# Chapter 1

# Introduction

This thesis is motivated by my work for the Empirical Modelling (EM) research project, with which I have been closely associated for six and a half years. Over this period, I have witnessed a significant development in the scope of application and degree of student involvement in the project. This has been represented in the completion of eight PhD theses on a wide variety of themes, the emergence of a new fourth year undergraduate "Introduction to EM" module and some 40 third year undergraduate projects. I have had an explicit role through the co-authorship of several papers relating to applications [BCSW99, BWM+00, BCH+01, BRWW01, BBRW03] and the development of the models *catflapWard1997*[1], *emhttpdWard1999*, *musicscoreexptWard1999*, *vcgWard1999*, *oasysprivilegesWard2000*, *definitivedmWard2001*, *lsmpresentationWard2001*, *backroomWard2002*, *blankpresentationWard2002*, *introtoempresentWard2002*, *sqleddiWard2003*, but yet more significant has been my role as the principal developer of EDEN[2], the primary tool of the EM research group, and as a major contributor to the management and dissemination of documentation and resources [WRB, Warb, Warc]. In particular, my extension and debugging of EDEN has contributed significantly to what is possible with the tools, and many of the more recent models could not have been constructed without it.

[1] Throughout this thesis, text of the form *projectAuthorYear* refers to the unique key name of a project which can be found in our 'empublic' archive [WRB]. Further information about a project can be found at `http://empublic.dcs.warwick.ac.uk/projects/`*keyname*.

[2] From version 1.0 for Solaris in October 1999 to version 1.50 for Solaris, Linux, Windows and Mac OS X in March 2003 — see §4.2.

My experience of EM technical support and consultancy has led me to identify the *study of dependency* as one of the central issues in enabling EM. It has also informed a perspective on the current status of EM principles and tools that has both favourable and critical aspects:

**Favourable** — EM leads to a different quality of human-computer interaction, characteristic of "thinking with computers" [BR]. It has also become clear to me that EM also potentially provides a radically new approach to software development. Its distinctive nature gives support for change, comprehension and reuse. The use of EM principles may offer a promising route to software that is more dependable and intelligible. These qualities have both motivated and been informed by my practical work on the EM project, which has involved the development of parts of EDEN "in itself", case studies in the use of dependency, and the development of new techniques for the maintenance and management of models.

**Critical** — My engagement with EM tools and models has also given me a complementary awareness of problematic issues that is deeper than that of a casual EM model builder. Designing and implementing EM tools exposes the tensions between human interpretation and machine implementation, challenges in documentation and version control and the difficulty of representing certain types of dependency.

My perspective on EM motivates the two main questions addressed in this thesis: How can we best exploit and develop our existing tools for implementing EM activity? What prospects are there for better tools in the future? The key issue in addressing these questions is the implementation of dependency on digital computers.

As a "second generation" thesis[3] on Empirical Modelling, this thesis assumes a considerable body of underpinning knowledge about EM that cannot be made explicit within its scope. Because the bulk of the thesis is concerned with implementation issues, its focus is necessarily somewhat internal to the EM project. This accounts for the extended introduction that follows, which not only serves to introduce the

[3] And possibly the first such thesis.

technical contribution of the thesis but to provide essential EM background and to situate my research in relation to external literature.

The next four main subsections consider EM in relation to philosophy, applications, development and implementation. With specific reference to *dependency*, the key concept in this thesis, these subsections are respectively concerned with "dependency in concept", "dependency in application", "dependency in development" and "dependency in engineering". Subsections describing related work, the thesis aims and outline then follow.

## 1.1 Dependency in concept: EM, Radical Empiricism and the making of meaning

Dependency is one of three primary concepts used in Empirical Modelling. This section explains these concepts, firstly without discussing their relationship to digital computers. The explanations inform the notion of *meaning* that is applied in this thesis. The concept of dependency plays a major part in the making of meaning.

The concepts of Empirical Modelling are based on a world-view similar to the Radical Empiricism of William James [Jam12]. Traditional empiricism is "the view that experience, especially of the senses, is the only source of knowledge" [her00]. Radical Empiricism goes further than traditional empiricism in its recognition that: "the relations that connect experiences must themselves be experienced relations, and any kind of relation experienced must be accounted as 'real' as anything else in the system" [Jam12, p.42]. To paraphrase Wild's discussion of Radical Empiricism [Wil69], we must dwell on direct experience, vague and subjective though it is, and attempt to use concepts to clarify and express the implicit meanings present within it. When we approach a new problem, we must begin all over again, letting the direct experience speak, without forcing it into our prior established categories. We may have our own systems and conclusions, but must be ready to examine any new fact at any time and make the necessary revisions and corrections. There is always potentially more to learn, so facts have an element of 'mystery' and all conclusions are tentative [Wil69, pp.413, 390, 394–5].

Most scientific disciplines "begin with what is known by direct acquaintance",

but then "each of them leaves this behind, in order to turn to the special objects of its field, and to deal with them in as objective a way as is possible." [Wil69, p.413]. For example, the discipline of Computer Science is largely concerned with the study of the Turing Machine, an abstract fictional object.

The Radical Empiricist world-view is appropriate in many fields concerned with analysis of phenomena. Accident investigations are one example. Section §4.1.6 illustrates how Empirical Modelling can be applied to the scenario of the Clayton Tunnel railway accident of 1861, as it is described in [Rol82]. The descriptions of the accident given by each participant are based on their situated, personal experience. Put together, the descriptions contain many gaps and logical inconsistencies. However it is still possible (in this case) to come to tentative conclusions about the probable cause of the accident. Further details are given in [Sun99, Bey99, BS99].

A simple type of analysis that can be performed in this kind of domain is illustrated by the 'Cause and Effect' diagram invented by Kaoru Ishikawa [IL85, p.63], nicknamed the 'fishbone' diagram due to its shape. Such a diagram is used to assist the identification of causal factors that contribute towards an undesirable quality characteristic (effect) observed within experience. The effect that is the subject of analysis is first recorded at one side of the diagram, then the 'backbone' of the fish is drawn and a systematic attempt is made to conceive potential causes. (When used in a manufacturing setting, categories of Man, Method, Material, Machine and Measurement are often used to assist in completeness.) Figure 1.1 contains an example (reproduced with permission from [Mor]) which shows possible causes of "missed free-throws" in a basketball tournament.

Analytical reduction, where problems are divided into distinct parts and then examined separately, is a common Computer Science technique for dealing with complexity. Checkland [Che99, p.59] states the limits on the appropriateness of reductionism.

> Descartes's second rule for 'properly conducting one's reason', i.e. divide up the problems being examined into separate parts — the principle most central to scientific practice — assumes that this division will not distort the phenomenon being studied. It assumes that the components of the whole are the same when examined singly as when they are playing their part in the whole, or that the principles governing the assembling of the components into the whole are themselves straightforward.

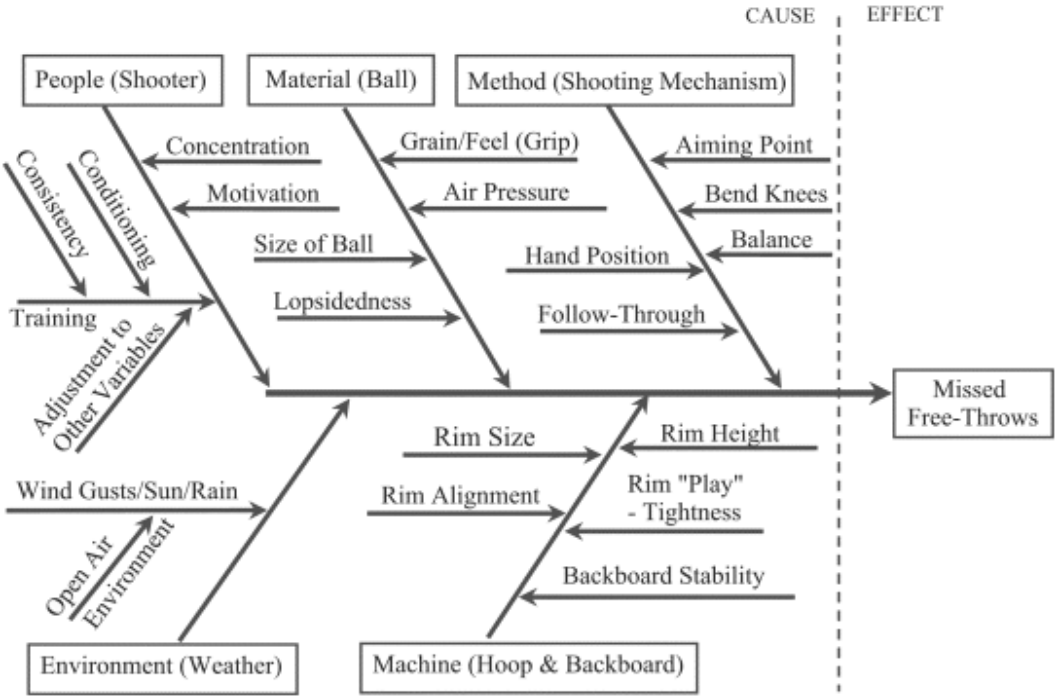



Figure 1.1: An Ishikawa fishbone diagram (reproduced with permission from [Mor])

Checkland's observation suggests that analytical reduction is not appropriate in the context of an analysis of cause and effect, where an initially holistic approach is required.

Rather than analytical reduction, Empirical Modelling is based on the identification of Observables, Dependency and Agency within experience that is initially personal and subjective.

- An *observable* is a characteristic of my environment to which I can attribute an identity. An observation of an observable returns a current value.
- A *dependency* describes how I perceive the act of changing one particular observable to change other observables predictably and indivisibly.
- An *agent* describes my perception of an entity (a cluster of observables) that is capable of initiating state change. I attribute all changes to the values of observables to agents.

(Beynon *et al* [BCSW99] gives an explanation of these concepts, as understood

within a system domain. It gives a more 'operational' explanation of the concepts, which I will return to in §5.1.1.)

Empirical Modelling is the name we have given to the activity of building artefacts to embody patterns of observables, dependencies and agent actions that are encountered in experience. EM involves the progressive development of understanding through interaction, whereby meaning is continually refined in the light of additional experience.

The use of artefacts for development of understanding is particularly relevant to the experimental activity that precedes the formulation of a theory. In this respect, it is related to the concept of 'construal' introduced by Gooding in his philosophical analysis of experimental procedures in science [Goo01]:

> ...I have labelled interpretative images and their associated linguistic framework as 'construals'. This term denotes proto-interpretative representations which combine images and words as provisional or tentative interpretations of novel experience.

The most primitive understanding of state change is based on pure agency. For instance, in the absence of alternative explanations, in some situations, we are prone to regard change as stemming from autonomous action. This is illustrated in the way in which the Elizabethans attributed change to agents with different levels of privilege to enact state change within the Chain of Being, beginning with God and passing to the earthworm via planets, royal personages and peasants [Til43]. Dependency is associated with a recognition that one change *entails* another. As we start to perceive dependencies within experience, we move from the perception of independent change that is represented in its most extreme form in animism[4] towards a view that presumes more predictability and interconnection.

