Chapter 4
Empirical Modelling in a nutshell

Empirical Modelling (EM) research was initiated by Meurig Beynon at University of Warwick more than 20 years ago. The name *Empirical Modelling* was adopted in 1994. EM can be conceived as an exploratory and human-centered approach to software systems development (Yung, 1993, Wong, 2003, Sun, 1999). The word *empirical* reflects two characteristics of the EM approach, namely, the experimental and the experiential (cf. Cartwright, 1999). In contrast to mainstream computing, Empirical Modelling is founded on William James's *Radical Empiricism* – a philosophical foundation that is radically different from that upon which mainstream computer science is based (Beynon, 2005). The potential of EM is not restricted to software systems development. Various researchers have explored the potential of EM in numerous application domains. These include computer-aided design (Cartwright, 1994), computer graphics (Cartwright, 1999), business decision-support systems (Rasmequan, 2001), financial modelling (Maad, 2002), participatory business process reengineering (Chen, 2001), computer-supported learning (Roe, 2003; Harfield, 2008).

This thesis develops a conceptual framework for efficacious groupware development that is based on the principles of EM (cf. chapter 7). As a preparation for the reader to fully apprehend the conceptual framework, this chapter discusses the fundamental principles of EM, its philosophical foundations, and the differences to other programming paradigms. As I mentioned in chapter 2, articulation work is one of the central concerns in collaborative work. For this reason, the discussion in this chapter is primarily concerned with the single modeller context, i.e. with how EM is carried out in a single modeller context. EM in a multiple modeller context and its fitness for collaborative modelling will be discussed in chapter 5.

In section 4.1, I briefly describe the basic features of EM. These can be divided into three

categories: the ODA framework, the modelling process, and the situated and cognitive aspects of the EM model. The discussion here aims to provide the reader with enough background to appreciate the discussion of the philosophical foundation of EM in section 4.2.

In section 4.2, the focus is moved to the philosophical issues that EM has taken seriously during the development of its principles over the past two decades. These include William James's (1912) Radical Empiricism, David Gooding's (Gooding, 1990) notion of construal, and some of the writings of Brian Cantwell Smith (Smith, 1987). This section reviews a contribution to the philosophical foundation of EM made by the author, based on Russell Norwood Hanson's (1958) philosophical inquiry into the notion of observation and theory building in empirical science.

Due to its distinctive theoretical foundation, EM has numerous distinctive features in practice and cannot be regarded as programming in the sense of traditional programming. Section 4.3 explains the ontological and practical differences between EM and other programming paradigms, and how the principal tool – tkeden – supports the experimentation principle of EM in practice.

4.1 Concepts of Empirical Modelling

In this section, I briefly describe the basic concepts of EM: the ODA framework, the modelling process, and the situated and cognitive aspects of the EM model. The discussion here is not attempting to formalize these concepts into a unified framework as other authors have done (e.g. Wong, 2003), but aims to provide the reader with enough background to understand the discussion of the philosophical foundation of EM in the next section.

4.1.1 The ODA framework

EM researchers use the acronym ODA to refer to the key elements in an EM model, namely, *observable*, *dependency*, *agent*, and *agency*. The ODA framework refers to the meaningful use of these key elements during the construction of an EM model.

Observable and observation

In EM, *observables* are entities that can be directly apprehended by a human observer. This can be virtually *anything* whose identity can be established through the experience of a human observer (Beynon, 2008). For instance, an observable can be a value (e.g. 8), a description (e.g. "the ball has flown in the air"), a feeling (e.g. happiness), an event (e.g. an apple has fallen from the apple tree), a volume (e.g. 500ml), a relative perception (e.g. loud music), a status (e.g. the train engine is switched on), etc.

In the construction of an EM artifact (also an EM model), the identification of observables is through making careful and meaningful *observation* of the referent, whether it is a thing in the real world or it is an idea in the mind of the human observer who made the observation. When the observation refers to a thing in the real world, the notion of observation in EM is similar to that in scientific experiment, where the EM artifact can be viewed as construal in Gooding's sense. However, observation admits different interpretations according to the context and status of the observer (see §4.2.1 below). What is regarded as relevant (and irrelevant) is subjective and reflects the current interest and *mode of observation* of the human observer (Beynon, 2008). It also depends on the level of detail of the observation and the current configurations of the environment within which the observation is made. This makes the identification of an observable, its presence and existence, a situated and subjective matter.

In contrast to the systematic identification process (e.g. Noun-Verb analysis in object oriented analysis and design (cf. Meyer, 1997) for entities (e.g. objects, and variables in objects) in traditional programming, the identification of observables in EM is much more fluid. Observables may be defined, redefined and removed throughout observation and interaction with the EM artifact during the modelling process. Taking all these characteristics into consideration, the intended character of *observable* in EM is much broader than what is known as a *variable* in the sense of traditional programming.

Dependency

In an EM model, a *dependency* can be represented by a *definition*, which defines the relationship between observables. Typically, such a definition has the following form:

$$d \text{ is } f(x_1, x_2, x_3 \ldots x_n)$$

where d, x_1, x_2, $x_3 \ldots x_n$ are observables, *is* is the keyword for separating the dependent and the observables upon which it depends. With this definition, observable d changes when any of the observables x_1, x_2, $x_3 \ldots x_n$ changes. Conceptually, the state-changing activity behind a definition can be interpreted as a cause-effect link between observables. Technically, it can be viewed as a transparent and atomic transaction for change propagation among observables. However, it is inappropriate to think that the stage-changes of observables are separate change-update steps as in traditional programming (e.g. change listeners in Java). The *state-changing* activities among observables occur simultaneously and atomically as in everyday experience of concurrency. For instance, as in everyday experience: expectations are framed by previous experience of the reliability of observations; observables and dependencies may be revised in response to new observations; dependencies reflect how state-changes to observables are perceived to be indivisibly linked (Sun, 1999). In these respects, the actual operation of a dependency in an EM model is closer to the dependency mechanism found in spreadsheets rather than component and service dependency as in Enterprise Java Beans (EJB) or *dependency injection* in design patterns.

Agent and Agency

In EM, an *agent* is typically conceptualised as a set of definitions and observables that causes state-changes within an EM model, and *agency* is associated with the "attributed responsibility (or privilege) for a state change to an agent" (Sun, 1999). In contrast to Artificial Intelligence (AI), EM has taken a much broader notion of agent. In classical AI, agents are thought of as entities that perform preconceived actions based on a stimulus-response model. In EM, by contrast the notion of agent is closer to what we may experience in attempting to understand a phenomenon in our everyday life. In this sense, agents can be "anything that has the capacity to change state" in an object (Harfield, 2008; also cf. Wong, 2003). For instance, I wake up when the alarm goes off (i.e. the alarm acts as an agent

which changes my state to 'awake').

Like observables and dependencies, agents may be defined, refined and removed in response to the current *state-as-experienced* by the modeller (see §4.1.2 below). In addition, the attribution of agency is "shaped by the explanatory prejudices and requirements of the external observer, and by the past experience of the system" (Beynon, 1997[59] cited in Sun, 1999, p.26). Considering the previous example, it may not be the alarm that wakes me up, but an irritating beeping sound (which it may be that the alarm has generated) that wakes me up. Whether the agency involved in waking me up is attributed to the 'alarm' or to the 'irritating beeping sound' depends on the perception of the external observer.

In relation to the identification of agents and agency during the modelling process, Beynon (2008) describes three views of agents:

View 1: every observable or object is an agent, as is the external observer

> In this view, an agent is an association of observables. Since every observable can potentially act as an agent that has potential effects on other agents (i.e. changing their states), an agent can be a collection of one or more observables that are co-existent and co-absent (Beynon, 2008; Wong, 2003). An analogy in object-oriented modelling may be a class with properties only (Wong, 2003).

View 2: agents are objects responsible for particular state changes

> In this view, agents are deemed responsible for state changes to other observables or agents (Beynon, 2008). For example, the church bell sounds when somebody pulls the bell-rope. It is, of course, impossible to say that pulling the rope is the only way to cause the bell to sound. However, careful observations (e.g. there is a priest pulling the rope when the bell has been sounding) may influence the external observer to believe that such agency (i.e. connecting the sounding of the bell with the pulling of the rope) exists. In EM models, this kind of agency can be express

[59] The lecture notes for a MSc module (Beynon 1997) are now part of the "Introduction to Empirical Modelling (CS405)" course (cf. Beynon 2008)

using definitions (as discussed above), where state changes to x_1, x_2, x_3 … x_n trigger state-change of *d*.

View 3: virtual agency in the closed-world

In this view, the context of observation for the agent is so circumscribed that there is little doubt about its existence and presence (Beynon, 2008). Moreover, the behaviour of the agent is so reliable that its operations can be conceived in terms of stimulus-response actions (Beynon, 2008). Such a kind of agent can be considered as *closed* in the sense that further exploration is unlikely to add new insight or to change its behaviour (Wong, 2003).

