*[10] relational calculus, which served as a benchmark for evaluating data manipulation languages based on the relational model [Codd70]. His 'relational model' was conceived as a tool to free users from the frustrations of having to deal with the clutter of storage representation details [Codd79]. He attempted to protect users from having to know how the data is organised in the machine (the internal representation). However, the activities of the end-users should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. The relational model offers a way to link real-world semantics to computer representations in an intelligible and manageable fashion. In this context, functional dependency between attributes of entities plays a crucial role.

In a relational database system, each table is identified with a mathematical relation [Date89]. The relational model gives a prescription for the representation of data (by means of tables), and a prescription for manipulating that representation (by means of operators, such as selection, natural join and intersection). The relational model addresses three aspects of data: data *structure*, data *integrity*, and data *manipulation*. By using relations, users can describe an abstract organisation of data without knowing in depth about any additional structure for machine representation purposes. In this respect, relations serve as a high-level data language that yields maximal independence between programs, on the one hand, and machine representation and organization of data, on the other.

As the relational database model gained commercial interest, many proposals for relational database query languages were devised. Todd introduced the Information System Base Language (ISBL) as a relational database query language in 1976 [Todd76]. ISBL was intended to serve as a language meaningful to the user, in which relations are treated as named variables and the concept of dependency is exploited to ensure that data is represented and manipulated in a natural way. The Peterlee Relational Test Vehicle (PRTV) was Todd's prototype environment for ISBL. In its time, PRTV was unusual as a database system that provided flexible, interactive database support and functional extensibility. New relations could be created and assigned at will. PRTV provided flexibility in mapping between system files and user relations and allowed greater freedom in the storing of data.

ISBL is the principal medium through which the user accesses data in the PRTV system, and is designed for manipulating bulk data held in relations. It provides for variables, expressions, and assignments in much the same way that conventional programming languages do. All variables denote relations, and the only operations that can be used in expressions are those that produce relational results. There are standard features for entering and listing of relations and for the basic arithmetic and string operations. ISBL does not have flow or control statements such as **DO**, **WHILE** or **GO TO**.

ISBL is a relational database query language that makes use of the dependency concept. For example, given the relations BOOKS and LOANS, the expression:

```
FULL_LOANS = N!BOOKS * N!LOANS;
```

---

[10] tuple mean approximately the same as the notion of *a flat record instance*, introduced by Codd.

expresses FULL_LOANS as the natural join of the relations BOOKS and LOANS in such a way that subsequent changes in BOOKS or LOANS are reflected automatically in FULL_LOANS. In this respect, ISBL code can be regarded as a form of definitive script.

The applications reviewed so far all illustrate the concept of *single-user applications*, as introduced by Nardi and Miller in [NM91]. Such applications play a central role in the exposition of modelling with definitive scripts in this thesis. They are in effect examples of 'one-agent' systems (cf. Chapter 2) in which a single agent plays a key role in controlling every aspect of the modelling activity. Such an agent might be a financial manager or analyst (in spreadsheets), a designer (in graphical modelling) or a data modeller (in a PRTV database system).

## 1.5 Other definition-based applications

The previous sections have discussed the role of dependency as it has featured in three main single-user applications: spreadsheets, graphical modelling and database system. This section discusses a wide range of emerging applications in which dependency aspects have been embedded. This includes both academic and commercial applications.

- The **make** utility in Unix

**Make** is one of the original Unix tools for Software Engineering. A makefile is supplied as a parameter to the **make** command. The **make** command serves to maintain the relationships specified in the Makefile. For instance, the Makefile:

```
program.o : program.c
cc –c –o program.o program.c
```

expresses the fact that the target file **program.o** depends upon the source file **program.c**. Whenever **make** is invoked, the dependent file **program.o** is regenerated if there has been a change in a source file, using the Unix built-in **cc** command to compile a C program. In effect, make operates as a 'dependency maintainer' to update the Makefile as a 'definitive script'.