The personal pronoun is used advisedly in the definitions of observable, dependency and agent above. Characterisation of experience in terms of observables, dependency and agency is initially a private, personal matter as it is based on one's own experience. EM follows Radical Empiricism in taking private experience as primary. Regions of stability are rare within the totality of experience, and so only a relatively small part of experience is associated with stable public 'theoretical' knowledge. The learning activities associated with the transition from pre-verbal

[4]The term animism denotes the belief that a soul or spirit exists in every object, even if it is inanimate. It was controversially linked with religion by E.B. Tylor in 1871.

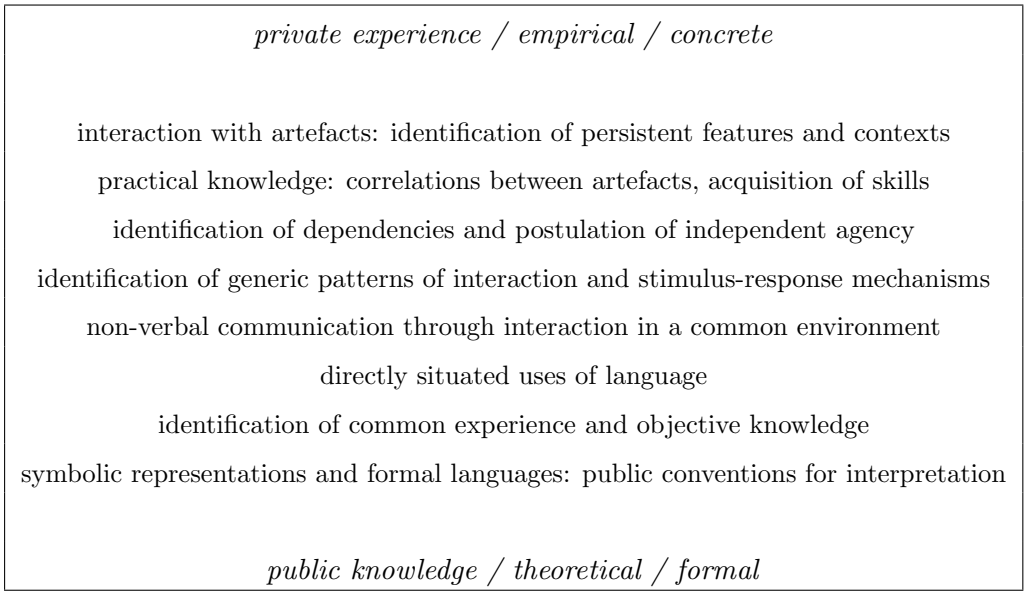



Figure 1.2: The Empiricist Framework for Learning (from [Roe03, p.74])

private experience to public knowledge are set out in the Empiricist Framework for Learning (Figure 1.2) [Roe03].

We use empirical investigation, then, in order firstly to gain private understanding that we may then attempt to make public. The development of understanding (private or public) may be our sole aim, or we may wish to *use* the understanding to make predictions — in particular, to examine the potential consequences of a particular action.

But many experimental environments are 'noisy' — there are many agents operating and it is difficult to distinguish the effects that arise indivisibly from one's own actions from those that arise from the actions of other agents [BNR95]. In some environments (for example, in the train accident scenario), experimental interaction may be dangerous, or the time between stimulus and response may be excessively long, or the values of observables may be hard to determine. These are some motivations for the use of modelling. We construct an environment to model the *referent*, and take measures to make the modelling environment 'quieter', safer, swifter in response and more perceptible[5] in observation than the referent, as appropriate.

[5]In this chapter, I sometimes use the word perceptible to qualify the word observable, meaning that the modeller can determine its value *directly*, rather than through some inference, in the spirit of Radical Empiricism.

A 'quiet' environment is one in which the modeller is the sole agent, providing a tractable context in which all changes to values of observables are due (directly or indirectly) to the modeller's interaction. We refer to a model of such an environment as a *1-agent model*.

The machine initially developed by Bill Phillips in 1949–50 to demonstrate the circular flow of income of the open-economy IS-LM model [Bar00, p.103] is an example of a 1-agent model [Phi00, p.68]:

> Fundamentally, the problem is to design and build a machine the operations of which can be described by a particular system of equations which it may be found useful to set up as the hypotheses of a mathematical model, in other words, a calculating machine for solving differential equations. Since, however, the machines are intended for exposition rather than accurate calculation, a second requirement is that the whole of the operations should be clearly visible and comprehensible to an onlooker.

The Phillips machine used the flow of water through transparent pipes and the gathering of water in reservoirs as a metaphor for the flow of income (see Figure 1.3). As the quote describes, the hydraulic solution was chosen for pedagogical reasons. Water flowing in transparent pipes made the observables perceptible — the possible alternative, electronic computers, at that time had no visual display units. The Phillips machine was a 1-agent model (leaving aside the pump that forces water to the top of the machine so it may cascade down) that was safe (leaving aside the possibility of leakage) and allowed experiments to be run faster than real time.

On what basis does the flow of water in the Phillips machine model represent the flow of income? Water and income are experiences of a very different kind. The experience of the water is said to represent the experience of income in respect of the modeller's pragmatic conventions for interpretation. This concept once again has a precedent in the work of William James, who discussed "the ways in which one experience can function as the knower of another"[6]. If, in the view of the modeller, experience of the model relates to experience of the referent, then the modeller may regard the model as *meaningful*.

EM supplies environments similar to that provided by the Phillips machine, where meaning can be negotiated through interaction [BS99] with the model, interrelated with other situational, explicit, mental and internal (SEMI) aspects of state

[6]This aspect of James's work is related to EM in [BS99].


Figure 1.3: The Phillips machine in the London Science Museum, with the author shown for scale. Note that the machine has been drained of water for preservation purposes. Photo: Mark Lloyd.

in the mind of the modeller [BRWW01]. The meaning comes about through the confluence of modeller, the model and these aspects of state — it is not a property of modeller or model alone. This, then, is our interpretation of the word 'meaningful'. It may seem a rather 'informal' definition, but our intention is to circumvent the difficulty of formalising personal understanding by grounding meaning in direct experience, in the manner of the Radical Empiricism of William James.

## 1.2 Dependency in application: Interaction with Meaningful State

This section briefly reviews digital computer artefacts that exploit dependency similar to that represented in Empirical Modelling to enable *interaction with meaningful state*.

Spreadsheet programs[7] are the most common example of an experimental environment similar to the Phillips machine, implemented on a digital computer. If a spreadsheet model of aspects of Keynesian economics is constructed (see below), the resulting spreadsheet provides a 1-agent environment in which experiments with the modelled economy can be performed safely and faster than real time. At all times, the values of spreadsheet cells are perceptible. The cell values however require more interpretation on the part of the modeller to form meaning than is required with the Phillips machine. The raw digital numeric values displayed by a spreadsheet do not provide the direct perception of absolute values and rate of change that the analogue display of water levels in the Phillips machine gives[8].

A spreadsheet program provides an environment that also enables interaction beyond simple use. A spreadsheet is constructed incrementally through progressive interaction. At any stage, 'what-if?' interactive experiments can be performed on the partially constructed spreadsheet in order to determine its response to a particular stimulus. If the response of the spreadsheet corresponds to that experienced

[7] Of which the first example was the 1979 [CKA96, p.251] VisiCalc by Bricklin and Frankston [BF].

[8] This deficiency can be rectified somewhat through the use of graphing facilities in the spreadsheet — but current spreadsheet programs do not provide the facility to create a graph as interaction proceeds over time.

with the referent, then the spreadsheet will be considered to have meaning[9]. The spreadsheet can always be extended or modified by adding or changing formulae, so it may never be considered 'complete'.

These properties of the spreadsheet environment that enable interaction for construction as well as use are also to be found in the Phillips machine, but construction and modification of the Phillips machine would require significantly more specialised engineering knowledge than that required to use the machine. In contrast, a spreadsheet may be constructed or modified, using interaction techniques that are the same (in many cases) as those used for spreadsheet use.

Database environments are another common example of software that attempts to model a referent — perhaps the reservation status of seats on a particular planned future aeroplane flight. Much effort has been expended in attempts to formalise the meaning of data held in databases in an objective way, independent of direct experience. The major problems such attempts encounter are illustrated in Kent's Data and Reality [Ken78], which contains many examples to support his hypothesis that formal modelling is inadequate:

> ... there is probably no adequate formal modelling system. Information in its 'real' essence is probably too amorphous, too ambiguous, too subjective, too slippery and elusive, to ever be pinned down precisely by the objective and deterministic processes embodied in a computer.

Both spreadsheets and databases emphasise interaction with meaningful *state* rather than automation of *transitions*. More examples of other computer applications with the same emphasis include word processors and music creation packages.

Ten principles to guide the designer in implementing interaction with meaningful state are stated in [app87]; they include recommendations to use metaphor, provide stability, direct manipulation and feedback. For example, the original Apple Desktop Interface is a *metaphor* for an office desktop. The desktop is a surface where users can keep documents which are perceived as *stable* — i.e. this is a 1-agent environment. The environment can be *directly manipulated* with *immediate feedback*.

Such interfaces give the perception of interaction with stable *artefacts*. It is the stability of values of observables as determined through interaction that gives artefacts coherence and meaning. In the Phillips machine, the fact that the total

[9] Note: both the correspondence and meaning are initially in the view of the modeller, as before.

amount of water in the machine does not change (modulo leaks or additions) is one of the guarantees of stability that enhances the machine's meaning. Such guarantees of stability in digital computer models must be programmed. Our current methods of programming make it easy for these guarantees to be omitted or subverted. For example, a currently recurring topic of much interest in the "Forum on Risks to the Public in the Use of Computers and Related Systems" (RISKS-LIST) [Neu] is the use of electronic voting machines without voter verification facilities. The problem occurs due to the lack of guarantees that votes that are made are correctly recorded and counted. Votes are entered into a black-box computer system and their consequences later emerge. The internal states are not perceptible to voters, unlike paper ballots or the water in the Phillips machine.

Interactive Situation Models (ISMs) [Sun99] are the name we have given to our computer artefacts that are intended to support Empirical Modelling. The artefacts are constructed with computer tools that give explicit support for reliable relationships, which can produce perceived stability.

Charles Care's model of a planimeter is a recently produced example of an ISM. The planimeter is a physical instrument, mostly commonly used around 1850 for land surveying — see Figure 1.4. In the type shown in the figure, a small wheel is placed in contact with a large disc. When the disc is rotated, friction causes the wheel to rotate in response. The wheel is positioned over the disc using a cantilever. The wheel can be moved horizontally across the disc (slipping as it does so) in order to change the gear ratio of disc to wheel movement. The movement of the disc and cantilever can be controlled by the horizontal and vertical motion of a pointer, shown in the foreground of the picture. Typically the pointer would be traced around an irregular closed curve, such as the boundary of a parcel of land on a map [ABCK$^+$90, p.167]. The wheel movement is then related to the traced area.