Figure 4.1 illustrates the differences between the three views of agents. From a systems development perspective, the interest of the EM process is in the progression of agents from view 1 to view 3 through continuous observation of, interaction with, and experimentation with the EM artifact (i.e. the EM model). This process for circumscribing agency, where the development of the EM model progresses from obscure knowledge of agents to many circumscribed agents, is elsewhere known as the agentification process (e.g. Wong, 2003).

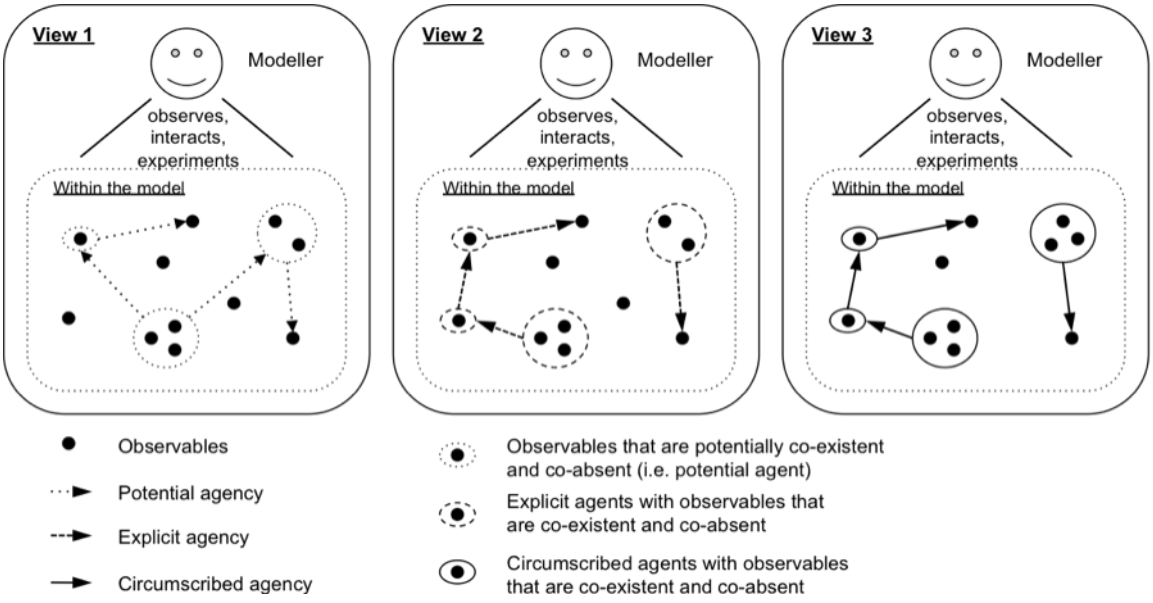

Figure 4.1 – Three views of agent and agency in EM

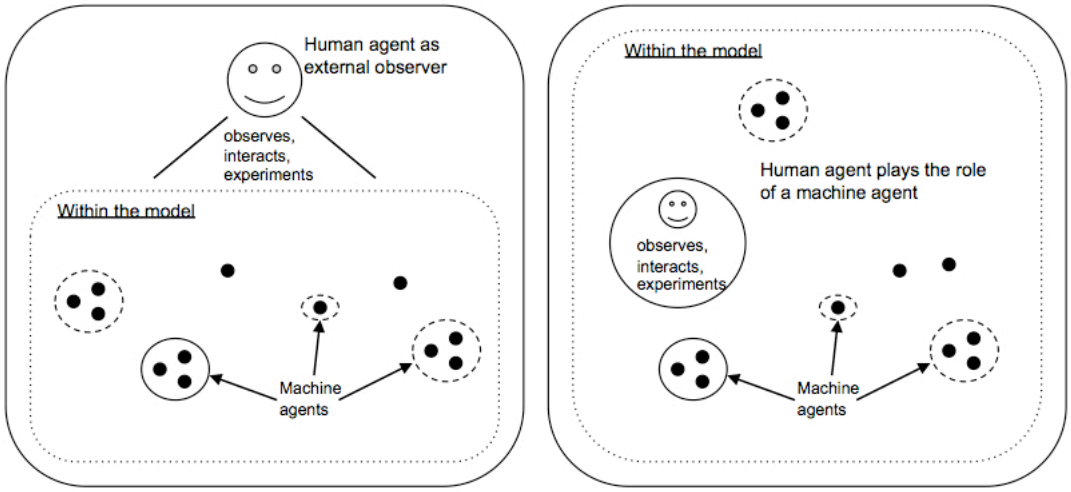

Figure 4.2 – External observer vs. internal observer in EM

It is worth noting that at any time during the EM process, the modeller (i.e. the human observer) may play the role of an agent as part of the experimentation and may also interact with other agents in the model through definitions and re-definitions. For this reason, it is important to distinguish two types of agent (and agency) within the EM process, namely, human agents and machine agents. The archetype for a *human agent* is the modeller who either, as an external observer, plays the role of super-agent that has the overall power to intervene the animation of the EM model, or, as an internal observer, plays the role of a machine agent within the model. Figure 4.2 illustrates the difference between a human agent as an external observer and a human agent as an internal observer.

When unspecified, the term 'agent' is used to refer to the agents in an EM model, i.e. the machine agents. In EM, the actions of a human agent can be circumscribed into a machine agent after prolonged observations (i.e. through the agentification process (Wong, 2003)). For example, the train drivers in the Clayton Tunnel model were initially human agents at the beginning of the modelling process, but some became machine agents in the model as the modellers gradually circumscribed them.

Illustrating ODA in the Jugs model

As I discussed earlier, *observables*, *dependencies*, *agents*, and *agencies* (ODA) in EM are open to change in response to continuous observation. However, for the sake of illustration, I shall temporarily 'freeze the temporal factor' of the modelling process. As depicted in figure 4.3, the ODA in the EM jugs model (EMArchive: jugsBeynon2008), as represented both in the definitive script and on the tkeden screen, intimately link to its referent in the real world. For instance, the liquid with Jug A is captured by the observable *'contentA'* in the definitive script and is shown as yellow blocks on the tkeden screen in the jugs model. Similarly, the dependency "*Bfull is capB==contentB"* captures the definition that "Whether Jug B is full is depending on whether the liquid in Jug B has reached the capacity of Jug B". In EM, dependency continues to hold even when, e.g., the size of Jug B has changed. This is in contrast to 'dependency' in conventional computer programming (as I mentioned in §4.1.1). The 'pour' agent closely resembles the pour action that can be performed on the pair of jugs by a human agent.

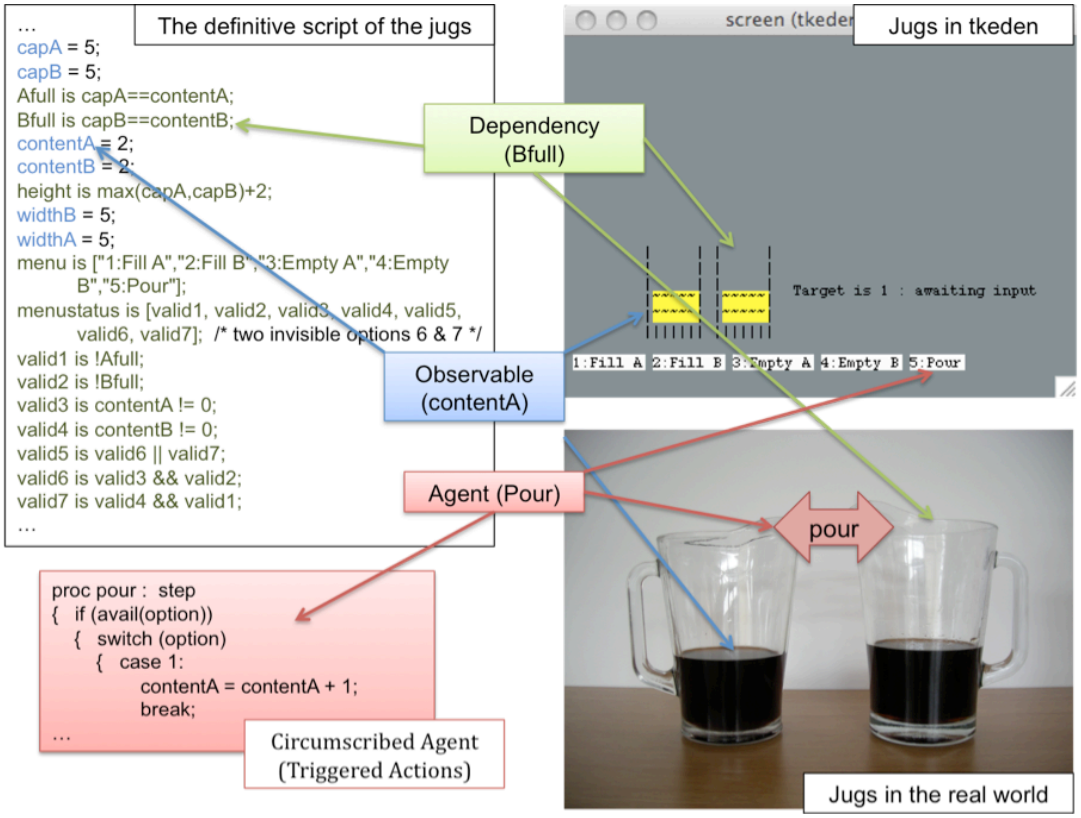

Figure 4.3 – The relationship between the definitive script, the screen, and its referent in the jugs model (EMPA: jugsBeynon1988)

4.1.2 The EM process

Central to the EM process (i.e. the modelling process) is the notion of *modelling state-as-experienced* (Beynon, 2008) and *modelling with definitive scripts* (MwDS) (cf. Rungrattanaubol, 2002).