- Agentsheets

Agentsheets is an interactive environment developed by Alex Repenning [AgentWeb] at the University of Colorado at Boulder. It features a versatile *construction paradigm* to build dynamic, visual environments for a wide range of problem domains such as art, artificial life,

education, environmental design and simulation. The construction paradigm makes use of the agentsheet, which consists of a family of autonomous, communicating agents organised in a grid. The grid is used to define agents and their roles and each agentsheet serves as a design space. By way of illustration, Figure 1-4 depicts an agentsheet to represent a simple electrical circuit. The components of the circuit (**L**ight, **C**onnector, **S**witch and **B**attery) are represented by icons on the external display and by corresponding agents (L, C, S and B) in the internal representation. The rules that define the behaviours of the agents L, C, S and B implement the dependencies in the scripts shown in the figure and maintain the relation between internal and external state[11]. The rules governing each agent are specified in terms of the states of adjacent agents.

In the Agentsheets application, designers can incrementally create and modify spatial and temporal representations and can define the look and behaviour of agents specific to problem domains. The behaviour assigned to agents determines the meaning of spatial arrangements of agents (e.g. what does it mean when two agents are adjacent to each other?) and also the reaction of agents to user events (e.g. how does an agent react if a user applies a tool to it?). The Agentsheets system maintains dependency defined between agents (defined in a grid) in essentially the same way that the spreadsheet does.

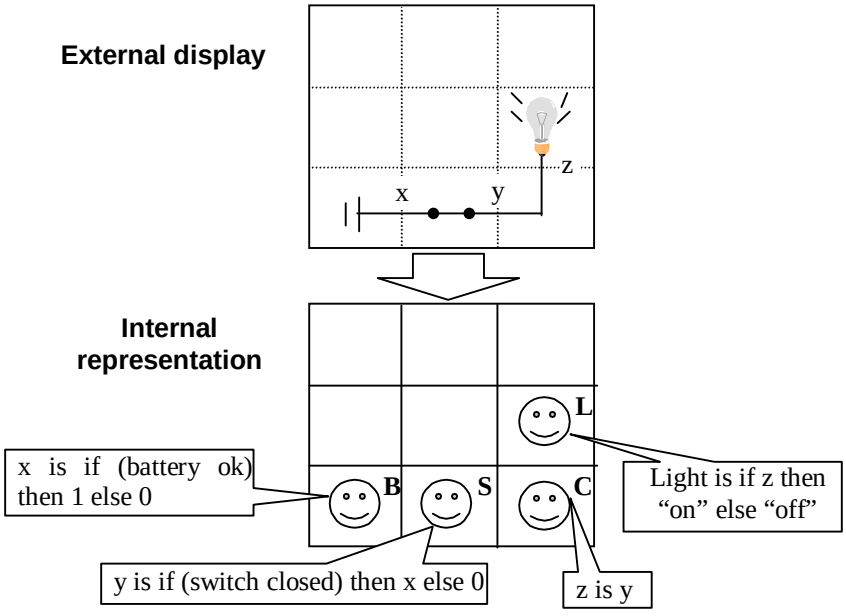


**Figure 1-4: A simple agentsheet**

[11] The annotations L, C, S, B, x, y and z, and the associated definitions are used to explain the essential operation of the model and do not feature in the agentsheet

Agentsheets postulates *participatory theater*, a human-computer interaction scheme combining the advantages of *direct manipulation*[12] (similar to spreadsheets) and *delegation*. This scheme offers a continuous spectrum of control and effort to the user. Users have maximal control over the components of their system through direct manipulation. With respect to the *theatrical metaphor*, direct manipulation interfaces are like hand puppets in the sense that users are completely in charge of the play. Extending the theatrical metaphor, tasks are delegated to actors by giving them scripts. Once a script has been given to an actor and the play has started, the audiences (or users) are left with no control over the play. This is because agents act according to the patterns of inter-dependency that have been pre-defined by the modeller. In this approach, actors in the participatory theatre will act according to their script unless the audience or user tells them to do something different.