Care's planimeter is implemented on a digital computer — see Figure 1.5. The mouse pointer can be moved within the square table shown at the top left of the figure. Movement in one axis causes the large disc to rotate, in turn causing movement of the small wheel. Movement in the other axis causes the wheel to move horizontally across the disc, slipping as it is moved[10].

[10]The Sasami feature used to produce the 3-dimensional display shown on the right of the figure is further discussed in §4.2.1.

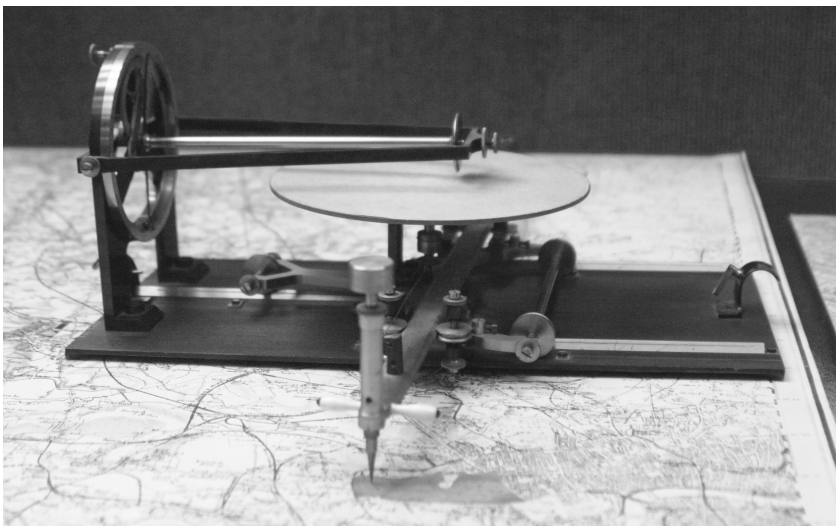


Figure 1.4: A physical planimeter shown in the London Science Museum (photo: Mark Lloyd)

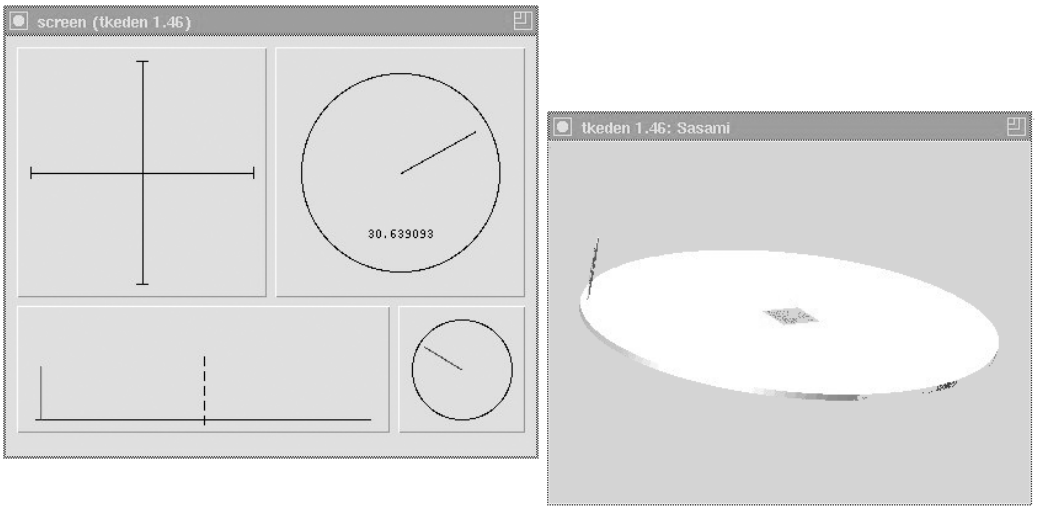



Figure 1.5: Charles Care's Planimeter ISM

Care's computer artefact represents the physical counterpart and has meaning if the user understands the conventions used by Care and stays within the realm of experience modelled by Care (described roughly above). An example of an aspect of experience that is not currently included in the ISM is as follows. The connection between the rotation of the disc and that of the wheel is maintained imperfectly by friction in the physical artefact and slippage occurs in some circumstances. Care's ISM does not at present model slippage. The relationship between disc and wheel is maintained perfectly by the ISM (floating-point imprecision aside), much as relationships between spreadsheet cells are maintained by the spreadsheet program, as I describe later in this section.

Our aim in developing an ISM is initially to produce a computer *instrument*, rather than a computer *tool*. This emphasis reflects the desire to create an artefact that enables interactive observation, together with the need for interpretation. Beynon *et al* [BCH+01] give more details, in particular characterising instruments as "maintaining relationships between state", echoing James's concept of an experience that 'knows' another, mentioned earlier (p.8).

The computer-as-instrument theme is developed further in [BWM+00], which describes the development and use of a prototype 'timetabling instrument' ISM. This ISM (*projecttimetableKeen2000*) is shown in Figure 1.6. It was used to assist with the timetabling of 120 student presentations in 2000 and 2001. The main aim of the model is to provide the timetabler with perceptible state as a basis on which to make decisions — this taking precedence over provision of automated scheduling routines. The model highlights double-booked rooms, clashes between bookings and availabilities, and other problems.

The timetabling instrument ISM illustrates a distinction between perceptible and meaningful state. The timetabling of two student presentations simultaneously in one room could be considered meaningful state: this is a situation that can be imagined and may be necessary or desirable in some circumstances. The timetabling of one person to two rooms simultaneously however is merely a perceptible state of the ISM: existence of the same person in two places simultaneously is not meaningful state for most modellers.

A third ISM, describing a digital watch interface, is shown in [BCSW99]. The

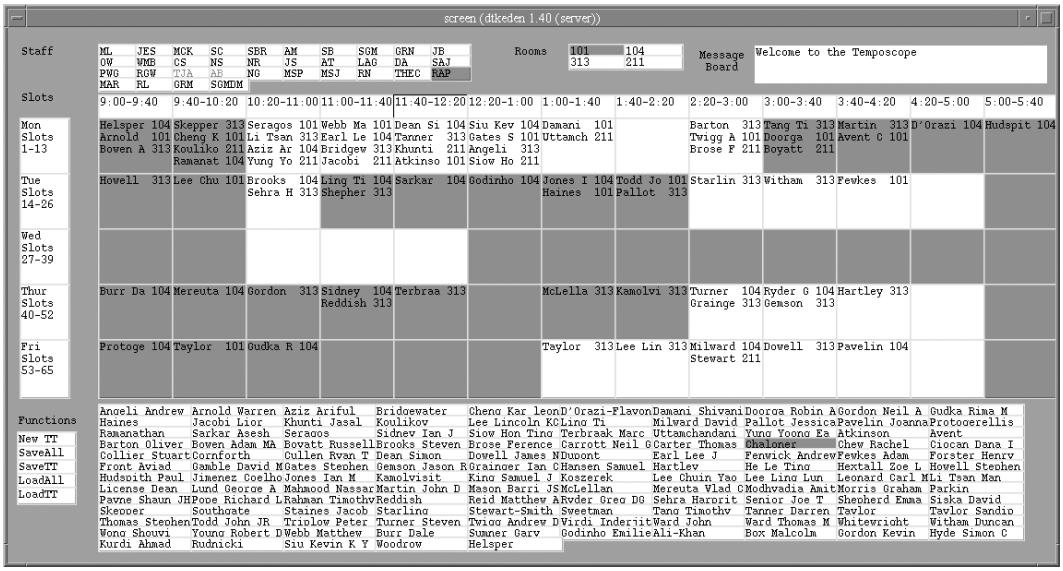



Figure 1.6: The Temposcope ISM

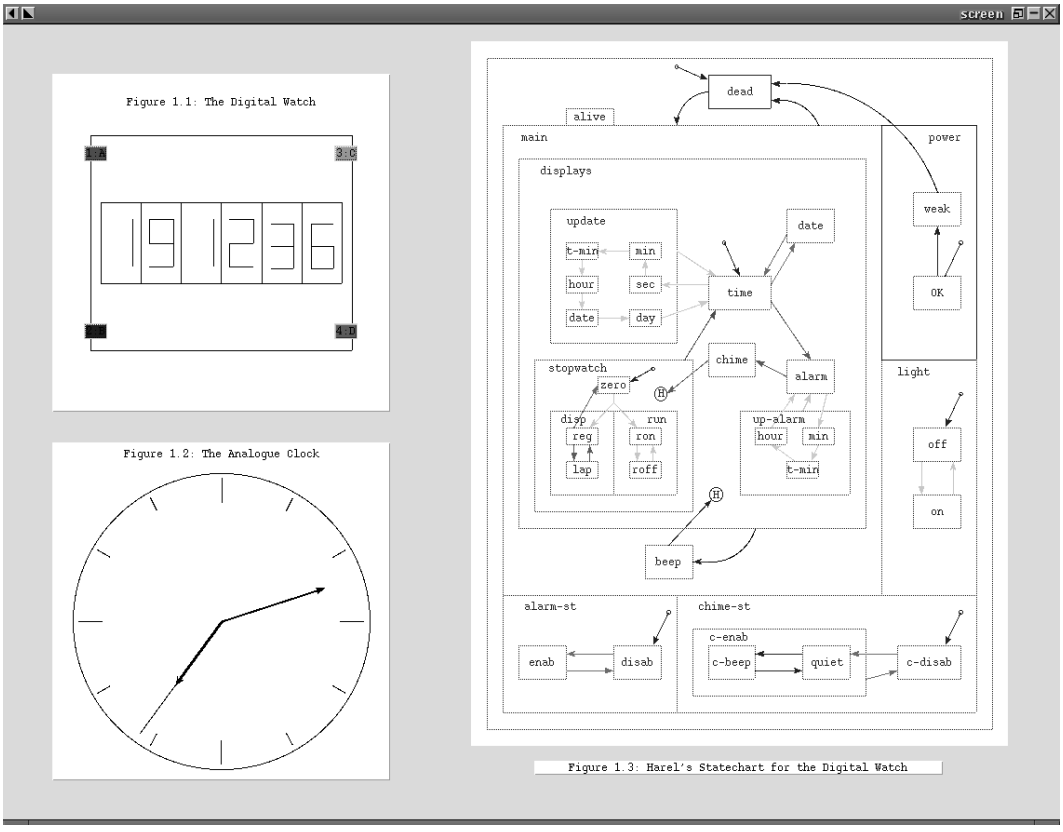



Figure 1.7: The digital watch ISM

paper attempts to show that ISMs can be used to provide support for the design of information systems. In this case, it could be imagined that an embedded system is being designed. The ISM includes a statechart [Har87] in order to make the internal state of the watch artefact perceptible to the user — see Figure 1.7. The paper gives an LSD account of the digital watch, expressing how the original modeller conceived the operation of the watch in terms of observables (states, oracles and handles), dependencies (derivates) and agents. We return to LSD accounts in §2.1.1.