Modelling state-as-experienced

In traditional computer science, state is typically characterised in relation to the *internal abstract behaviour* of the *program*, which assumes the behaviour of a program can be fully apprehended within a fixed context for use. This conception can be reflected in the adoption of state machines, e.g. Turing Machines, state charts in mainstream computing, e.g. software engineering. In contrast, EM has a broader conception of state – a sense of state such as we apprehend in an everyday phenomenon. Such an everyday sense of state is open to diverse interpretations and can be refined through continuous observations and interactions that are based on personal experience (Harfield, 2008). It is this everyday sense of state, which Beynon (2008) calls *state-as-experienced*[60], that the EM process is most concerned with.

Figure 4.3 shows the 'final' jugs model (EMPA: jugsBeynon2008) as it was submitted to the EM repository. It consists of two jugs and a pouring agent. The model reflects the aspect of the situation as taken into account by Beynon at the time of submission, but this neither means that the situation had been modelled fully nor that it would be the same had another modeller modelled the same situation. For instance, the scripts of the model reflect the fact that modeller, when he made his observation of the jugs, was not interested in (or did not experience issues concerned with) the viscosity of the liquid in jugs, the speed of pouring action, the fact that the jugs liquid may be poured to somewhere other than into the jug next to it, or that the glass jug may be broken if it is filled with boiling liquid, and so on and so forth.

[60] Note that, state-as-experienced is not to be equated with 'what the modeller actually experiences at a particular time'. Instead, the term relates rather to the conception of state, and in particular to the potential to take account of more than has been explicitly observed and recorded.

Modelling with definitive scripts

"State-as-experienced encompasses the holistic experience of interacting with the computer-based artefact" (King, 2007, p.30). To support modelling state-as-experienced, EM artifacts (i.e. EM models) are represented in definitive scripts. This allows the modeller to construe[61] the state in many semantically different ways by adding new definitions and making redefinitions to the definitive script (Beynon, 2008). For example, a redefinition may represent a state-change of an observable in the model, a refinement of a dependency based on new observations, an interaction with the model, a re-attribution of agency, etc.

A *definitive script* typically consists of a number of definitions and these can be in various *definitive notations*. At the conceptual level, definitions define observables, dependencies, and agents (i.e. the ODA framework as discussed in §4.1.1). For effective modelling purposes, a number of definitive notations have been developed in the past 20 years for modelling in various domains in addition to the general purpose EDEN notation (Yung, 1993). These include, for instance, DoNaLD for line drawings (Yung, 1993), SCOUT for screen layouts and organising of information on screen (Yung, 1993), Sasami for 3D modelling (Carter, 2000), ARCA for Cayley diagramming (Beynon, 1983; Ward, 2004), EDDI for database modelling, etc. In the EM process, the modeller may observe, interact or experiment with the EM model. Hence, the major activity in the EM process can be viewed as *modelling with definitive scripts* (Rungrattanaubol, 2002).

In systems development, two typical kinds of activities can be identified (Beynon et al., 2006): exploratory and prescribed. While the former focuses on sense-making (i.e. process focus), the latter focuses on meeting predetermined goals (i.e. product focus). In contrast to a traditional perspective on systems development, EM has less interest in the predetermined goals, and is more concerned with sense-making activities through interaction with the computer-based artifacts – that is, through modelling with definitive scripts. As shown in figure 4.4, some of these exploratory actions can be later transmuted into actions that are no longer 'sense-making', as when a *ritualised* pattern of interaction and interpretation is

[61] The term 'construe' is used to refer to the activity of *making a construal* in the sense introduced in Gooding (1990). The connection between EM and Gooding's idea on construal will be elaborated in §4.2.

discovered during the modelling process. Consequently, the EM process is often fluid and dynamic.

As shown in figure 4.5, the process of sense-making and ritualisation in EM involves continuous observation and interaction with the definitive script that reflects the modelling situation and the referent. In principle, the EM process will progress until the set of definitions reflect (or closely approximate to) some state of the referent as it is experienced by the modeller. However, state-as-experienced is an evolving object of study – observing the referent and the model and interacting with the model may provoke further insights, and therefore, create a *new* state-as-experienced. For this reason, this process may never be considered as 'completed'.

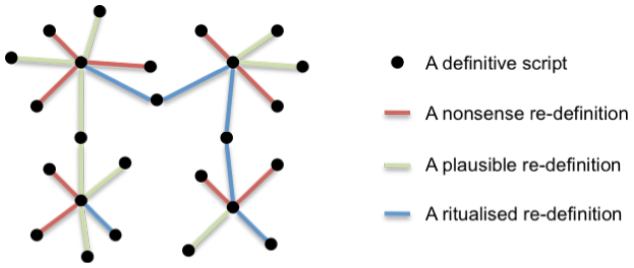

Figure 4.4 – Exploring ritualised definitions in EM[62]

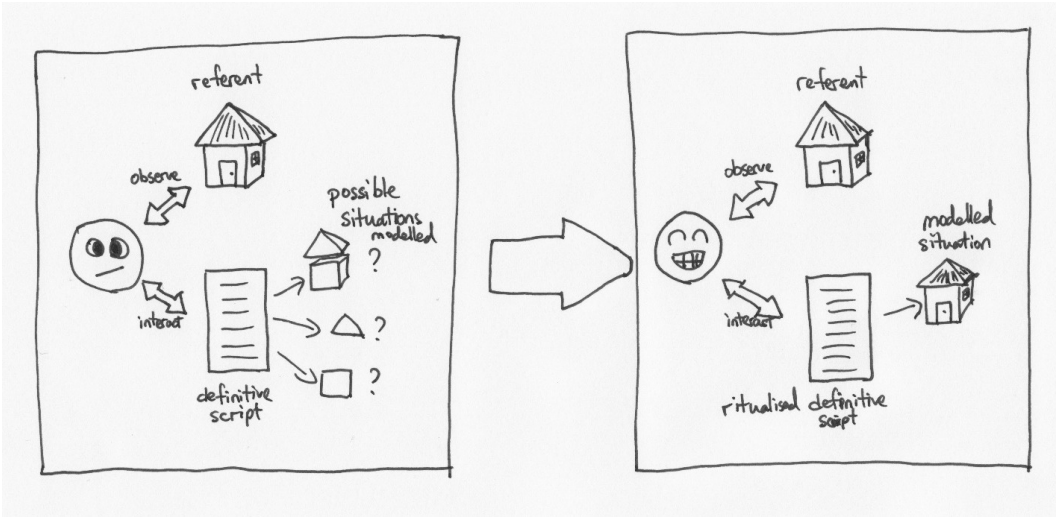

Figure 4.5 – Sense-making and ritualisation in EM[63]

[62] This figure is modified from (Beynon 2008).

4.1.3 The EM artifact

As I discussed in section 4.1.2, EM artifacts (or EM models) are typically represented by definitive scripts which make use of various definitive notations. However, it is a mistake to conceive EM artifacts as merely 'scripts'. The script as-is can merely reflect a snapshot of the current status of the artifacts and the historical definitions and re-definitions. They do not reflect the modeller's experience with the EM artifact and with the referent. Indeed, the interaction between the modeller and the EM artifact and between the modeller and the referent provokes 'meaning' for the interaction that 'scripts' cannot describe.