- MSWord automatic formatting style and drawing tool

The style option in MSWord, which allows editors to define their own textual style, can be regarded as one form of definition. Once a style (e.g. normal) is changed, all text marked as normal will be automatically updated with correspondence to this change.

The grouping facility in a drawing package is another aspect of using dependency. Each drawing component is created as an individual object but components can also be grouped together. After grouping selected objects, the positioning and scaling of these objects become interdependent. When enlarging the grouped object, each object is scaled up with a consistent proportion. It can be seen that there is a use of dependency in grouping.

- Visio

Visio is a tool, developed by Microsoft, to help in visualising complex concepts. It is promoted as the business-diagramming standard, which empowers effective visual communication, and provides a common visual language that everyone across the enterprise can use to communicate more effectively. Instead of just providing text-based documents, Visio offers a variety of kinds of diagrams to clarify text documents [VisioWeb].

[12] The term 'direct manipulation' was coined by Shneiderman [Shne83] who describes three principles of direct manipulation systems: continuous representation of the objects of interest; physical actions or presses of labelled buttons instead of complex syntax; and rapid incremental reversible operations whose effect on the object of interest in immediate visible.

The components (e.g. text boxes and flowcharts) are connected by a dependency, for instance, a text box and an arrow is linked, so that moving the box will cause moving an arrow. Visio diagrams can be linked to databases so that information is synchronised between the programs directly.

Visio, moreover, provides a facility in which users can modify shapes or create customised ones to suit organisational or personal needs. Visio allows the user to define shapes in a form of 'definition', as shapes can be edited and saved in a template. Intelligence can also be added to shapes by writing formulae, as in a spreadsheet. Shapes can also be made data-aware or their behaviour modified in other ways. For example, a door shape can 'know' when it is hinged to a wall shape and rotated appropriately. A file cabinet shape has a drawer that pulls out to show clearance. A network equipment shape includes properties for the manufacturer and other information with sample data already assigned.

- OLE (Object Linking and Embedding)

OLE is a technique for creating a compound document[13] developed by the Microsoft Corporation [OLEWeb]. It features embedding and linking objects. Each element in the compound document is stored in such a way that it can be manipulated by the application that created it. OLE allows users to mix different forms of expression rather than artificially separating them. It enables the creation of objects with one application, which can then be linked or embedded in a second application. Embedded objects retain their original format and links to the application for which they were created. The concept of linking or embedding with OLE is similar to dependency. Embedded objects always link to their original applications, which is similar to a dependency concept.

## 1.6 Modelling with definitive scripts (MWDS)

The applications reviewed above all feature uses of dependency in which human involvement and interpretation have an essential role. The interactions with spreadsheets, geometric models and databases that we have discussed are guided in a dynamic way by observation of an external

[13] A compound document is a document that contains elements from a variety of computer applications such as, a single compound document including text from a word processor, graphics from a drawing program and a chart from spreadsheets applications.

situation[14]. Our particular interest is in those interactions where the modeller is uncertain about the outcome, not only in the familiar sense of 'what-if' experiment – where the framework for interpreting interactions is typically well established, but where interaction has a genuinely open character – as when a spreadsheet is first being constructed, or exploratory geometric modelling is involved.

The concept of modelling with definitive scripts (MWDS), as introduced at Warwick in the design of the ARCA notation in 1983, is a primary ingredient of what is now known as the Empirical Modelling (EM) project. As a technique, MWDS had been anticipated by several of the applications discussed above, but its significance within EM stems from the alternative to a closed-world semantic framework that it can be viewed as offering. The primary aim of studying MWDS is to account for situated modelling – of the kind represented in applications that we have reviewed – where new observables are introduced, and new interpretations are invoked in ways that have not been preconceived. As will now be discussed, the special characteristics of MWDS that support open development relate to the close connection that can be established between experience of interactions with a computer-based model and of interactions with the external situation to which it refers. On this basis, MWDS can be regarded as the archetype for open-development modelling.