The ISMs above are each implemented using the same tool which is named EDEN. As has already been mentioned, I have been the principal developer of EDEN since October 1999. In terms of enabling Empirical Modelling, EDEN is our most successful tool so far. This thesis examines EDEN closely in Chapter 4 and also two other tools: `am` (an implementation of the ADM, in Chapter 2) and the DAM machine in Chapter 3. EDEN is more focussed on interaction with the modeller than the other two tools.

EDEN implements interaction in terms of observables, dependency and agency. Interaction with EDEN, if it is used in a principled way, is more characteristic of interaction with a spreadsheet than with a conventional programming environment. Each of the EDEN-based ISMs described above, similar to a spreadsheet, 1) is provisional and incomplete; 2) can be modified at any time for revisions or 'what-if' experimentation; 3) is constructed incrementally. As in the spreadsheet, use and modification of models in EDEN can entail the same kind of interaction.

Figure 1.8 compares use of a spreadsheet program and EDEN when modelling a situation simplified from that represented by the Phillips machine. Use of the concepts of observable, dependency and agency is highlighted in each.

The top right of Figure 1.8 shows a small set of definitions written in the Eden language. A *definition* represents an indivisible relationship between observables by associating a single expression with each named observable. Similar to the spreadsheet formula, a definition implies a one-way relationship: changes to the values of observables on the right-hand side imply a change to the value of the observable on the left-hand side, but not *vice versa*. Taken together, a set of definitions forms a *definitive script*. We term a change to a definition a *redefinition*. The act of making a definition or redefinition is an act of *agency*.

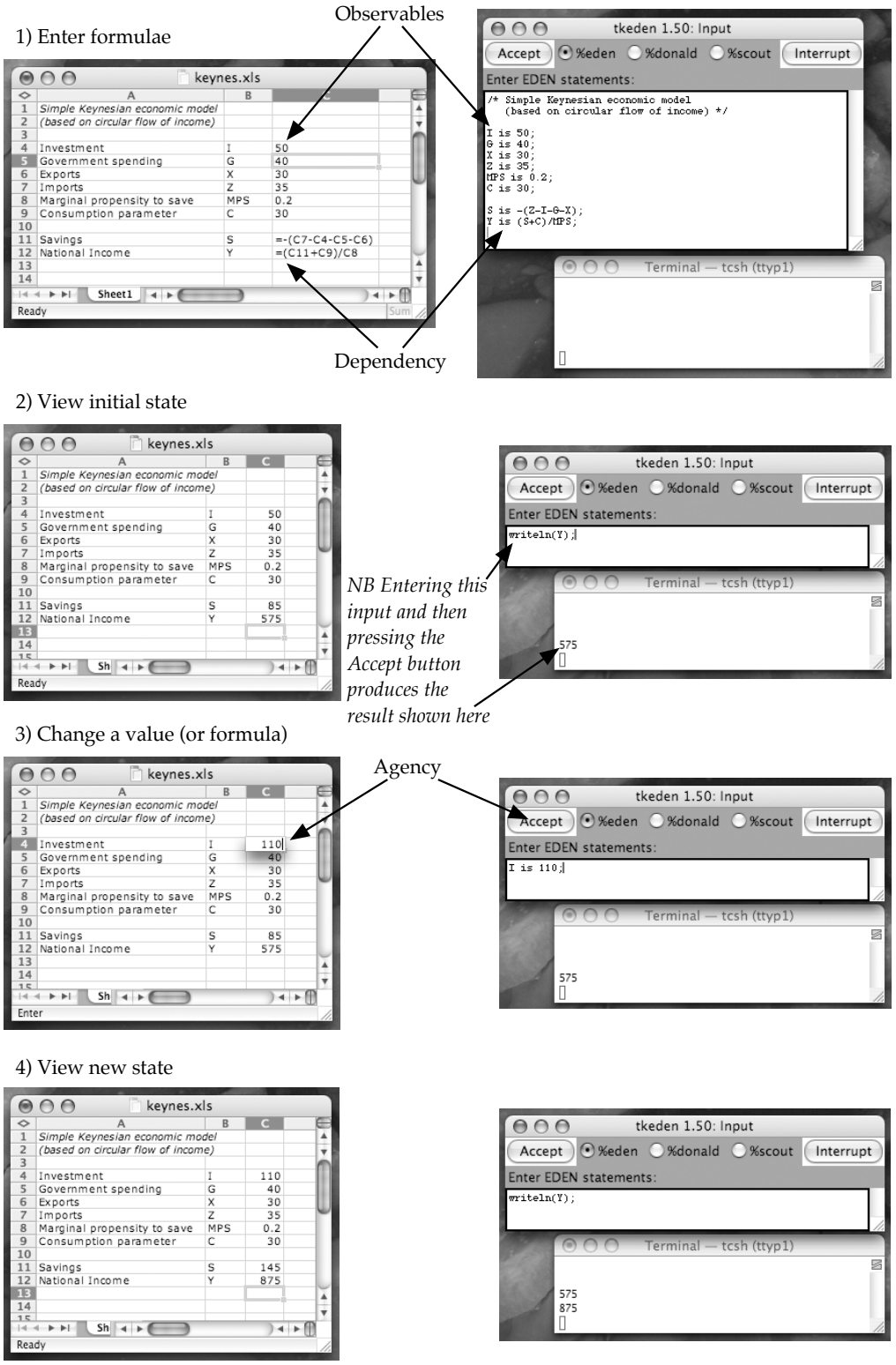



Figure 1.8: A spreadsheet program and EDEN, showing observables, dependency and agency

The dependency in a definitive script forms a graph. Appendix §3.A (p.178) explains the *script graph*. Cycles are not permitted in the script graph. (Note that the Phillips machine describes a cyclic process, which we cannot model directly in a definitive script.)

A definitive script is written using syntax specified by a particular definitive notation. We have created many domain-specific *definitive notations*, each providing operators and types specific to the domain.

There are now many sources that examine EM and EM in application. Along with 80 refereed journal and conference papers, there are currently 25 research reports, eleven PhD theses, five MSc-by-research theses, ten taught MSc project reports (including one by myself [War98]) and numerous third year project reports.

Existing documentation was not a problem to the 'first generation' researchers of EM, but the now abundant collection poses a significant challenge to a researcher working on the 'second generation' of EM today. In various attempts to improve the organisation of our library, I have created an EDDI database of many of the paper documents (the EDDI database environment in which this particular database is constructed is described in §4.2.2 and [BBRW03]); organised the scanning of much material; and maintained reference lists for our website. I have also designed and implemented a web archive 'empublic' [WRB] to hold our electronic data from various projects (mostly ISMs), dating from 1987. With the assistance of Chris Roe and Meurig Beynon, the archive now holds over 120 projects.

EM has broad scope of application, as can be seen by looking at just the subjects of the eleven PhD theses. The classification of ten of these theses by subject area gives an idea of the extent of potential application.

- Computer Aided Design / Geometric modelling — [Car94], [Car99]
- Business — [Ras01], [Ch'01], [Maa02]
- Educational technology — [Roe03]
- Software system development — [Yun93], [Nes97], [Sun99], [Won03]

This section has tried to give a flavour of the various applications of EM to modelling. More information can be found in the PhD thesis that has been omitted

from the above classification:

- A Treatise on Modelling with Definitive Scripts — [Run02]

which — rather than focussing on a particular application — gives an overview of modelling with definitive scripts.

This thesis is complementary to [Run02] in that, rather than focussing on a particular application, it concentrates on the problem of *implementing* dependency.

## 1.3 Dependency in development: a novel abstraction

Conventionally, in Computer Science we make a sharp distinction between software use and software development, corresponding to the human roles of user and programmer and to the distinction between program and data in the computer. In the interests of clarity, it is helpful in this introduction to make the conventional distinction between *application* and *development*.

However, in the previous section, we indicated that use and modification of a spreadsheet or ISM often entails the same kind of interaction. Changing the contents of a spreadsheet cell or making a redefinition can be construed as a use or a modification of the artefact. Often, one such interaction can be construed in either way.

Dependency is also an abstraction that (unusually) conflates program and data. For example, the definition '`a is b+c`' denotes a data variable '`a`' and also describes a recipe for recalculating its value.

In Empirical Modelling, the distinction between application and development is then a rather artificial one. Empirical Modelling is not only an activity to be implemented, for which we seek a supporting computer tool. The concepts of observable, dependency and agency and the underlying principles of EM[11] are also applicable to software development itself.

A full exploration of the concept of dependency in software development is outside the scope of this thesis, where the primary focus is on implementation. Many

[11]The expression "EM concepts" refers to the concepts of observable, dependency and agency. The distinct expression "EM *principles*" usually refers to the *application* of those concepts.

other sources (including the four PhD theses listed under 'Software system development' in the previous section) are concerned with this exploration. These theses and the body of practice of the EM research group since its inception in 1981 (some of which is recorded in the 'empublic' archive [WRB]) indicate that dependency provides a kind of abstraction which:

- is novel;
- is flexible;
- provides useful guarantees;
- is so simple as a concept that it is embodied in the use of the keyword 'is' in `a is b+c`, and
- may be implemented in a variety of ways, some of which may be quite simple.

This section indicates ways in which these qualities of dependency as a novel abstraction are significant in software development.

The construction of software systems is still a hard problem, despite over fifty years of accumulated experience. Neumann [Neu95, p.309] presents a table summarising cases of computer-related risk described in the RISKS-LIST[12] archive over the period 1985–1993. It shows a total of 1174 cases, 81 of which have each involved at least one death. A recent article [GMT04] reports six fatal accidents since 1993 involving software-related problems.

A (highly simplistic) division of the problems in constructing software systems is:

**P1** How can we know what we want?
(The requirements problem)

**P2** How can we know we have constructed what we want?
(The comprehension/validation problem)

Frederick Brooks [Bro87] divides the difficulties in software construction into accident and essence, and states that the accidental problems are largely solved.

[12]Forum on Risks to the Public in the Use of Computers and Related Systems.

He considers the "irreducible essence of modern software systems" to have four inherent properties: conformity, changeability, complexity and invisibility. Brooks's conformity and changeability correspond roughly to problem P1, complexity and invisibility to problem P2. The following subsections briefly examine problems P1 and P2 respectively.

### 1.3.1 Modelling requirements

The full difficulties of problem P1 were not at first recognised, perhaps because of the initial prevalence of 'accidental' problems. In 'The Mythical Man-Month' [FPB95], originally written in 1975, Brooks recommends that we "plan to throw one away; you will, anyhow". Brooks justified discarding the prototype partly due to the evolution of the requirement whilst learning happens during construction. In [Bro87], he goes further, stating that "The hardest single part of building a software system is deciding precisely what to build." In a review chapter added to [FPB95] in 1995, Brooks recommends the 'incremental-build model' over the waterfall model, 'growing' the software and having a running system at every stage. This approach allows the modelling of requirements. Many other authors concur — although there is some controversy over whether requirements should be modelled before or during development.