In traditional programming, the 'meaning' of a program is typically prescribed by its abstract behaviour in a particular context. In contrast, EM artifacts are *cognitive artefacts*, i.e. a tool that aids individual thoughts (Beynon and Cartwright, 1995). The meaning of EM artifacts is typically shaped by the modeller's personal experience through continuous observation of and interaction with the artifact and its referent, though it is possible to prescribe an EM artifact in the same way as in traditional programming. In the same vein, Beynon argues:

> *"EM construals do not require an intended functional specification or a specific context for observation and interpretation. This is a matter of ontology, and is significant despite the fact that an EM construal can exhibit system-like and program-like behaviour subject to restricted forms of interaction. Indeed, imposing a strict functional specification and specific modes of observation and interpretation upon an EM artefact subverts its character as a construal." (Beynon, 2009)*

To highlight the cognitive aspect of EM artifacts, Beynon (Beynon, 2008) adopts the term *construal* from Gooding (Gooding, 1990), where the term is used to refer to the pre-theory conceptual constructs that experimental scientists use to record and communicate their understanding of an observed phenomenon:

> *"Construals are a means of interpreting unfamiliar experience and communicating one's trial interpretations. Construals are practical,*

[63] This figure is inspired by (Beynon 2008) and (Harfield 2008).

> *situational and often concrete. They belong to the pre-verbal context of ostensive practices." (Gooding, 1990, p22)*
>
> *"... a construal cannot be grasped independently of the exploratory behaviour that produces it or the ostensive practices whereby an observer tries to convey it." (Gooding, 1990, p88)*

In Gooding's (1990) sense, construals are provisional, situated and often associated with tacit knowledge that is in a pre-verbal context that may only be conveyed through direct demonstrations (i.e. ostensive practices). To this end, King (King, 2007) summarised the essence of construal elegantly:

> *"… a construal is a particular understanding of a situation, "a communicable representation" developed empirically as the result of situated observation, interpretation and experiment. It is always subject to future embellishment and re-interpretation." (King, 2007, p.32)*

As mentioned earlier, an EM artifact is always open to changes in response to the modeller's new insight into the referent through continuous observations. These observations are based on the modeller's subjective and personal experience of the situation being observed. More importantly, this allows the modeller to explore through and within the model to find *an efficacious configuration* that reflects the state-as-experienced (by way of adding new definitions or changing the existing definitions in the model). Like Gooding's construal, EM artifacts are provisional, situated, tacit, interactive, and ostensive (cf. e.g. Beynon, 2008, Harfield, 2008). To exploit these characteristics, EM models are often viewed as *interactive situation models* (ISM), a term coined by Pi-Hwa Sun (Sun, 1999; Wong, 2003, Beynon et al., 1999). ISMs complement text-based documents and are to be used to construe *situations* in systems development (e.g. the situations in user-artifact interaction) and as "a communicable representation" to cultivate requirements (Sun, 1999). Figure 4.3 shows a snapshot of a jugs model (EMPA: jugsBeynon1988). Such a model can be thought of an ISM because it facilitates the analysis of the *use* situations for a pair of jugs. Besides being used to analyse interaction between users and artifacts, EM models have also been used in other domains. These include geometric modelling (Cartwright,

1999), creative product design (Ness, 1997), education (Roe 2002), systems development (Wong, 2003), and constructivist computing (Harfield, 2008).

From a systems development perspective, the status of an EM artifact may be considered to be dynamic[64]:

> **EM artifact as *artifact*** – when viewed merely as an artifact, an EM artifact is simply a source of experience for the modeller. That is to say, there are observables, dependencies, agents and agencies associated with the artifact, but there is not necessarily an external significance for the ODA patterns that the modeller encounters in the artifact.
>
> **EM artifact as *construal*** – when viewed as an construal, an EM artifact serves an exploratory and explanatory function. The observables, dependencies, agents and agencies typically have counterparts in some external phenomenon, but the correlation between experience of the artifact and experience of the phenomenon is provisional in character. The ODA patterns in the construal are subject to change as the modeller's understanding of the external phenomenon develops, and as the construal potentially acquires a clearly defined referent through further experimentation.
>
> **EM artifact as *model*** – an EM artifact can be viewed as an model when it can be placed in a context in which the ODA patterns it embodies are closely matched to an external referent. In this case, subject to discretionary interaction on the part of the modeller, there will be a precise correspondence between observables, dependencies, agents and agencies in the EM artifact and in its referent. For instance, interacting with the model may be a reliable guide to what happens when interacting with its referent, and vice versa.

[64] Notice that the stage of the EM model is not the same as the process stage as in systems development such as analysis, design, implementation, and maintenance. The ideas for this classification is drawing on two unpublished manuscripts written by Meurig Beynon: i) a working document Beynon prepared for an AHRC ICT Methods Network Expert Workshop held at Loughborough University in May 2008; and ii) a presentation manuscript Beynon prepared for the introductory talk for the Thinking Through Computing Workshop held at University of Warwick on 2-3 November 2007.

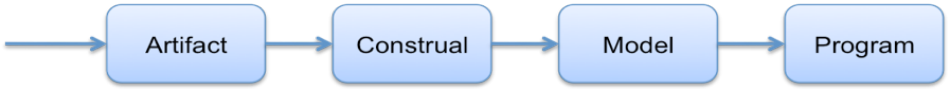

Figure 4.6 – The EM perspective on artifact, construal, model, program

EM artifact as *program* – an EM artifact can be viewed as an program when it can be placed in a context in which the ODA patterns it embodies are closely matched to an external referent that has the characteristics of a computing device. That is to say, within a suitable circumscribed context, there are patterns of interaction and interpretation with the artifact that can be reliably reproduced and that realise purposeful changes that can serve a 'computational' function.

Figure 4.6 depicts a possible systems development progression view of an EM artifact, from artifact to program. Note that the dynamic status of the EM artifact is not an indication that the EM development process has well-defined stages similar to those in traditional systems development, such as analysis, design, implementation and maintenance. It is inappropriate to conceive EM model-building as proceeding through transformation of an initial artifact to a construal, to a model, then to a program. The actual status of an EM model can be much more fluid and subtle. As argued previously, EM artifacts are always open for changes and new interpretations based on continuous observations and at the discretion of the modeller. The status of the artifact reflects possible views and interpretations that can be made by the modeller at any given time during the development, depending on the maturity of their understanding of the model and its referent and the context. At any one time, the status of a given artifact may sustain more than one interpretation, depending on how the modeller chooses to interact with it. In the richness and diversity of interrelated resources that inform the modeller's understanding, the scope of the EM process and model-building activities is analogous to lifelong learning (cf. Beynon and Harfield, 2007).

4.2 The philosophical foundation of Empirical Modelling

Much has been written in the EM literature about the philosophical foundations for EM. Relevant works (e.g. Beynon, 2005; Beynon and Russ, 2008; Beynon et al., 2008; King, 2007) relate to connections with William James's (1912) *Radical Empiricism*, David

Gooding's (1990) *Experiment and the Making of Meaning*, and various works by Brian Cantwell Smith, particularly his *Two Lessons of Logic* (Smith, 1987). Familiarity with these philosophical connections is most helpful in understanding this thesis, but a resume of the relevant ideas is beyond the present scope. The author has however made his own original contribution to scholarship on the philosophical foundation of EM with reference to Russell Norwood Hanson's (1958) *Patterns of Discovery* (cf. §4.2.1) as will be briefly outlined in the following.

4.2.1 The notion of observation and theory building

Among all the EM concepts, the notion of observation is arguably the most important – because all the activities in EM are either initiated or triggered by observation (cf. §4.1.1). From a philosophical perspective, observation is more than just visionary experience or *seeing with eyeballs*. Observation is a personal experience. In his influential work *Patterns of Discovery*, Norwood Hanson (1958) argues that observation is a complex and unified process that involves visionary sense-datum experience (i.e. "*seeing as"* in Hanson's terms) and conceptual organisation (i.e. "*seeing that"* in Hanson's terms). Based on his analysis of how physicists observe, he argues that *seeing as* and *seeing that* are indispensable and inextricably linked:

> *"Seeing is not only the having of a visual experience; it is also the way in which the visual experience is had." (Hanson, 1958, p.15) "'Seeing as' and 'seeing that' are not components of seeing, as rods and bearings are parts of motors: seeing is not composite. … 'Seeing as' and 'seeing that', then, are not psychological components of seeing. They are logically distinguishable elements in seeing-talk, in our concept of seeing." (Hanson, 1958, p.21)*

Hanson's argument is that in observation seeing and thinking are intertwined. This is in contrast to the traditional analytical philosophy which treats the mind (i.e. thinking) and the body (i.e. the sensory experience) as separable. Hanson further argues that observation is actually shaped by the observer's prior knowledge of the phenomena (i.e. is *theory-laden* in Hanson's terms). In the other words, the observer knows what to observe prior to the observation being made, and what can be observed is influenced by the 'theory' (e.g.

knowledge, trainings, experience, etc.) that the observer already has. Following Hanson's argument, observation is not an equivalent process to all people: the *interpretations, connexions*, and conceptual organisation involved in the observation are different for all people.