We should acknowledge at this point that the narrow characterisation of open development, the close association of open development with MWDS, and the sharp distinction between closed-world and open-development approaches may seem inappropriate. Of course, in computing practice, there are many contexts in which new observables are introduced and new interpretations invoked without the conscious adoption of alternative principles. Such contexts arise, for instance, in the use of prototyping techniques in software development, the development of formal specifications to support exploratory modelling, and the construction of mathematical models (perhaps using explicitly programmed dependency) using Mathematica [MathWeb], all of which activities have features in common with MWDS. The discussion of such activities is outside the scope of this thesis because they are far more sophisticated than primitive MWDS. As a loose but useful parallel, the distinction between informal empirically-guided computer-based or computer-related modelling and MWDS is similar to that between

[14] The term 'external situation' is used here to refer to the wider context, outside the computer model itself, that provides the experiences, possibly stemming from the modeller's imagination, that actively inform the ongoing modelling activity.

what colloquially is understood to be an experiment and what a scientist would regard as a 'scientific experiment'. In representing the latter distinction, it is easier in the first instance to describe the characteristics of 'scientific experiment' without reference to the status of informal experiment. It is also apparent that an informal experiment can only be viewed as scientific if it conforms to certain principles and practices that might at first seem to restrict rather than enhance the experimenter's power to discover knowledge.

**State in closed-world and open-development modelling**

In building a computer model, a modeller's interaction with an external situation has both closed-world and open-development aspects[15]. The closed-world aspect is associated with theory-style knowledge about the global properties of the objects with which he/she interacts. Such knowledge is appropriately formulated using declarative constraints. The open-development aspect is associated with knowledge about what kind of pattern of local changes is encountered when interacting in the world or can be exploited when constructing a computer model. As discussed in section 1.1, our aim is to show that knowledge of this kind can be appropriately formulated using explicit functional dependencies (cf. Figure 1-2).

The closed-world and open-development aspects of modelling are related to the modeller's experience in quite different ways. Theory-style knowledge requires a deep understanding of the situation that is usually acquired through extensive experience (possibly on the part of the theory builder rather than the modeller) prior to modelling rather than during the model building. Knowledge about patterns in local changes of state is acquired through confronting a particular state and – through experiment – refining our expectations of what will happen in response to a particular interaction. Expectation, in this context, is expressed in terms of how we expect a change in the value of one observable to affect the values of other observables indivisibly. This kind of knowledge is essentially different in character from absolutely reliable knowledge, in that it is accessed only through interaction and is subject to empirical validation. Every time we act, our expectations about changes of state are being put to the test – they could be confounded.

[15] The distinction between closed-world and open-development modelling that is developed in this chapter leads to a distinction between two meanings for the term 'computer model'. Closed-world modelling is associated with traditional formal mathematical models, whilst open development leads to models that are more similar to the physical artefacts that a designer, artist or engineer might construct.

The above discussion shows that, in computer-based modelling for open development, it is critically important that the modeller can recognise latent state-transition patterns in the computer model and associate them with an external situation. This focuses our attention on the way in which states and transitions are implemented and interpreted in closed-world and open-development dependency models. In broad terms, a state in such models is associated with an assignment of values to the relevant observables (viz. $x_{i1}$, $x_{i2}$, …, $x_{in}$ in Figure 1-1 and $x_1$, $x_2$, …, $x_5$ in Figure 1-2), and transitions[16] are associated with the technique adopted for dependency maintenance.