- The construction of 'use-cases', recommended as a starting point for object-oriented design [JC92], is an explicit attempt to model requirements to guide the development that follows.
- eXtreme Programming (XP) methods [Bec99] include the recommendation to keep the system as flexible as possible so that development can proceed in any direction as the requirement is realised. An XP project starts with a quick requirements analysis which continues throughout development.
- [Coo99] asserts that "all programming is design" which affects the possible interaction with the final product. He recommends primacy of 'interaction design' and emphasises that this must come *before* construction.
- Open Source as a methodology is partially justified on the basis that programmers will "scratch their itch" [Ray]. Open Source software can in principle

> be modified to the personal requirements of the user, if that user is also a programmer.

Modelling assists by giving meaning to requirements — it is easier to evaluate an artefact that can be directly perceived and manipulated than an abstract requirements specification. The potential application of EM to requirements modelling has been a prominent theme in much previous research [BR95, Sun99, BCSW99, Ch'01] and will not be considered further in this thesis.

### 1.3.2 Program comprehension and validation through abstraction

Problem P2 ("How can we know we have constructed what we want?") corresponds roughly to the remaining two of Brooks's "inherent properties of [software's] irreducible essence": complexity and invisibility [Bro87].

The Unified Modeling Language (UML) [BRJ99] specifies a collection of (currently twelve) types of diagram for visualising some aspects of software. [Mil02] describes the aim of the Unified Modeling Language (UML) as:

> ...the kind of tool mature engineering disciplines have had for centuries — a shared graphical language for descriptions and specifications.

He then introduces the debate surrounding the UML specification, concluding that:

> We'll know we have what we need when, as with blueprints, topographic maps, and circuit diagrams, such debates are no longer necessary.

Following Brooks, who states that "the reality of software is not inherently embedded in space" [Bro87], I would argue that software as currently constructed does not lend itself to uncontroversial diagrammatic representational forms.

In contrast, software constructed using dependency *can* be represented in an uncontroversial diagrammatic form, making use of "script graphs" (to be described in Appendix §3.A, p.178) that are directly analogous to circuit diagrams.

How can complexity be tackled? It is still a problem of essence. Dijkstra [Dij01] echoes Brooks, making a distinction between intrinsic and accidental complexity:

> I would therefore like to posit that computing's central challenge, "How not to make a mess of it," has *not* been met ...
>
> ...

> ... while we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces.
>
> We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy. We do not know yet whether trade-offs will be possible. We do not know yet whether we can invent a meaningful concept for intricacy about which we can prove theorems that help ...

The conventional way to cope with complexity is to use abstraction. Abstraction can be defined ([LT77], quoted in [Bis86]) as

> a general idea which concentrates on the essential qualities of something, rather than on concrete realizations or actual instances.

Abstraction therefore relies on the principle of analytical reduction as defined earlier (p.4) and is subject to the limitations stated there.

The philosophy of Radical Empiricism that underlies EM seems at first to preclude abstraction. Earlier, I paraphrased Wild's observation [Wil69] to the effect that: we may have our own systems and conclusions, but must be ready to examine any new fact at any time and make the necessary revisions and corrections. This suggests that in using EM for software development, we cannot abstract away any details.

However, the concept of dependency is compatible with Radical Empiricism. Dependency can be considered to be a type of abstraction[13], but one rather different from the other types of abstraction common in programming. It introduces *relationships* between observables — in the computer, guaranteed, yet reconfigurable, connections within state. Dependency allows revisions and corrections to be made at any time in the light of new experience, but also maintains consistency within state. It provides a way to structure a program in a way that is meaningful to the modeller.

Complexity in experience is not limited to computer programming. I would suggest that making a program *meaningful*, relating it to other complex experience through the guarantees given by computer-maintained dependency, is a promising way to manage complexity.

Dependency is an abstraction that produces meaningful *state*. This differs from

[13] A traditional empiricist would identify dependency as an abstraction whereas Radical Empiricism treats it as a relationship "given in experience".

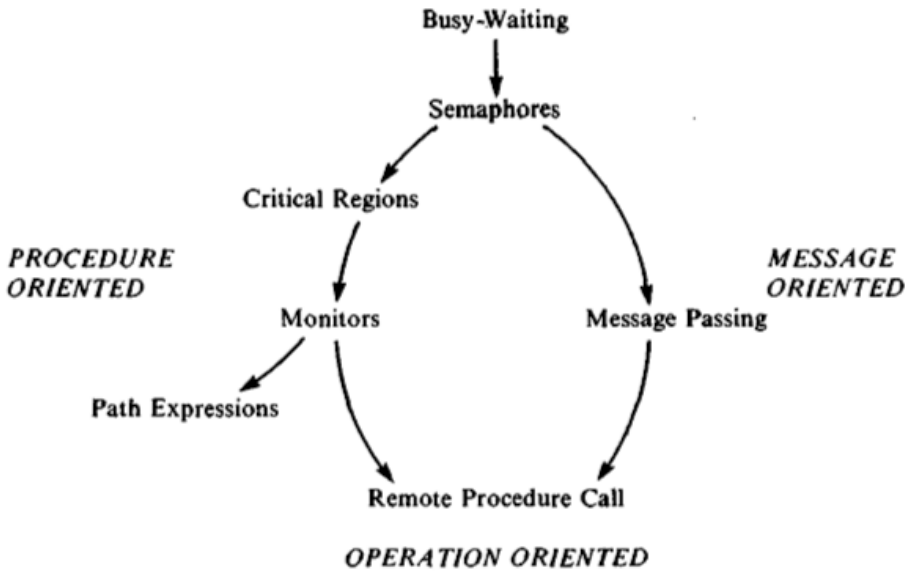



Figure 1.9: Synchronisation techniques and language classes (from [AS83, p.38], © 1983 ACM, reprinted by permission of the Association for Computing Machinery)

other common types of computing abstraction that aim to provide meaningful *operations*. The subroutine is one example of this type of abstraction. The abstract data type (ADT), now incorporated into object-orientation, hides data representation behind a set of applicable operations.

Abstractions used in concurrent programming have a similar focus. Figure 1.9 is taken from [AS83, p.38], where it illustrates historical and conceptual relationships among synchronisation primitives. Brinch Hansen's book [Han02b], which is subtitled "From Semaphores to Remote Procedure Calls", also traces this progression.

The abstractions that are currently common in both sequential and concurrent programming are thus aimed at providing meaningful operations. Programs constructed with these abstractions are built from hierarchically abstracted sequences of actions. With reference to EM concepts, the abstractions therefore relate to *agency* rather than dependency and observables.

Such abstractions assist somewhat with program comprehension, since they reduce the amount of operational detail that must be examined. However, I would contend that the meaning of the state (in the terms defined earlier — p.8) is not encoded in programs that use such abstractions. Instead, meaning must be derived

through an understanding of the operations that act upon the state. This is easiest to demonstrate through an analogy.

Sequences of actions are rather like describing a journey to a traveller without interaction with the surroundings. For example, the program code:

```
for (i=1; i<=87; i++) { fwd(); };
turn(35);
for (i=1; i<=12; i++) { fwd(); };
```

is analogous to the sequence of directions:

> "move forward 87 metres;
> turn left 35 degrees;
> move forward 12 metres..."

The traveller will reach the destination if the directions were described in enough precision and followed faithfully. If there is a minor error in one step of the directions, the eventual result may be wildly inaccurate. For example, perhaps the second instruction should have read:

```
turn(-35);
```

(*cf.* "turn right 35 degrees.")

Errors in the directions may be caused:

- by simple slips in construction:
  using `++` rather than `--`,
  (*cf.* saying 'left' instead of 'right'),

- by a mistaken belief of the constructor:
  "I didn't know that `turn()` set an absolute rather than relative heading"
  (*cf.* "I thought we were starting from the post office"),

- by an unnoticed change in the domain:
  the dynamic library file containing code for `fwd()` was removed,
  (*cf.* the road was closed for repairs).

When sequences of actions are used, there is a short distance from a working to a non-working program. The sequence of actions does not itself contain any clues to the meaning and hence to the locations of any errors: it must be stepped through from the start and the meaning of each step (with reference to the requirement) determined. After the problem is located and fixed, the sequence must usually be re-run from the start.

When dependency is used, in every state there is a wealth of possible transitions to other 'near' states. The other 'near' states are all perceptible (if the appropriate redefinitions are made) and some are meaningful. Like a spreadsheet, the script is always 'working' — although some states have no meaning. When a problem is discovered, experimentation can take place starting in the problem state. When meaning is restored, progress can continue from the point reached.

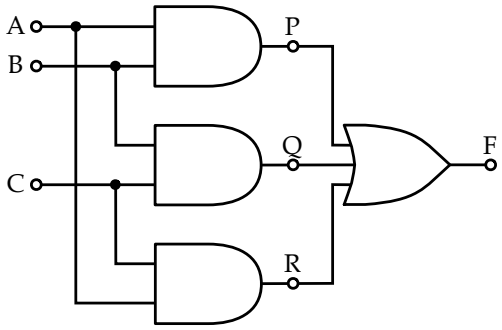
In contrasting conventional programming with the use of dependency within EM in this way, I speak from practical experience in comprehending and maintaining EDEN. Section §4.2 describes some initial attempts I have made to reconstruct EDEN 'in itself' in order to increase the internal meaning of the software. The next section describes dependency yet closer to the machine.

## 1.4 Dependency in engineering: simple and consistent relationships

The type of relationships expressed by dependency are a major factor in engineering disciplines other than computing. Leveson [Lev95, p.509] quotes an unpublished essay [McC] "When Reach Exceeds Grasp" which asserts the need for explicable relationships within software and software systems.

> [Software] developers have always had to explain relationships within and between their systems. If they can explain those relationships with the simplicity and consistency demanded of other engineering disciplines, they will succeed. If not, it probably means that a dash for novelty has sprinted too far, too fast, and too soon.

At a high level of abstraction (we shall consider a lower level shortly), a digital combinatorial logic circuit embodies a simple and consistent relationship that can be described with Boolean algebra [Boo54]. The circuit shown in Figure 1.10, for example, describes four internal indivisible relationships. Each of the three AND




```
P is A and B;
Q is B and C;
R is A and C;
F is P or Q or R;
```

```
10 INPUT A
20 INPUT B
30 INPUT C
40 P := A AND B
50 Q := B AND C
60 R := A AND C
70 F := P OR Q OR R
80 PRINT F
90 GO TO 10
```

Figure 1.10: A digital combinatorial circuit, definitive script and 'i/o equivalent' computer program

gates indivisibly relates a different pair of values of the inputs A, B and C to the values at points P, Q and R. The OR gate indivisibly relates the values of P, Q and R to that of the output F. These four internal relationships can be summarised to one indivisible input-output relationship. By common convention, this particular interpretation (at this level of abstraction) is intrinsic to how this type of diagram is understood.

In the spirit of this thesis, we could describe the gates as 'relationship maintainers'. They can then be described with a set of definitions, and the whole circuit forms the definitive script shown in the centre of the figure.

The circuit can be extended or composed with other circuits at any of the named points, with certain known limits: for example, maximum gate 'fan-out' limits the number of connections that can be made at any point.

The properties of easy extension and composition are rather like that of a spreadsheet or definitive script. In a spreadsheet, a new formula can be added to an empty and unreferenced cell without affecting the existing spreadsheet. In a definitive script, a new definition can be added to extend the script without affecting the existing script.