EM is concerned with modelling through a notion of observation that is analogous to what is described by Hanson. In EM, observation is intricately intertwined and inseparable within the EM process. The EM process is advanced through the modeller's insights that are provoked within the context of observation. Through observing, experimenting and interacting with the EM model and its referent in the real world, the knowledge of the referent in the real world and the knowledge of the EM model can inform each other in the modeller's head.

The notion of observation as described here is relevant to abductive reasoning. Hanson (1958) argues that abduction (also known as retroduction) (cf. Peirce, 1931-1958) plays a crucial role in the scientific theory building process and that the influence of abduction is often overlooked in most hypothetical-deductive (H-D) accounts of physical theory[65]. He examines the experimental process that leads to the discovery of facts. He argues that, contrary to H-D accounts, which presume that a hypothesis is the primary ingredient in all experiment, many scientific theories are actually 'developed backwards'. That is to say, the scientist begins with the results, not from a hypothesis, and usually seeks an explanation of his/her observation of the results. Hanson studies the process that eventually led Galileo to formulate the conception of constant acceleration between 1604 to 1632. He further argues that Johannes Kepler was in fact working bottom up from Tycho Brahe's data, from "explicanda to explicans", when he discovered that Mars' orbit was actually elliptical (Hanson, 1958, p.85). These examples of scientific theory development lead Hanson to assert that scientific reasoning and the process of theory building do not follow a hypothetical-deductive system:

---

[65] Peirce writes "[induction] sets out with a theory and it measures the degree of concordance of that theory with fact. It never can originate any idea whatever. No more can deduction. All the ideas of science come to it by the way of Abduction. Abduction consists in studying facts and devising a theory to explain them. Its only justification is that if we are ever to understand things at all, it must be in that way. Abductive and inductive reasoning are utterly irreducible either to the other or to Deduction, or Deduction to either of them …" (quoted in Hanson 1958, p.85)

> *"By the time a law has been fixed info an H-D system, really original physical thinking is over. The pedestrian process of deducing observation statements from hypotheses comes only after the physicist sees that the hypothesis will at least explain the initial data requiring explanation. This H-D account is helpful only when discussing the argument of a finished research report, or for understanding how the experimentalist or the engineer develops the theoretical physicist's hypotheses; the analysis leaves undiscussed the reasoning which often points to the first tentative proposals of laws" (Hanson 1958, p.70-71)*
>
> *"H-D accounts all agree that physical laws explain data, but they obscure the initial connexion between data and laws; indeed, they suggest that the fundamental inference is from higher-order hypotheses to observation statements. This may be a way of setting out one's reasons for accepting an hypothesis after it is got, or for making a prediction, but it is not a way of setting out reasons for proposing or for trying an hypothesis in the first place." (Hanson, 1958, p.71)*

For Hanson, the intellectuality of empirical science actually involves both discovering the phenomena and working out a H-D account to convince others to accept that account as an objective explanation. The intentional character of EM, as an approach to computer-based modelling, is precisely concerned with discovery and identification of the agents in the context of observation that is analogous to the discovery process in empirical science. As Beynon and Russ (Beynon and Russ, 2008) pointed out, EM has been developed as an approach to fit the purpose of *pre-theory* experimentation with computing. Such interest in pre-theory experiments indicates that EM pays attention to the importance of abductive influences in the creation of innovative artifacts that is subsumed in the systems development process. As mentioned in section 4.1, the EM process does not predefine any path for exploration. It cannot (and should not) be seen as an approach that exclusively follows a deductive or an inductive approach to computer-based modelling.

4.3 Characteristics of practical EM

Despite the fact that the central activity of EM is modelling with program-like definitive notations, EM is often disregarded as an approach for programming. This section concerns the practical (and technical) differences between EM and common programming paradigms in various aspects. These include conceptual constructs, instant feedback, dialogical interaction, openness and flexibility, and experimentation. This is not meant to establish a dichotomy between EM and other programming paradigms, but rather, to help the reader to understand EM. This is done by comparing, contrasting and relating EM concepts to concepts in other programming paradigms.

4.3.1 The tkeden tool

Tkeden is the primary modelling tool for EM. It is a modelling environment which supports the core EM activity – *modelling with definitive scripts* (cf. §4.1.2). Tkeden is developed using the C language and the Tcl/Tk toolkit, and it runs on various platforms[66] (Ward, 2004). Perhaps the most noticeable and most discussed[67] feature of the tool is its dependency maintainer. The built-in dependency maintainer keeps the conceptual integrity of the model by maintaining the atomicity of the state-changes among observables within an EM model (cf. Wong, 2003), as they are experienced by the modeller when interacting with the referent in the sense of everyday concurrency (cf. §4.1.1).

The development of tkeden can be dated back to 1987, when Y. W. Yung developed EDEN (the Evaluator for DEfinitive Notations) in his final-year undergraduate project. Yung's (Yung, 1990) EDEN later became ttyeden, which is a terminal-based EDEN interpreter and the precursor of tkeden (Ward, 2004). The first version of tkeden was developed by Y. P. Yung in connection with his post-doctoral research (Yung, 1996; Ward, 2004), and there have been a number of subsequent contributors including Pi-Hwa Sun (1999), Ashley Ward

[66] These include Solaris SunOS Linux, MacOS, MacOS X, and Microsoft Windows

[67] Much of the literature in EM has discussed, and more or less emphasized, this feature in comparison with other programming languages and environments, e.g. (Wong 2003), (Heron 2002), (Sun 1999), (Yung 1993).

(2004), Antony Harfield (2008), Charles Care, Russell Boyatt, and myself[68].

Figure 4.7 shows the current tkeden modelling environment (version 1.71) loaded with the jugsBeynon2008 model, a standard example of an EM model. The environment includes an input panel for the modeller to interact with the model in a command-line fashion, a console panel for text-based responses (e.g. the definition of an observable when queried), a window for displaying the graphical components of the model[69], and a window for displaying the interaction history. It is also possible to invoke a window for displaying the current set of definitions that constitute the model. As shown on the toolbar of the input panel in figure 4.7, the current tkeden modelling environment, by default, pre-loads four core definitive notations when it is invoked, namely, EDEN, DoNaLD, SCOUT, and AOP:

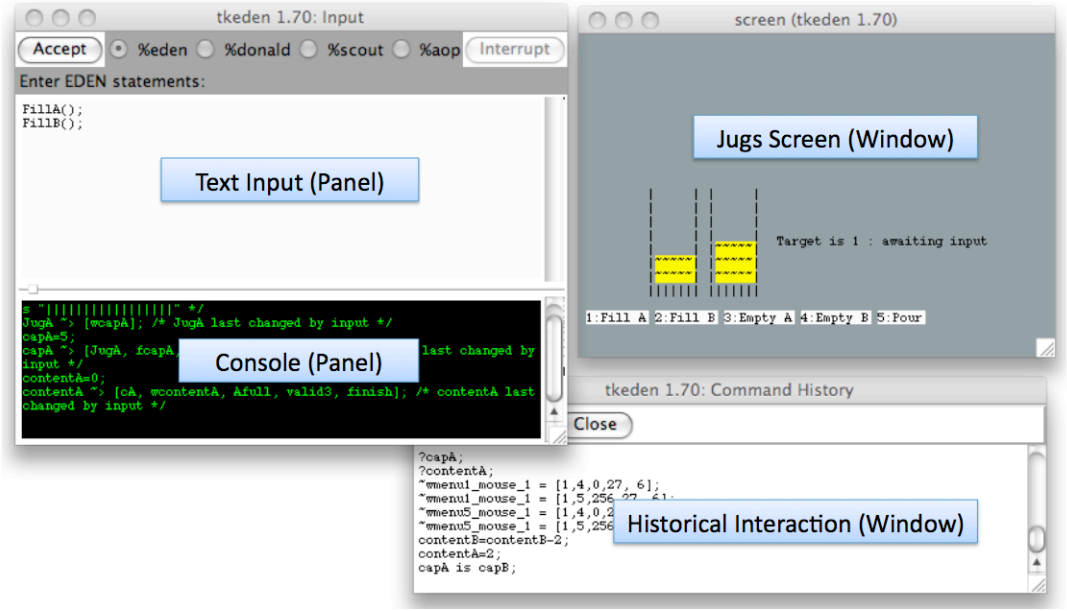

Figure 4.7 – The tkeden modelling environment loaded with the Jugs model (EMPA: jugsBeynon2008)

[68] My interest has been in improving and extending dtkeden (the distributed version of tkeden which is developed by Pi-Hwa Sun for distributed modelling) for collaborative modelling purposes.

[69] In theory, the number of screens of a model can vary from zero (i.e. no graphical components) to the max number that is allowed by Tcl/Tk and the host operating system.