In a closed-world dependency model, the states – in the strict sense – are associated with possible solutions that satisfy the constraints (cf. Figure 1-1, where the possible states are the values $X_1$, $X_2$, $X_3$, …, $X_k$ that satisfy the constraint $\Gamma$). A transition from one state to another (e.g. from $X_1$ to $X_2$ in Figure 1-1) is automatically performed by the system using a constraint satisfaction method. Strictly speaking, any intermediate state associated with this transition cannot be interpreted in the closed-world model. This will suit a modeller who does not wish to be involved in the constraint satisfaction activity and who may not even comprehend what is going on in the transition. For such a modeller, transitions in the closed world are like jumping from state to state.

In using a closed-world dependency model, the interpreted states are not always strictly restricted to possible solutions to declarative constraints (cf. Figure 1-1). Consider, for instance, a logic programming environment, where constraint satisfaction is used to determine a solution state $(x_1, x_2,\ldots, x_{s+t}) \equiv (i_1, i_2,\ldots, i_s, o_1, o_2, \ldots, o_t)$, for given values of the inputs $i_1$ through $i_s$. Because of the undefined values, the initial state of the program, in which the inputs have been specified and the outputs are yet to be determined does not correspond to a solution of a constraint. Despite this, all the assignments of values to $x_1$ through $x_t$ that are made during the execution of the constraint satisfaction method are interpreted. They correspond to the states of the executing logic program as it passes from its initial state to its final state. Where declarative constraints are used to frame programming activity in this way, the modeller (who is here the logic programmer) must have essential knowledge about these states. For instance, in Prolog, it is necessary to use the 'cut' [CM87, p. 75] in debugging and optimisation. Similarly, in Juno-2, the *double-view editor* is designed to display the program or constraint that draws the picture as well

[16] In this context, a transition is interpreted as atomic, involving a single state-changing step

as the picture itself. However, although the modeller may be able to understand the constraint satisfaction mechanism used to maintain the closed-world dependency (cf. Figure 1-3), the transitions are primarily under the control of the mechanism rather than the modeller.

Transitions in an open-development dependency system are different from those in the closed world since they can be explicitly comprehended and are under the open-ended control of the modeller. Open-development dependency maintains relationships between observables in a less constrained way than closed-world dependency. This means that the value of an observable can be changed and a functional dependency modified by the modeller in an exploratory way (for instance, as in 'what-if' activity in spreadsheets). In this respect, the way in which the modeller experiences and interacts with the computer model is well matched to the way in which the modeller acquires knowledge about patterns of state change in open development.

In open development, the correspondence between atomic transitions in a state represented in the computer and atomic transitions in the state of the external world to which it refers is highly significant. In transitions from state to state in closed-world modelling, the modeller only has to be able to interpret the initial and final states, but in open development, there is not only an abstract correspondence between transitions in the model and transitions in the external world, but also a correspondence between each transition in the model and its counterpart transition in the external world *as experienced by the modeller*. This can be interpreted as saying that observables in the external-world referent are directly reflected in the computer model and that there are direct counterparts in the computer model for changes to observables in the external world.

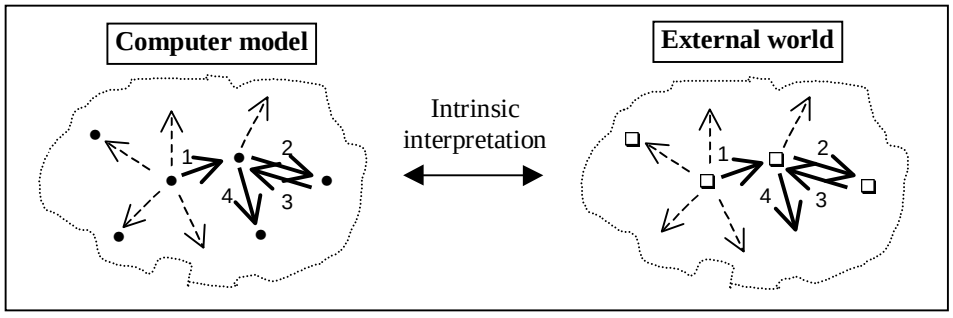


**Figure 1-5: The interpretation of state in open-development modelling**

The most significant implication of open-development modelling is that it enables an association between states of the computer model and external-world states that, being based on the perceived close correspondence between possible atomic transitions, relies upon intrinsic and