The circuit diagram form shown in Figure 1.10 is uncontroversial. A geometric reality is captured in a geometric abstraction [Bro87]. The circuit and circuit diagram are both located in space (which is typically planar) and there is an obvious way to make a mapping between the two. In Appendix §3.A (p.178) I describe the script graph, which is a similarly uncontroversial way of drawing a definitive script.

An input-output relation deemed to be 'the same' as that of the circuit can be maintained by a computer program such as the one shown at the right of the figure[14]. However, whereas there is a direct correspondence between the definitions in the script and the gates in the circuit in Figure 1.10, there is no equivalent correspondence between the program and the circuit. The intermediate states that arise during program execution (for example, between lines 40 and 50) are not meaningful. In a larger example, it can be difficult to determine when the state is meaningful. Making extensions to the program or composing it with other programs is then a difficult undertaking, compared to extension of the circuit, spreadsheet and definitive script.

Potentially, a single assignment from the program in Figure 1.10 might be seen as corresponding to a single gate in the circuit. For instance, if the assignment

```
P := A AND B
```

is wrapped within a procedural program as in Figure 1.11, the input-output relation of the gate in Figure 1.11 and that maintained by the computer program are 'the same'. However, with the information given, it is not easy to see how the program could be composed with others representing other gates. If the program is able to interact with other programs via the INPUT statements and the assignment, perhaps through shared memory or a data pipe, and if the program can be run concurrently or interleaved with the others, and if the program executes speedily enough, then composition may be possible. Such issues are the theme of this thesis. In particular, the discussion of the operator scheduling of the DAM machine in §3.1.2 and §3.2.5, the operational semantics of EDEN in §4.3 and the concurrent synchronisation in §5.1.2 are directly relevant to these issues. The essential problem is determining how to schedule execution of such gate-programs so as to allow interaction with meaningful state.

It may be noted that the programs in Figures 1.10 and 1.11 are non-terminating and have to be treated in conventional semantic frameworks as mapping sequences of inputs to sequences of outputs over time. The semantic problems of composition of such programs are well-recognised. Relevant references include Milner's "Elements of interaction" [Mil93], Wegner's "Why interaction is more powerful than

[14] Which is written in a BASIC-like language in order to emphasise the sequencing of actions.

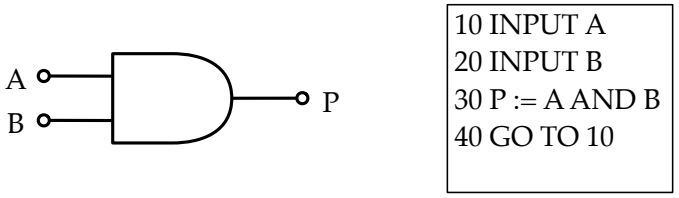



Figure 1.11: A single gate, compared with 'i/o equivalent' computer program

algorithms" [Weg97] and Wadge's proposal for the addition of the 'hiaton' as a new type of data object to the Lucid dataflow programming language to enable a nonprocedural dataflow approach to real time [WA85, p.159].

Backus [Bac78] identifies problems in composition and extension as a general feature of "von Neumann languages". He criticises such languages for a "word-at-a-time style of programming", in which:

1. semantics are too closely coupled to state transitions: every detail of a computation changes the state;

2. programming is split into an orderly world of expressions and a disorderly world of statements;

3. there is a lack of useful mathematical properties for reasoning about programs, and

4. combining forms cannot be used to build new programs from existing ones.

EM adopts an alternative perspective on computation in which dependency serves to address each of these concerns. Interaction with definitive scripts is not a word-at-a-time style of programming, as a single redefinition may cause the values of many dependent variables — and perhaps the dependency structure itself — to change in response. With reference to (1), Backus's concern relates to the 'meaningless' and low-level nature of transitions in the von Neumann model. In contrast, in EM, the atomic changes of state associated with redefinition match meaningful high-level interactions in the modeller's mind. Of course, in implementing dependency on von Neumann architectures, it is impossible to eliminate the intermediate states that arise in dependency maintenance. For many practical purposes, this

need not concern the modeller, to the extent that EM tools can provide reliable mechanisms for dependency maintenance. Where the implementation of dependency maintenance is concerned, or where the identification of dependency itself is unusually subtle, a deeper analysis is required[15].

Where (2) is concerned, definitions can be informally viewed as associating expressions with names of observables. A pure definitive script is thus based wholly in the "orderly world of expressions", and algebraic techniques can be used to manipulate and reason (3) about scripts (although this has not been much explored in our research). Finally — name space clashes aside — definitive scripts can be composed (4) with interesting results.

The functional languages which were highlighted in Backus's paper (and also the pure versions of the related data flow languages [Hud89, ŠilcRU01]) abstract away variables and hence state. For interaction with meaningful state, this is a major limitation. Backus also considers this a major limitation: "The primary limitation of FP systems is that they are not history sensitive", and proposes combining an applicative functional programming subsystem with a state and transition rules, forming an Applicative State Transition system (AST system). The definitive systems investigated in this thesis are similar to AST systems in some respects, but more emphasis is placed upon state and state-changing interaction.

So far in this section, digital logic gates have been viewed as implementing an *instantaneous* relationship between inputs and output. The abstraction by which we ignore the time taken for propagation of change allows Boolean algebra to be used.

A similar abstraction is commonly employed when using spreadsheets. We may be aware that the spreadsheet takes time to recompute, but as long as the time is small, we choose to disregard this observation. This abstraction is also a part of the EM research group's abstraction of dependency. It fits with some of the experience we wish to model — for example, in law, the act of signing a deed by convention instantaneously confers ownership of the corresponding house.

However, in circuitry, the abstraction breaks down in some situations involving particular kinds of observation and particular circuit configurations. For example,

[15] *Cf.* the discussion of major and minor transitions in the ADM in §2.3.3.

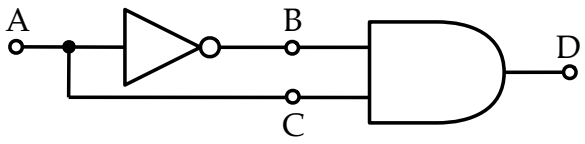



Figure 1.12: A 'pulse generator' combinatorial circuit where propagation time is significant (from [Tan99, p.144], © 1999, reprinted by permission of Pearson Education, Inc., Upper Saddle River, NJ)

consider the 'pulse generator' circuit of Figure 1.12 (taken from [Tan99, p.144]). If we consider the relationships within the circuit to be maintained instantaneously, then D will always be 0, since B will always be ¬C.

However, if the propagation delay through the inverter is considered[16], after a positive edge (0 to 1 transition) occurs at A, the output D becomes 1 for a short period of time. The width of the pulse is equal to the gate delay of the inverter (typically 5 nsec or less [Tan99]).

This issue does not prevent effective use of the higher-level 'instantaneous' abstraction. Propagation delay considerations need only be considered under particular observational regimes and with particular circuits. Both these situations are quite simple to specify.

The 'instantaneous' abstraction can still be used if observers do not register and/or respond to signals that exhibit this particular timing. The way in which we choose to ignore update time in a spreadsheet illustrates this. In the circuit example, if a human were to observe point D using a simple LED, the flash of light would be imperceptibly short. However, if we were to connect a circuit able to respond to a pulse of the given width to point D, the 'instantaneous' abstraction does not apply and propagation delay would need to be considered.

The issue also does not arise if the propagation delay of each path to the inputs of each gate is the same. To achieve this, a commonly used rule of thumb is to construct circuits with the same number of gates in each path. In this example, another inverter could be added in the path between A and C, and the 'instantaneous' abstraction then holds.

[16]The propagation delay from point A to point C is assumed negligible.

Use of the 'instantaneous' abstraction corresponds to treatment of the phenomenon as dependency. If this is not possible, we consider it as agency. However, it is clear that a definition maintainer constructed on a digital computer must implement dependency as agency.

When concurrent definition maintenance is considered, the time for propagation of change becomes significant, in ways similar to that described here. One solution analogous to the solutions commonly used in circuitry is to introduce synchronisation. I examine this topic in §5.1.2. The amount of time taken for propagation of change is also an interesting issue for study in itself, for example because of its potential implications for building reliable software for real-time applications — this is briefly considered in §3.1.2 (particularly p.104).

In this kind of analysis, then, definitive scripts appear to have more in common with circuits than procedural programs. Making the digital computer a reconfigurable analogue device[17] is a theme of this thesis. I include two particular examples that show two ways of considering this. In §3.5.4 I show how the video hardware of the computer can be considered to be performing 'definition maintenance'. In §4.2.8 I show how the EDEN definition maintainer can be considered to be an extension of a model railway control circuit.

## 1.5 Related work

This section surveys some of the literature relating to the topic of dependency maintenance and tools that implement it. Dependency is a well-known concept, but due to varied aims and focus, the work is spread across many sub-fields within Computer Science and can use varied terminology. Here, we start by considering application-level tools with the familiar example of the spreadsheet program, proceed through work related to software development and finally close with the dependency concept exhibited in hardware as dataflow computing.

The spreadsheet program provided one of the precedents for the concept of dependency as implemented in EM tools — indeed many EM sources explain the concept through analogy with a spreadsheet (eg [Bey85], [Bey90], [Yun89]). In

[17] In the sense that it 'continuously' maintains relationships rather than processes real-valued variables.

Figure 1.8 (p.17), we compared a simple model in our EDEN tool with a simple spreadsheet in the Microsoft Excel spreadsheet program.

Campbell-Kelly [CK03] sets out the 'prehistory' of spreadsheets. During the 1970s, the word 'spreadsheet' referred to a piece of paper ruled with a grid, facilitating the recording of financial data. This changed after the advent in 1979 of an interactive 'visible calculator' application for the Apple II computer. It was written by Bricklin, named 'VisiCalc' and calculated the effect of changes in (near) real-time. The product was superseded in the market by Lotus 1-2-3 in 1984 and then again by Microsoft Excel in 1990, which dominates today.

Scholarly interest in this type of computing artefact was at first limited. Early papers commonly describe spreadsheet program implementations constructed by the author in a particular language (e.g. in Smalltalk-80 [Pie86] and in a combination of APL and C [Puc87]).

An article by Amsterdam [Ams86] clearly illustrates some issues relating to choices of data structure and algorithms that are relevant to this thesis, by describing a progression of implementations in Modula-2. Amsterdam's first implementation uses a 'naïve' evaluation strategy, re-calculating the value of every cell when any change is made. The second implementation takes the dependency graph established by the spreadsheet formulae into account: a change starts a recursive update of all affected cells. The possibility of a loop in the graph then becomes a problem — the recursive update may become an unbounded loop, following cyclic formula references. Amsterdam suggests fixing the problem by associating a boolean flag with each cell, which is then used to detect the reoccurrence of an already-calculated cell during an update.