The ***EDEN*** notation is a general purpose language which supports the principles of EM. It was initially developed in Y. W. Yung's (1987) final-year undergraduate project. Technically, the EDEN language has a C-language-like syntax and it allows the modeller to construct models with a mix of definitive constructs (i.e. observables, dependencies, and agents) and procedural constructs (i.e. sequential functions and procedures). However, procedural constructs can be re-defined in Eden (the existing definition of an identifier is *replaced by a new definition* rather than *overloaded* in an object-oriented language such as C++) (cf. Yung and Yung, 1988).

DoNaLD (a Definitive Notation for Line Drawing) notation was developed by Steve Hunt and Tim Bissell on the basis of the ideas introduced by David Angier in his final-year undergraduate project[70] (cf. Wong, 2003; Beynon et al., 1986). Unlike Eden, DoNaLD is a pure definitive notation, and allows observables to be arranged in a hierarchical structure (cf. Beynon et al., 1986). Line drawing agents may consist of primitive agents (e.g. *line*, *shape*, *arc*, *circle*, *ellipse*, *rectangle*) and other line drawing agents, and they are grouped into *viewports* (i.e. a drawing canvas).

SCOUT (the definitive notation for SCreen layOUT) was originally developed for describing screen layout, i.e. prototyping text-based user-interfaces, in Y. P. Yung's (Yung, 1988) final-year undergraduate project. However, the usage of SCOUT soon exceeded its intended scope. It has been used in various ways not initially envisaged by the original author of the notation. For example, it has been extensively used to specify panels to arrange information on the screen (EMPA: projecttimetableKeen2000, racingGardner1999), and as a presentation tool (EMPA: drawSlideKing2004). This exemplifies the way in which the users may 'redesign' the use an EM artifact in ways which it is hard, if not impossible, for the 'designer' to limit.

The ***AOP*** notation[71] is a definitive notation which allows the development of new

[70] The DoNaLD interpreter was first implemented by Y. W. Yung in 1987, and subsequently enhanced by Chan (1989) and Parsons (1991).

[71] The Agent Oriented Parser (AOP) was originally developed by Chris Brown (2001) in his final-year

notations without re-developing tkeden source code, or even on the fly! (cf. Harfield, 2002; 2008) For instance, in an undergraduate project, Graeme Cole has developed a notation for the player to interact with the wumpus game (EMPA: wumpusCole2005).

An EM model often uses various definitive notations. For example, King exploited both EDEN and SCOUT in the Sudoku model (EMPA: sudokuKing2006). Beynon exploited EDEN, DoNaLD, and SCOUT in the Lift model (EMPA: liftBeynon2003). It is worth emphasizing that additional notations (e.g. EDDI, a definitive notation for database modelling) can be loaded on demand.

Further discussion of the history and the technical implementation of the tool are beyond the scope of this thesis. However, a more detailed discussion of its history and its implementation is available in Ashley Ward's (Ward, 2004) doctoral thesis. Ward was the primary developer of the tool between 1996 and 2005.

4.3.2 Conceptual constructs

Petre and Winder (1990) described a spectrum of programming languages between two extreme paradigms, imperative and declarative. While object-oriented and procedural languages fall into the imperative paradigm, functional and logic programming languages are declarative. Definitive notations (in EM), however, are neither imperative nor declarative, but somewhere between these two extremes (Yung, 1993). In section 4.1, I have discussed the basic concepts in EM. Unlike previous comparisons (e.g. Yung, 1996; Heron, 2002; Wong, 2003; Harfield, 2008), the following discussion does not contrast *modelling with definitive notation*s (MwDN) with individual programming paradigms in detail (as they cannot be 'compared' effectively because they are based on a different conceptual foundation), nor compare EM with programming in general. Instead, I relate, compare, and contrast conceptual constructs in EM's definitive notations with the closest conceptual constructs in other programming paradigms (cf. table 4.8).

undergraduate project, and enhanced by Antony Harfield (2002). It was turned into the aop notation by Charles Care in 2005, and was incorporated into tkeden (with minor modifications) by Antony Harfield in 2005 (cf. http://www2.warwick.ac.uk/fac/sci/dcs/research/em/notations/aop/).

Conceptual constructs in definitive script (EM)	**The closest conceptual constructs in other programming paradigm**
Observable	Variable in imperative programming paradigm. A cell in a spreadsheet.
Definition (or dependency)	Formula in spreadsheet.
Agent	Classless object in a prototype-based programming paradigm.
Virtual agent	Classless object in multiple namespaces in a prototype-based programming paradigm.
Triggered action (agent)	Event handler in an event-driven programming paradigm.

Table 4.8 – Conceptual constructs in EM versus the closest conceptual constructs in other programming paradigms

Observable

Technically, an observable in a definitive notation resembles a variable in an imperative programming paradigm; they are both a kind of identifier for some objects in a program (in the former) and in a model (in the latter). Furthermore, values can be assigned to both variables and observables. However, the identification of a variable is in relation to an abstract context, whereas in definitive notation (or EM) it is in relation to the experience of a human observer with reference to some referent in the real world (cf. §4.1.1).

Another point of technical difference between a variable and an observable is that the latter can be assigned with a definition (or dependency). In this sense, an observable is similar to a cell in a spreadsheet, whose value can be assigned using a formula involving other data values. However, an observable is type-less (e.g. in EDEN), which is different from a cell in a spreadsheet.

As mentioned earlier in section 4.1.1, the intended character of observable in EM is much broader than variables in traditional programming. Definitive notations address this concern by allowing the assignment of a definition to an observable. Since the definition of an observable gives information about how the state-change of the observable is produced and how it relates to other observables in the model, this can be thought of as giving a ‘meaning’ to the observable (Yung, 1993). Traditional program variables do not necessarily have a ‘meaning’ in this sense.

(a) (b)

Figure 4.9 – Cascading definitions and cyclic definitions in EDEN

Definition (or dependency)

As mentioned above, a definition (or dependency) in definitive notation is technically similar to a formula in a spreadsheet application. State-changes between the observables on the left hand side and the right hand side of the definition (cf. §4.1.1) are maintained as indivisible actions, in a similar fashion as transactions in data manipulation languages in databases. As in many spreadsheet applications, definitions can be cascaded but not cyclic. For instance, it is possible to have the EDEN definitions as in figure 4.9a, but not as in figure 4.9b.

Agent

If agents are seen as a natural extension of objects (cf. the discussion in (Wong, 2003)), perhaps the closest conceptual construct to an agent in EM is the classless object in a prototype-based programming language (cf. Noble et al., 2001). One example of such programming language is Javascript. In contrast to the manipulation (i.e. creation, inheritance, reuse) of class-based objects in object-oriented programming, manipulation of classless objects is done through cloning of existing objects in the program. However, the cloning process in definitive notations in EM is not the same as the cloning as in, e.g. Javascript. While the latter is supported by built-in functions in the language, cloning of agents in definitive notations needs to be done manually by the developer. The potential advantage of the classless structure is that the developer (or modeller) does not have to plan the object structure in the modelling process in advance, and the manual cloning makes every detail of a cloned agent explicit – which is necessary in EM, where the semantics of all features in the model is to be derived from their counterparts in the referent. The disadvantage is that such explicit cloning can make maintaining large models difficult (King, 2007).

```
proc yield : contentA
{
        writeln("The content of Jug A has changed !");
}
```

Figure 4.10 – Triggered action in EDEN

Virtual agent

The *virtual agent* is a special kind of agent in EDEN that was introduced by Pi-Hwa Sun in relation to his work in dtkeden (a distributed version of tkeden). A virtual agent in EDEN has the form:

virtualAgentName_identifier

where *virtualAgentName* is the name of the virtual agent, the underscore character (viz. '_') is the delimiter for virtual agent in EDEN, and *identifier* is the observable, dependency, or name of an agent within the context of the virtual agent. For Sun (1999), virtual agents are used as a mechanism for the human agent at the dtkeden server to "introduce definitions in a context *as if* they were being generated by an agent in a similar fashion" (Sun, 1999, p.171, italics in original). Therefore, virtual agents can be thought of as classless objects in multiple namespaces, as in prototype-based programming paradigms.

Triggered action

The EDEN notation, unlike the auxiliary pure definitive notations in the tkeden environment, supports triggered actions. Technically, a triggered action is similar to an event handler (also known as listener in some language) in event-driven programming paradigms – the event handler is invoked when a given event is raised in another thread within a program. Such event handling mechanisms are typically used to process input/output data (e.g. file access) or graphical user interface events (e.g. mouse clicks).

In EM, a triggered action is used to represent an agent that responds to a particular state-change of an observable. For instance, the EDEN fragment in figure 4.10 depicts a triggered

action *yield* that will print a message to the console when the content of Jug A changed[72]. A triggered action can be thought of as an event handler because it responds to a particular state-change and handles multiple state-changes of various observables in a similar fashion as event handler. The qualities of triggered actions in EDEN are much enhanced by their use in close conjunction with definitive scripts. The appropriate use of triggered actions in conjunction with definitive scripts typically rationalises the way in which updates of observables are synchronised and greatly simplifies the maintenance of triggered actions.