local knowledge of interaction (cf. Figure 1-5). In Figure 1-5, the dashed arrows represent possible transitions between states in the computer model and between states in the external world. The transitions labelled 1, 2, 3 and 4 represent a possible sequence of interactions by the modeller. The transition labelled 2 in the external world might represent an experimental interaction. The transition labelled 3 in the external world might resemble the action needed to restore the experimental context, and the transition labelled 4 the repetition of the experiment.

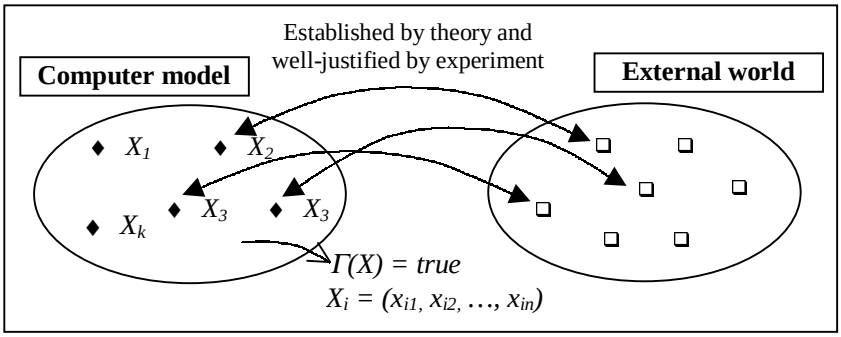


**Figure 1-6: The interpretation of state in closed-world modelling**

The dotted boundaries in Figure 1-5 indicate the open character of the space being constructed and explored by the modeller. The interpretation of the state of the computer model can evolve to reflect experience and change in an external situation as they develop. This means that open-development modelling offers the possibility of dynamic re-interpretation of the computer model as artefact. In contrast, the interpretation of a state in a closed-world model, once initially established, is fixed. In Figure 1-6, $X_i$ represents a state that is associated with a set of values for the variables $x_{i1}, \ldots, x_{in}$ that satisfies the predefined constraint $\Gamma$, and these variables correspond to observables in the external-world state in a preconceived way that has been well-justified by theory and experiment. In this way, the semantics of state in the closed world is defined with reference to an entire state space, rather than state-by-state.

**Empirical Modelling (EM)**

The applications of open-development modelling, and the relationship between closed-world and open-development modelling are central themes of the Empirical Modelling (EM) project. EM generalises the modelling principles represented in the applications reviewed in previous sections, providing a framework for computer-based modelling based on three key concepts: **observation**, **agency** and **dependency**. EM refers to a situated modelling activity in which the computer-based model, the modeller and the situation stand in a special relation. The character of the modelling activity is as depicted in Figure 1-5 and involves creation and discovery on the part

of the modeller. This gives key roles to the modeller's understanding and to observation and experiment in the external world. The significance of the computer model cannot be appreciated in isolation: the actual experiences of the modeller, as expressed in knowledge of interactions and expectations, are the essential part of the design.

As is illustrated in Figure 1-2, the dependency used in open development can be naturally represented by a definitive (definition-based) script (cf. Section 2.1). The tools developed under the EM project include definitive notations and dependency maintenance systems (cf. Section 2.1.2) that provide a means for a modeller to construct such scripts. The construction of computer-based models using open-development dependency based on definitive scripts is a major theme of this thesis.