A second problem described is one involving indirect forms of reference. (For example, in the modern Excel, the formula '`=INDIRECT(A1)`' takes the value of the cell referenced by the string currently held in the cell `A1`.) The recursive update procedure no longer suffices when formulae can use these forms of reference. Amsterdam proposes three ways to resolve the problem. The first proposal does not actually constitute a solution: simply disallow indirect forms of reference. The second involves special treatment of cells whose formula contain an indirect reference. These cells are marked as 'volatile'. The 'naïve' implementation is then used for volatile

cells, re-calculating their value when any change is made. Non-volatile cells can be updated recursively as before. The third, and most complex proposal involves elaborating the dependency graph to include information on indirect references. A requirement then follows to keep the graph information up to date, deleting and adding information when a formula reference changes. I believe that the 'volatile-cells' approach is currently used in the Excel implementation. Alternative open-source spreadsheet programs include the OpenOffice/StarOffice 'Calc' [Ope, sta], which I believe currently uses the volatile-cells approach, and 'GNUmeric' [Gol], which I believe currently uses the 'elaborate-dependency-graph' approach.

Amsterdam goes on to address the question of scalability, first eliminating the necessity of storing a representation for every referenceable cell by creating a sparse data structure of 'chunks' of cells. He then goes on to describe an implementation where the least-recently-used 'chunk' of cells can be swapped out to disk if this becomes necessary.

One might expect the volume of scholarly publications relating to spreadsheets to fall after an initial burst of interest. Figure 1.13 indicates the number of scholarly articles that include the word 'spreadsheet' in the title, counted per year since 1979. The two sources used for this search were the ACM Digital Library (a collection of ACM journal and newsletter articles and conference proceedings) [acm] and the IEEE/IEE Xplore service (a collection of IEEE/IEE journals and conference proceedings) [IEE]. The hypothesis that the number of publications would fall seems incorrect, or perhaps the "initial burst" of interest has yet to cease. The count from ACM articles appears to be increasing over time, showing something like a cycle of popularity with a periodicity of 5 years.

Kay [Kay84] distills a large amount of information regarding the spreadsheet paradigm in his *value rule*: a cell's value is defined solely by the formula explicitly given to it by the user. As formulae are functions, the spreadsheet uses a form of functional language. However, attention from the programming languages community was slow to come. Casimir [Cas92] notes the lack of attention, but then suggests this is because "spreadsheets are intrinsically uninteresting". He illustrates his contention by attempting to create solutions for traditional program assignments (Fibonacci, factorial, finite automaton, game of life, selection sort, combinations,

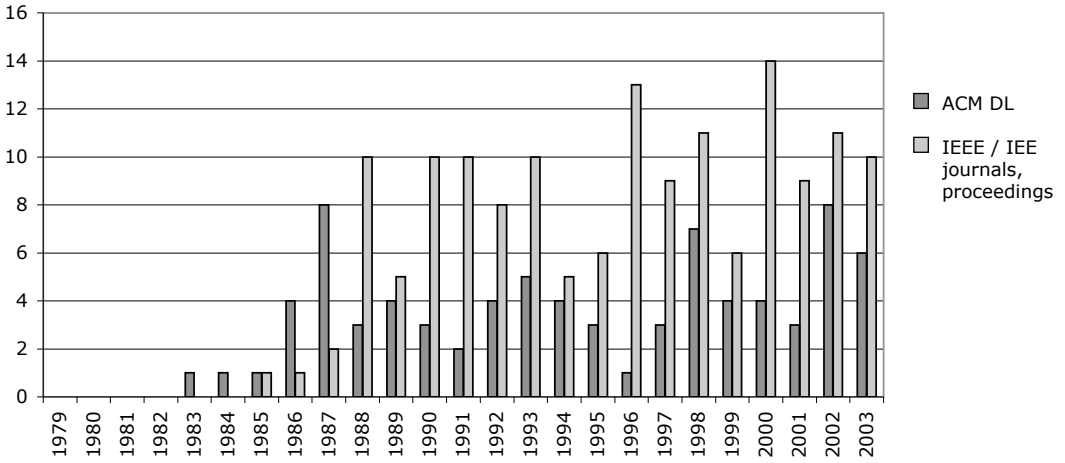



Figure 1.13: Count of articles including the word 'spreadsheet' in the title

permutations, towers of Hanoi) in Lotus 1-2-3. He concludes that "... the power of the pure spreadsheet language of Lotus 1-2-3, without the addition of macros, is equivalent to general purpose languages", but it has various shortcomings, most of which "would have been prevented if the originators of spreadsheet languages had had a more fundamental knowledge of programming languages, especially LISP and APL". Yoder [YC94] responds to the shortcomings by creating a new spreadsheet language named Mini-SP which is "structured, concurrent and scalable". It does not appear to have been implemented, although attempts have been made to implement spreadsheet programs in various functional languages (e.g. in Clean [dHRvE95], in Plane Lucid [DW90] and more recently in Haskell [Han02a], [LM02]).

Spreadsheet programs are now very popular. Boehm *et al* [BCHW95] estimates that in 2005, 55 million practitioners will work in the US "end-user programming" sector (using spreadsheets and other "application generators"), compared to only 2.75 million practitioners in the "infrastructure" sector (developing operating systems, database management systems, middleware etc.).

Some research examines how spreadsheets are used in practice in the end-user programming sector. Taking an ethnographic approach, Nardi and Miller [NM90] conclude that spreadsheet programs give strong support to collaborative working of people with different levels of programming and domain expertise. Nardi (an anthropologist specializing in the study of technology) studies end-user computing in "A Small Matter of Programming" [Nar93]. Her findings include the observation

that spreadsheet users can build spreadsheets that fulfill their needs using only a small number of functions in each spreadsheet, normally less than ten [Nar93, p.43]. Most recently, Peyton Jones *et al* [JBB03] propose extensions to the Excel spreadsheet program in order to integrate user-defined functions into the spreadsheet grid, basing their analytical approach on the cognitive dimensions of notations framework [GP96] and the attention investment model of abstraction use [Bla02].

In the late 1980s, articles began to appear stating that erroneous spreadsheet models were an increasing problem. Cragg and King [CK93] summarise the situation and present the results of a survey of spreadsheets, finding that 25% of the models examined contained errors.

Some researchers have suggested ways to reduce the number of spreadsheet errors. Isakowitz *et al* [ISL95] suggest that errors occur "because the lines of logical design and physical implementation are blurred in the conventional setting of a spreadsheet program". They suggest that every spreadsheet can be characterised by four principal properties:

spreadsheet = schema + data + editorial + binding

which is intended as a contrast to Wirth's [Wir76]:

program = algorithm + data structure

They describe a means by which these four components can be extracted from a spreadsheet and then later synthesised into new spreadsheets.

As an alternative, Fisher *et al* [FCR+02] describe a "What You See Is What You Test" methodology whereby as a spreadsheet is incrementally developed, it can also be incrementally tested. The user validates calculated values given current input data by ticking "decision boxes". The spreadsheet program tracks the test coverage over time and through cell dependencies, graphically highlighting untested cells.

The success of the dependency concept employed by the spreadsheet program has encouraged some to build similar applications in order to experiment with the possibilities. Each application adds to the basic spreadsheet design in some way. Typically, these applications still exhibit three properties, in common with spreadsheet programs (from [Hud94]):

- visibility of all intermediate results;

- continuous execution;
- integration of input, output, and "program".

'NoPumpG' by Lewis [Lew90], which was followed by 'NoPumpII' by Lewis and Wilde [WL90], extend the spreadsheet notion to include interactive graphics. A PhD student of Lewis, Repenning, created 'Agentsheets' [Rep93], an environment containing a number of autonomous, communicating agents organized in a grid much like a spreadsheet. Rausch built "The Agent Repository" [Rau96] which was the inspiration for our own 'empublic' archive [WRB]. Agentsheets is now the basis for a commercial company, AgentSheets Inc [age]. NoPumpG was also the inspiration for the 'Penguims' system, a "Programmable ENvironment for Graphical User Interface Management and Specification" intended to be used by end-users for user interface customisation [Hud94]. Forms/3 [BAD+01] is a prototype interactive environment constructed with the intention of testing the limits of the "spreadsheet paradigm" as defined by Kay's value rule.

Moving away from spreadsheet programs, dependency is also a concept central to database systems, where a view is linked indivisibly to some table data. Database tables behave as literal values do in a spreadsheet — their value does not change unless they are assigned to. Database views behave as spreadsheet formula do — their value changes along with their source values. Typically, the data contained in a database view is evaluated only on demand — for example, only when the `SELECT` command is given in the sequence of SQL commands below.

```
CREATE VIEW name_only
  AS SELECT fname, lname
       FROM customer;
SELECT * FROM name_only;
```

In the most basic configuration, a *materialized view* (which is created using the `CREATE MATERIALIZED VIEW` syntax in Oracle SQL) behaves as a database table: it is a 'snapshot' of data as evaluated at the time the materialized view was created and is not updated when source values change. However, a materialized view can be configured to act as a standard view (using `REFRESH ON COMMIT` in Oracle SQL), being automatically updated whenever source values change, or other possibilities in the continuum between view and table — for example, being automatically updated

```
pgm: codegen.o parser.o library
        cc codegen.o parser.o library -o pgm   # load

codegen.o: codegen.c definitions
        cc -c codegen.c          # compile

parser.o: parser.c definitions
        cc -c parser.c           # compile

parser.c: parser.y
        yacc parser.y            # generate parser into
                                 # file y.tab.c
        mv y.tab.c parser.c      # change name of output
```

Listing 1.1: A description file for the `make` tool, taken from [Fel79]

once per day. A materialized view is typically used to pre-calculate data required for expensive queries, perhaps during an off-peak time of database use. As it uses pre-calculated data, a query using a materialized view is likely to give a faster response time. Gupta *et al* [GMR95] consider how to efficiently update materialized views when source data changes. Halevy [Hal01] survey the techniques available for automatically rewriting queries to transparently use hidden materialized views in order to improve response times for standard queries.

We now draw attention to use of the dependency concept for software development. Firstly, many tools exist to direct code compilation to maximise efficiency and minimise response time, effectively propagating source file modifications through to compiled executable files by invoking a sequence of tool chain actions, guided by file system timestamps and a graph of dependency information. The UNIX tool '`make`' is an early and well-known example of this technique. Listing 1.1, taken from the original paper on the `make` tool by Feldman [Fel79], shows a description of the dependency between targets (for example, the output executable file `pgm`) and sources (for example, the input file `codegen.o`, which also happens to be a target of the second rule). The dependency information is augmented with a description of a shell script action, which, when invoked, will make the target file consistent with the source file.

Many variants of the `make` tool are now common, including versions that can

perform actions in parallel (e.g. [Baa88], [FH89]) or even optimistically, ahead of time [BZ89]. The original `make` tool is independent of source and target file language, but uses shell script syntax extensively for description of actions. Similar tools also exist that are targeted towards a particular source language (for example 'Jam' is designed for C and C++ [WS97, Per]), or that use a different language to describe actions (for example, Apache 'Ant' [ant] is extended using Java classes to configure actions). These facilities are also commonly incorporated into Integrated Development Environment (IDE) tools, and even compilers [CP03].