The above sub-sections have reviewed the features of the tkeden tool and associated conceptual constructs. As is depicted in Figure 4.14 (on page 130), the qualities of EM as a modelling approach stem from key technical characteristics. These include – the fact that EM scripts are interpreted rather than compiled, that they exploit definitive notations rather than conventional procedural commands, and that sets of definitions entered into the model serve as a record of the interaction history. To help the reader to appreciate the distinctive character of EM, I shall now discuss specific qualities of the modelling approach in such a way as to expose the contribution that each of these technical characteristics makes.

4.3.3 Instant feedback

The tkeden modelling environment potentially offers instant feedback to the modeller as she develops her construal. The term *instant feedback* here refers to the fact that some state-changes of the model (or some actions) are visible to the modeller promptly or within a short period of time after a redefinition is input to tkeden.

This instant feedback feature can be attributed to the fact that definitive notations are interpreted, i.e. that tkeden animates the EM models by interpreting the definitive scripts, instead of compiling. Figure 4.11 illustrates the difference in feedback cycles between interpretation and compilation. In the case of interpretation, the program (or script) is analysed, parsed, and compiled in one step. In contrast, the program is stored in a machine executable form. The programmer (or modeller) will have to invoke a compiled program (or model) after compilation. Despite the better code security and performance that compilation

[72] Assuming the Jugs model (EMPA: jugsBeynon2008) is loaded in the tkeden modelling environment.

may offer, the time to obtain feedback is significantly slowed down.

In principle, any programming or modelling languages (or notations) can be interpreted. Therefore, languages that use interpreters may provide a certain degree of instant feedback. However, this potential further depends on the time to interpret the script (i.e. the interpretive overhead). Due to the interpretive overhead, the turnaround time can be significant if the script to be interpreted is huge. This becomes apparent, for instance, in database query languages (e.g. SQL) interpreters in which the turnaround time (i.e. the time to resolve a given query) is highly dependent on the size of the data set to be searched and the complexity of the query.

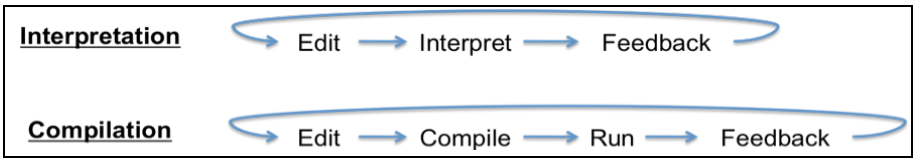

Figure 4.11 – The difference in feedback cycles between the interpretation and compilation

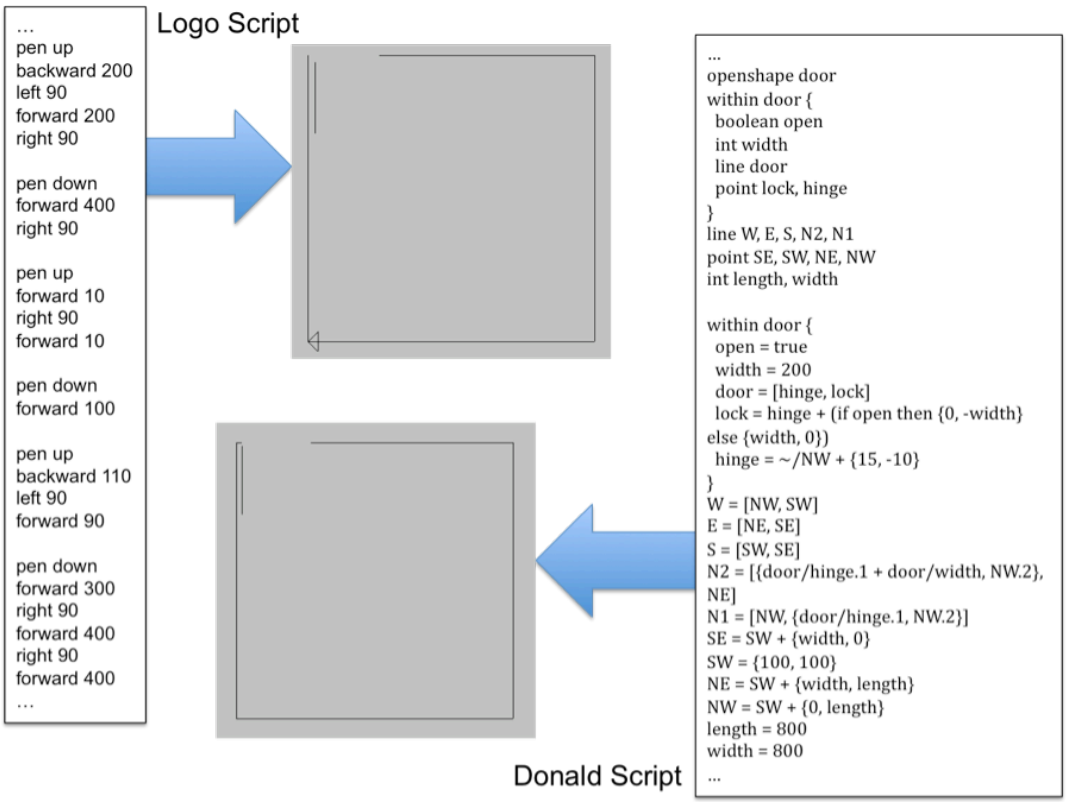

Figure 4.12 – Logo script vs. Donald script

The virtues of EM are not captured fully by merely considering the instant feedback that an interpreter offers. In this context, the special characteristics of EM are just as important. To illustrate this, consider, for instance, the small sequence of Logo commands that serves to define the outline floor plan of a room with a door with a Donald script that specifies the same drawing (cf. figure 4.12). Even though the sequence of Logo commands is interpreted, so that the modeller sees the line drawing being built up as the commands are entered, the sequence of commands is to be understood by the modeller as a recipe for drawing a floor plan. There is nothing comparable in the Logo script to the direct correspondence between observables in the Donald script and actual observables in the room. In modelling with the definitive script, the correspondence that is set up in the modeller's mind is then not between a recipe and the line drawing it creates, but between a dependency structure in which the elements (e.g. points, lines, attributes etc) have direct counterparts (e.g. as corners, walls, doors, widths etc) in the domain in which the line drawing is to be interpreted. This greatly enhances the semantic value of the instant feedback.

In addition, whereas the scale of revision necessary to adapt the line drawing recipe to make a meaningful change to the floor plan – e.g. to change the dimensions of the room, to make the door thinner, or open or close the door is quite hard to predict (and depends critically upon the particular drawing recipe developed), the redefinition of an observable is typically an update that is constrained in size. In this respect, EM encourages the modeller to make small changes incrementally rather than to make the batch changes, as is typically the norm in traditional programming due to the overhead in compiling or interpreting the program.

4.3.4 Dialogical interaction

The use of an interpreter not only potentially offers instant feedback; it changes how the modeller (or programmer) interacts with the model (or program). Because the use of an interpreter has shortened the interaction cycle (cf. figure 4.11), it encourages the modeller to interact with the model more seamlessly as in dialogical conversations in an everyday sense. As the use of interpreter removes significant time for compilation from the interaction cycle, the modeller switches between the development context and the use context more frequently than is the case with compilers. Furthermore, environments designed for

interpreting programming languages often blur the distinction between the development context and the use context. For instance, a typical Logo environment offers the modeller direct access to the Logo script both for editing and running. In this respect, the use of interpreter promotes a conflated context of development and use. As observed by Bornat, this conflation of context is crucial for experimentation:

> *"In an environment where program design is constantly changing, where programmers are experimenting with the design of programs or where people are simply learning to program, an interpreter certainly saves human time and effort because it permits more effective interaction with the object program than is possible when the program is compiled." (Bornat, 1990, p. 356-357)*

Apart from the use of interpreter, the tkeden modelling environment provides a number of other features that promote dialogical interaction between the modeller and the model. In the first instance, these can be seen as stemming from technical advantages of modelling using spreadsheet-like definitive notations rather than program-like procedural commands:

> **Interactiveness of definitive notation** – In practical EM, the unusual qualities of the dialogical interaction that is achieved arguably stem from the non-sequential nature of a script of definitions, that is to say, the fact that the order of the definitions is not significant. In contrast, the order of constructs in other programming paradigms, particularly in the procedural programming paradigm, can be significant. This flexibility of the notation structure affects how the modeller interacts with the model. For instance, 'redefining' a statement in an object in an object-oriented programming paradigm implies a reload of the containing object, if not a reload of the entire program. The significance of definitive notations in relation to flexibility will be discussed further in section 4.3.5.
>
> **Interactive prompt** – The nature of the interactive prompt is similar to that of command shells in some operating systems (e.g. Linux, MacOS, DOS), in which users can create, interact, and amend 'objects' that are managed by the operating system. In practical EM, the modeller can manipulate the definitions of the EM

model, interact with the EM model through redefinitions and, in addition, query the definitions of observables within the same modelling environment through the tkeden input panel (cf. figure 4.7). Interactive prompts are more common in databases, in which a database administrator can define, amend, and query data and the data model within the database through the interactive prompts without restarting the database. Some functional programming environments, e.g. Miranda, also provide an interactive prompt for the user to manipulate specifications interactively, and in the case of hybrid environments such as Mathematica allow such manipulation to be expressed visually through changes to graphics. In contrast, most programming environments for procedural programming languages do not provide an interactive prompt for the programmers to manipulate or query constructs on the fly, except with run-time debuggers. However, the programmers can only make changes to the program source code and reload the program to see the difference, or make changes to the objects which may not persist when the program is terminated in debuggers, rather than interact with the same program representation as in the development (i.e. non runtime) context.