The use of a definitive script to represent acyclic dependency (cf. Figure 1-2) provides the modeller with a rich mechanism to manipulate patterns of dependency. As illustrated in Figure 1-7, we can redefine a variable by a value or a formula, introduce a new definition and delete a definition. The interpretation of modelling activity with a definitive script is also very open-ended. It can represent various interactive activities such as changing the value of an observable, understanding the pattern of dependency through experiment, deleting a constraint, introducing a new observable and adding a new constraint. By way of illustration, Figure 1-7 shows how the value of an observable can be arbitrarily changed (as *b* is changed from (1) to (2)); how a dependency can be introduced (as *b* is changed from (2) to (3)); how a dependency can be eliminated (as *a* is changed from (3) to (4)) and how a new observable can be introduced (as variable *d* is introduced in (5)).

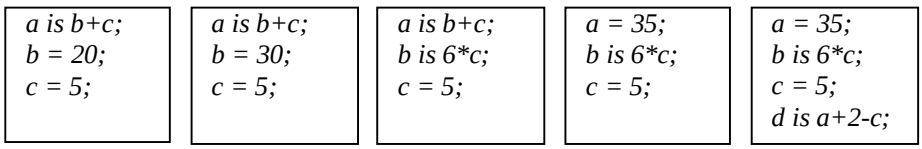


**Figure 1-7: Scripts to illustrate modelling activity in open-development sense**

This modelling approach enables the modeller to take account of new observations and hitherto inconceivable states in the external situation in a way that respects the ongoing exploratory character of open development. (In contrast, if strictly interpreted, closed-world modelling deals with novelty in the external situation by replacing one closed-world model by another.) The model can be initially constructed with no specific goal and then experimentally

refined and modified until it satisfies the modeller's needs. As Russ points out (cf. [Russ97]), this is particularly useful in areas where there is no adequate theory):

> "In such areas we may wish to build models simply in order to aid our understanding; any specific purpose may be unknown, or provisional, and it is then only an impediment to make early commitments to certain properties we wish to preserve in the model."

Open-development modelling is essentially concerned with the way in which the modeller observes, perceives and interprets the world. This perception and interpretation, typically based on a modeller's experience, may be altered by a shift in perspective possibly resulting from a change in the modeller's role or understanding of the situation, an interaction with a model or a change in the external situation. There is no fixed interpretation for such shifts in perspective, which can either be viewed as originating from the modeller ('The behaviour of the world is different from what I thought it was') or from external factors ('The world is now in a different state').

A major theme in MWDS is the way in which patterns of interaction with a definitive script can reflect different roles for the modeller as a state-changing agent. For instance, the redefinition of the value of an observable may signify a change in the state of the artefact ('modeller as *user*') or the redesign of the artefact itself ('modeller as *designer*'). The use of a definitive script is not confined to representing the interaction of external agents such as designers and users with an artefact – it can also extend to the representation of internal agents that are the components of the artefact itself. The essential principle is that patterns of dependency amongst observables are seen as characterising the perspective of any state-changing agent. The potential for combining open-development dependency with agency in this spirit is illustrated in Agentsheets, where the states of neighbouring agents supply the observables of a particular agent and dependencies can be defined and arbitrarily redefined to reflect changes in this agent's perspective (cf. Figure 1-4).

MWDS focuses on using a script to represent state as experienced and observed, not just on using definitions as a programming device. It is connected with a wide range of modelling activity. A model built based on MWDS is exploratory, unpredictable and hand-driven: human interaction is essential. People can change the state of the model by redefining the value of variable, change a pattern of transitions by introducing a new definition (or a definitive script) and bring in new observations by including a new set of definitions. The outstanding characteristics of such models are broad interaction; comprehension via interaction; openness;

flexibility; easy modification; fast prototyping; learning by doing and experienced-based design. Throughout this thesis, the characteristics of MWDS are studied by reviewing many models. These models illustrate its essential semantics, its wide range of potential uses and the principles and techniques it offers to support modelling activity.