More abstract than a tool, but still with influence over software development, the Model-View-Controller pattern used for graphical user interfaces ([KP88], [BMR+96, pp.125–143]) separates processing, output and input respectively, joining them again using dependency. Typically the Publisher-Subscriber pattern [BMR+96, pp.339–343] is used for change propagation here.

A more concrete application of dependency to graphical user interfaces is demonstrated by Vander Zanden and Myers *et al*'s 'Garnet' and 'Amulet' graphical toolkits [VHM+01] that use one-way dataflow constraints. (Heron [Her02], an EM-related MSc thesis, contains many references to a "dependency tracker" by Perry [Per01] that performs a similar task.) The systems appear similar to the Penguims system mentioned earlier, but the toolkits are intended for use by developers, rather than end-users, in order to create interactive graphical applications. The Garnet and Amulet systems also incorporate pointer variables, which, when referenced in a constraint (formula), form an indirect reference constraint. This feature brings the problems of indirect forms of reference described earlier in respect of spreadsheet program implementation. Vander Zanden *et al* [VMGS94] describe the uses for this feature and various implementation strategies.

The above references generally assume a sequential uniprocessor environment. Bharat [BH95] presents a one-way constraint satisfaction algorithm that is distributed and concurrent. It is based on the sequential algorithm used in the Penguims system. It aims to meet the following guarantees:

- Each participant sees changes in the order they occur.
- Each participant always sees the cause of a modification before the effect.
- Whenever a colleague communicates with you, your view shows every change that theirs does.

Another way to consider the problem of efficiently updating spreadsheet-like state after a change is as an "incremental computation" problem. The language 'INC' [YS91] was designed to make it easy to write "incremental programs", which are performed repeatedly on nearly identical inputs. Ramalingam and Reps [RR93] give a guide to some of the literature in this area. The 'incXSLT' incremental transformation framework [VL02], a proposal for XML document manipulation, is a more recent example paper on this theme. Not all agree that the term 'incremental' is an appropriate label in this context: the recent paper by Acar, Blelloch and Harper [ABH02] on "Adaptive Functional Programming" uses the word 'adaptive' as an alternative for 'incremental'.

In the software literature, it now seems agreed that the idea of "dataflow languages" is subsumed within "functional languages" [Hud89, p.380]. However, dataflow *hardware* has a long and rich heritage, which was recently reviewed by Šilc *et al* [ŠilcRU01]. They conclude by critiquing the curious present situation, where the von Neumann interface to the processor forces much translation between dataflow and sequential code:

> Why should:
>
> - a programmer design a partially ordered algorithm, and
> - code the algorithm in total ordering because of the use of a sequential von Neumann language,
> - the compiler regenerate the partial order in a dependence graph, and
> - generate a reordered "optimized" sequential machine code,
> - the microprocessor dynamically regenerate the partial order in its out-of-order section, execute due to a micro dataflow principle,
> - and then reestablish the unnatural serial program order in the completion stage?

They suggest that the trend in dataflow research has been to incorporate explicit notions of state into architecture, and that, at the same time, von Neumann computing has incorporated many of the dataflow techniques, the multithreaded model of computing emerging within the resulting continuum.

The main ideas of spreadsheet programs and the `make` tool are, of course, well-known. The purpose of this section has been to document their development in more detail than has appeared before, either in EM-related theses or elsewhere. Certain other ideas and techniques, for example those related to the topic of materialized

views and those discussed in [VMGS94], appear to be related to some of the technical work described in this thesis. They have come to the author's attention too late to have the detailed analysis they deserve, and will be a focus of future research. The distinctive feature of the technical work described in this thesis is that it is conceived within the broader framework of "Interaction with Meaningful State". The concept of meaning here embraces personal understanding that arguably cannot be expressed in a formal and objective fashion but is implicit in the interaction between the modeller and the model. This can be best appreciated by contemplating the richness of the agency invoked in modelling with the Authentic ADM (*cf.* Chapter 2) or the `dtkeden` interpreter (*cf.* §4.1.6).

## 1.6 Thesis aims

This introduction has given a brief overview of Radical Empiricism and possible uses of Empirical Modelling. The thesis itself is primarily about the questions posed on p.2:

- How can we best exploit and develop our existing tools for implementing EM activity?
- What prospects are there for better tools in the future?

These questions involve significant technical issues but are also foundational and therefore somewhat philosophical in nature.

The EDEN tool used for the planimeter model is now a 30,000 line C program. It implements dependency but is itself constructed from a non-trivial amount of procedural agency. Considering that the von Neumann machine it executes on is constructed partially from combinatorial logic circuits, it is a valid question to ask if dependency can be implemented and used at a lower level.

A long term aim (which goes beyond this thesis) is therefore: to construct a basic microcomputer which implements and uses dependency at a low level. Interaction with the machine should allow questions such as "why is that pixel white?" to be answered using techniques similar to those that might be applied to a mechanical

typewriter. In such a machine, internal state would be linked with dependency, giving it meaning beyond the raw von Neumann state.

This thesis works towards this long term aim by attempting to answer relevant smaller questions: How can we improve our EDEN tool? Can we replace some of the 30,000 lines of C with a smaller core that implements dependency? What is this small core? What precisely is dependency? And a definition?

On current digital computers, in order to implement dependency, it is first necessary to specify the agency that is involved in definition maintenance. This can be seen as adopting a 'dependency-as-agency' perspective. This motivates the final aims of the thesis, which are: to elucidate the difference between dependency and agency with respect to a digital computer, and to determine a framework enabling the two worlds of dependency and agency to be bridged. If this is done, we have taken the first steps towards enabling interaction with meaningful state on a digital computer.

This introduction has divided the topic of dependency into four sections organised 'top-down'. However, both the division and the top-down organisation were primarily for the purpose of exposition. This thesis takes a bottom-up view of the topic that is unprecedented in previous EM literature.

The concept of dependency, together with the intimately related concepts of observable and agency appear to have wide-ranging applicability. Examples of dependency have included:

- the possibility of a "missed free-throw" in a basketball tournament is linked to the size of the basketball;
- in the Phillips machine, the surplus balance in the economy is represented by the level of water in a particular tank;
- the reservation status of a seat on a particular scheduled future aeroplane flight is recorded in a database;
- a document icon in a GUI is used as a metaphor for the document's contents;
- the wheel rotation of a planimeter is linked to the rotation of the disk and the cantilever position;

- the colour of a time slot in our computer 'timetabling instrument' ISM can be linked to the availability of the slot;
- the external state of a digital watch ISM is linked to the internal state as described by a state chart;
- the national income is linked in a spreadsheet to the consumption parameter and marginal propensity to save;
- the output of a digital logic gate or circuit (feedback not considered) depends upon the current inputs.

These examples illustrate the broad range of perspectives that our dependency concept is intended to envisage: the examples here relate to all aspects of explicit, situational, mental and internal (SEMI) state [BRWW01]. In this thesis, we will be concerned with the implementation of dependency with reference to all these perspectives.

## 1.7 Thesis outline

This thesis contains six chapters. The outline is as follows.

Chapter 1 is this chapter, which motivates and states the aims for the thesis as well as providing an overview.

Chapter 2, "States and transitions in the ADM", describes the concepts of the LSD account and a machine developed to animate such accounts, the Abstract Definitive Machine (ADM) [Sla90]. Slade describes three possible strategies that can be used in the implementation of a definition maintainer. The ADM is an example of the first strategy: "evaluate at every use (storing only formulae)". The main emphasis of the ADM is *agency*. The critical review reveals subtleties and inconsistencies in the way that the ADM concept has developed over time. An alternative algorithm for ADM execution is formulated and termed the 'Authentic' ADM. The notion that states and transitions can be placed into 'major' and 'minor' categories, depending upon the perceived granularity, is also developed, which is used later in the thesis. The appendix to Chapter 2 demonstrates the use of Slade's '`am`' implementation of the ADM.

Chapter 3 examines the Definitive Assembly Maintainer (DAM) machine [Car99]. The DAM machine is an example of the second evaluation/storage strategy: "evaluate at each redefinition (storing formulae and values)". A topological sort algorithm [Knu73] is used to schedule calls to operators. The system is written in ARM assembler. A graphical front-end '!Donald' uses the DAM machine to perform definition maintenance in an implementation of the "Definitive Notation for Line Drawing", DoNaLD. The DAM machine emphasises only *dependency*, and so graphics must be created by operator side-effect. The chapter shows how the DAM machine and !Donald were extended to allow low-level definitive programming in 'DAMscript' and also how the system was extended to show the state of the DAM machine store directly on screen, when the video hardware can be considered to be maintaining dependency between the DAM machine store and screen pixels. Three character glyphs are created on the screen using dependency. In this example, "why is that pixel white?" is a question which can be answered through investigation of a script graph of indivisible, 'instantaneous' and statically traceable relationships, lending the answer more meaning than would be the case without a definition maintainer. The appendix to Chapter 3 explains my concept of the 'script graph': this is an acyclic directed graph representing the dependency in a definitive script. An enumeration of the possible script graphs for N=4 and N=5 is presented.

Chapter 4 examines the Evaluator of DEfinitive Notations (EDEN) [Yun90]. EDEN has been the primary tool of the research group at Warwick since 1987. It is explained after the ADM and the DAM machine due to the greater complexity of the tool, which gives equal emphasis to both dependency *and* agency. An overview of the historical evolution of the tool is given. The first Eden model (*texteditorYung1987*) is used to illustrate EDEN's dependency and agency features. An overview of developments since 1999 involving the author is given, which includes work on the theme "constructing EDEN in itself". The chapter then contains a close study of EDEN's scheduling and execution mechanisms, firstly from perspective of a tool user, who must treat the tool as a black box, interacting by using notations only. The virtual machine used by EDEN is explained and the scheduling algorithm is summarised in pseudo-code (given in the appendix to Chapter 4) and two diagrams. These results are used to explain the differences between dependency and agency in EDEN.

Finally Y.P. Yung's proposal [Yun96] for a re-implementation based on a Four Layer Prioritised Action Scheme is briefly examined.

Chapter 5 investigates three deep problems in dependency maintenance that run through the earlier chapters in the thesis. The Eden language allows the distinctions between dependency and agency to be illustrated, but the EDEN implementation is firmly rooted on a sequential machine. The notion of the script graph is extended by adding agents to implement dependency-as-agency. There is more than one possible way to add agents to a script graph, and this is outlined. The resulting definition-agents can in principle operate concurrently. The question of how they should be synchronised is examined and a prototype solution in the 'SR' concurrent language is given in the appendix to Chapter 5. Some problems with definitive lists are revealed and the related theory of moding [Mez87] examined. The `%edensl` notation is developed as a prototype solution to problems with definitive lists. A 'tri-box' framework for concurrent dependency maintenance is presented, based on coordination between Observing, Changing and Updating agents. The script graph is represented by 'boxes' in store which can themselves be maintained by dependency, giving a sound basis for Higher Order Dependency.

Chapter 6 gives some conclusions, outlines the contribution of the thesis and contains some brief statements about possible future work.