Interaction history – Interaction history potentially helps the modeller to make sense of a particular pattern of interactions with the model and to revisit a particular scene. For this reason, interactive environments, such as those for database query languages, may capture interaction history in log files. However, the interaction history that features in the use of EM is unusual in being of the kind which can be immediately fed back to the same interpreter for exploration[73]. In the tkeden modelling environment, interaction history consists of redefinitions which are either entered by the modeller through the input window or triggered by other agents in the model. Interaction history plays a crucial role in modelling with definitive script, as - reflecting a strong form of conflation of development with use - it can be viewed as

[73] To appreciate the advantages of the script, contrast the potentially wild effect of randomly choosing and re-executing a subset of commands from the Logo recipe in figure 4.12 (on page 124) with that of randomly choosing and re-interpreting a subset of definitions from the associated Donald script (which has no effect irrespective of the choice of subset and the order of interpretation).

itself part of the definitive script (cf. §4.1). In fact, the interaction history is the definitive script if the modeller initiates the modelling with an empty set of definitions rather than resumes a modelling session from a set of previously saved definitions. In tkeden, the interaction history is available to the modeller through the interaction history window (cf. figure 4.7).

4.3.5 Openness and flexibility

Openness to further exploration and interpretation (i.e. sense-making), and flexible structure are two interrelated characteristics of an EM model (cf. §4.1). In practical EM, the openness and flexibility of EM model are realised with the use of an interpreter and definitive notations in the tkeden modelling environment. In contrast to the use of compilers, where the program (or model) have to be compiled and/or encrypted into some form of low level human unreadable text, the use of an interpreter means that the script has to be *open source*[74] in principle. This makes the script always open to change.

Figure 4.13 – The difference between definitive notation and procedural programming

The virtue of a definitive notation is that the order of the definitions has no significance to the state of the model. As a simple example, figure 4.13 illustrates this character of definitive notation with two fragments of definitive scripts and two code fragments in procedural programming. The use of definitive notations in practical EM gives the modeller a flexible structure to construe a situation, in which, new definitions can be added to the definitive scripts and redefinitions can be made on the fly, or at any time during the modelling process. This can be done through the interactive prompt in tkeden modelling environment (cf. §4.3.1 and §4.3.4). Furthermore, the modeller can interact with the definitive script at the definition

[74] The term “open source” is here used in its literal sense to refer to the situation where the source code of a program (or model) is unencrypted and available to examination and modification by anybody who is interested.

level – a lower level of abstraction in contrast to, e.g. an object-oriented programming paradigm[75], where the data structure (or data model) and the methods for manipulating the data are often encapsulated in the same object. In such a case, it is not easy to make changes to individual data structure without affecting their common parents (i.e. classes). Indeed, the flexible structure in definitive notation eases the exploratory and experimental model-building activities in EM.

4.3.6 Experimentation

As mentioned at the beginning of this chapter, experimentation is one of the key motives for EM. Beynon and Russ (2008) argue that EM provides a holistic conceptual framework for *exploratory experiment*. In the journal paper *"Experimenting with Computing"*, Beynon and Russ (2008) distinguish two different kinds of experimental activity: post-theory and exploratory. In post-theory experiment, the context for observation is usually stable, and the outcomes can readily be observed when the key observables are changed. There is often a reliable basis for predicting criterion for successful outcome, whether drawn from theory or previous experience (Beynon and Russ, 2008). Despite the fact that there is some degree of human intelligence involved in setting up an experiment of this such kind, human interpretation is often a "marginal element" (Beynon and Russ, 2008). In contrast, the context for observation in exploratory experiment is preliminary and provisional, and human engagement in this context is crucial.

Beynon and Russ (2008) argue that it is in the context of exploratory experiment that the term 'experiment' has its authentic meaning, namely, "taking an action whose effect is unknown". They distinguish such a context sharply from the context for post-theory experiment. In the context of exploratory experiment:

[75] The only exception is when the statement is enclosed in a procedure in the EDEN notation, in which case the modeller may have to redefine the entire procedure. This is due to the fact that EDEN is a hybrid notation that consists of both definitive and procedural constructs. However, there is no such restriction in other definitive notations such as DoNaLD or SCOUT.

> *"… the effect of an action cannot possibly be exactly predicted – indeed, it cannot even be uncontroversially identified or comprehensively registered."*
>
> *(Beynon and Russ, 2008, p.477)*

Consequently, the support for exploratory experimentation is not straight forward in contrast to the support for post-theory experimentation, where the experiment can be set up with reference to suitable formalised procedures when certain key observables have been identified (Beynon and Russ, 2008).

Taken together, the various characteristics of tkeden discussed in the previous sub-sections (cf. §4.3.2 ~ 4.3.5) can be seen as supporting the use of EM for experimentation. Figure 4.14 depicts how these techniques and approaches support different characteristics of practical EM and how they give support for exploratory experimentation using EM.

Arguably, programming paradigms such as procedural, functional, and object-oriented fail to offer the qualities that definitive notations potentially afford for exploratory experimentation. For instance, procedural programming does not allow flexible structure (cf. §4.3.4); object-oriented programming maintains a high level of abstraction which may restrict effective interaction (cf. §4.3.4) and flexible structure (cf. §4.3.5); functional programming offers a poor correspondence between everyday observables and variables in the program which makes interaction and comprehension difficult (cf. Beynon and Russ, 2008).

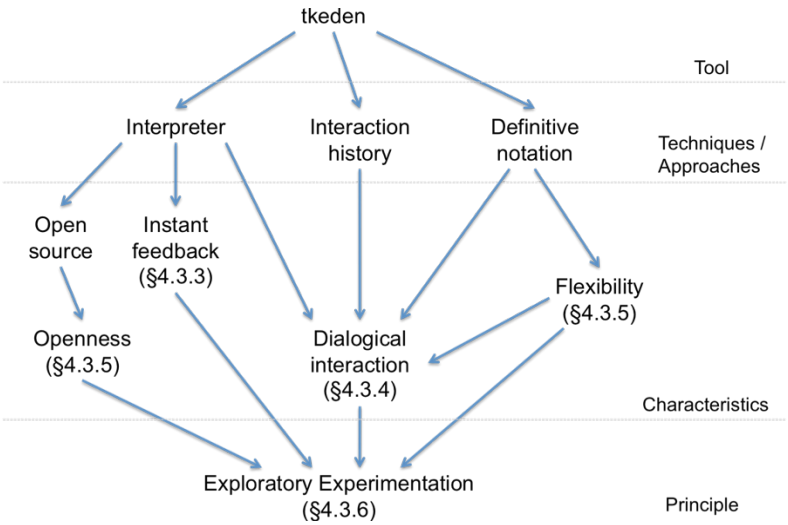

Figure 4.14 – Potential influence between instant feedback, dialogical interaction, experimentation, openness and flexibility

Experimentation not only needs support from programming paradigms, but also the support from the tool (or environment) as well. For instance, Javascript is an interpretable language, but it is very hard, if not impossible, to experiment with the script in the web browsers because they do not normally provide support for such interactions[76]. By contrast, spreadsheet environments provide much better support. A spreadsheet potentially offers instant feedback, openness and flexibility for creating numerical models. However, it does not provide an interaction history so that the modeller has no idea how he has reached the current situation.

In summary, I argue that current EM practice with the tkeden modelling environment potentially offers better support to exploratory experimentation. In the next chapter, I will discuss how EM can be practiced in collaborative modelling, an activity which I claim is central to groupware development (also cf. chapter 7).

[76] An interactive prompt is not a common feature and is not usually available in web browsers. Furthermore, the javascript console, if provided, is merely for displaying error messages, and not for taking input from the developer.