Before and Beyond Systems:
An Empirical Modelling Approach

by

Allan Kai Tung Wong

Thesis

Submitted in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy in Computer Science

Department of Computer Science
University of Warwick

January 2003

Contents

List of Figures

List of Listings

Acknowledgements

First of all, I would like to thank Dr. Meurig Beynon for his continuous support not only as an ordinary supervisor but also as a teacher, a mentor and a friend who has given me tremendous encouragement throughout my years of academic research. I have learnt more than I could write down in this thesis.

I would like to thank Dr. Steve Russ who has given me much invaluable advice on research direction and has helped to review the contents of this thesis.

Many thanks are also due to my colleagues, Ashley Ward, Chris Roe, Dominic Gehring, Jaratsri Rungrattanaubol, Richard Cartwright, Pi-Hwa Sun, Soha Maad, Suwanna Rasmequan and Yih-Chang Chen, in the Empirical Modelling research group for many stimulating discussions in seminars and meetings.

Last but not least, I would like to thank all my friends and family for supporting me throughout these rewarding years.

* * * * * * * *

Further thanks go to Prof. Chrystopher Nehaniv for giving me useful comments on the final version of this thesis.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated.

Abstract

Empirical Modelling (EM) is an approach to the construction of *interactive computer-based artefacts* that embody *construals* of phenomena. The modelling activities underlying EM are associated with the identification of *agents*, *dependencies* and *observables* that reflect a commonsense way of understanding phenomena as experienced in everyday life. Model construction is guided by heuristic *observation*, *interaction* and *experiment* rather than by rational prescribed steps.

This thesis is a comprehensive study of the potential of EM as an approach to system development. EM rests on a philosophical foundation that is radically different from that underlying conventional approaches to system development. This is reflected in the reference to two perspectives on systems in the thesis title: *before* and *beyond* systems. The 'before systems' perspective is concerned with the extensive preliminary activities that help to inform the system conception. These activities are supported by building models as prototypes to address the personal and interpersonal demands of system development. The 'beyond systems' perspective is concerned with fulfilling the functions of a system without the circumscription characteristic of traditional systems. This has particular relevance for potential applications of EM to ubiquitous computing, where system conception is inseparably linked with system use in a situation.

This thesis seeks to consolidate and extend diverse concepts drawn from previous EM research within a unified framework: the Definitive Modelling Framework (DMF). The DMF supplies a suitable setting for both the *cognitive* and the *collaborative* aspects of system development in which the emphasis is on *heuristic human problem solving* and *maintaining conceptual integrity in a system design*.

Evaluations of software tools that support modelling within the DMF are conducted. The prospects for future tool development are extensively studied and illustrated by the construction of three new prototypical visual tools.

The research reported in this thesis provides a solid foundation for future research on applying EM to meet the challenges of system development in a modern computing context.

Keywords:

Empirical Modelling, agent-oriented modelling, system development, dependency, conceptual integrity, human problem solving, ubiquitous computing, user interface, visual modelling tool

Abbreviations

ADM	Abstract Definitive Machine
AR	Augmented Reality
CPSS	Construal of Problem Solving Situation
DMF	Definitive Modelling Framework
DMT	Dependency Modelling Tool
Donald	Definitive Notation for Line Drawing
DTkEden	Distributed version of TkEden
Eden	Evaluator for Definitive Interpreter
EM	Empirical Modelling
EME	Empirical Modelling Environment
GUI	Graphical User Interface
ICM	Interactive Control Model
IDM	Interactive Device Model
ISM	Interactive Situation Model
LSD	Language for Specification and Description
OO	Object-Oriented; Object-Orientation
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
PSE	Problem Solving Environment
Scout	Definitive Notation for Screen Layout
SICOD	Soft Interfaces for the Control of Devices
SQL	Structured Query Language
SSADM	Structured System Analysis and Design Methodology
TkEden	Tcl/Tk version of Eden
UML	Unified Modelling Language
VR	Virtual Reality
WING	Windowing and Graphics tool

1 Introduction

This introductory chapter gives the reader an orientation for this thesis. Section 1.1 describes the background to Empirical Modelling (EM). Section 1.2 describes the motivations and aims that underlie the research reported in this thesis. Section 1.3 outlines the contents of the thesis. Section 1.4 discusses the principal contributions of the research described in this thesis to EM.

1.1 Background

The Empirical Modelling Project is an ongoing research programme, based in the Computer Science Department at the University of Warwick. The project was initiated by Dr. Meurig Beynon in 1981, and has subsequently received sponsorship from British Telecom, IBM, Matra Datavision and the BBC, and conducted collaborative research in association with universities in Europe, Japan, Russia and the US. The published output of the project includes over 100 publications in the form of international conference and journal papers, research reports and doctoral theses. In addition, several software tools have been developed, together with several hundred models generated through postgraduate and undergraduate project work to study the potential practical applications of EM.

A major theme of the project is studying principles and building tools for constructing ‘construals’ of phenomena, typically as special computer-based models. The emphasis is not on making a product, but on model-building activities that assist the personal cognitive process of understanding phenomena and the interpersonal sharing of understanding. The choice of the epithet ‘empirical’ reflects the fact that the modelling activities are rooted in observation, interaction and experiment, and in this respect are distinguished from perspectives that are associated with formal methods and traditional mathematical models.

A radical distinction between EM and many other approaches to computer-based modelling is its explicit emphasis upon modelling state as directly experienced rather than behaviours as circumscribed. Key modelling activities are associated with the identification of agents, dependencies and observables that arguably resemble our commonsense way of understanding phenomena as experienced in everyday life. Accordingly, the modelling activities are situated and subjective in nature, so that – unlike most conventional methods – EM presumes no fixed steps in the model-building process. Instead, modelling activities are guided by heuristic observation, interaction and experiment supported by the construction of an interactive computer-based model which metaphorically represents the modeller's construal that is always open to evolve in response to changes in the current situation and viewpoint.

EM research has found potential applications in many areas that include human-computer interaction, software development, computer-aided design, geometric modelling, behavioural modelling, concurrent engineering, educational technology, cognitive technology, artificial intelligence and business systems. In the context of these various applications, ideas about system development in general have been discussed, but only at the level of detail that is most relevant to the particular application area. This thesis seeks to consolidate and extend different ideas relating to system development that have been generated in previous research, and gathers them into a unified framework. It is the first thesis to focus on the EM approach to system development in the most general context.

1.2 Aims and motivations

The primary aim of this thesis is **to explore EM as a distinctive approach to system development**. The thesis is motivated by a large body of literature in philosophy, experimental science, cognitive science and computer science that stresses the importance of the experiential, experimental and informal nature of interaction between humans and the world that has become topical in relation to computer-based modelling in recent years. We briefly introduce some of the most influential ideas that motivate this thesis as follows:

- In his essays on radical empiricism [Jam96], first published about a century ago, William James sets out a philosophical position that accounts for knowledge in experiential terms. EM emphasises techniques for the acquisition and representation of knowledge that are experiential in nature, and that are associated with the process of model-building in system development.

- In [Goo00], Gooding describes the importance of using concrete artefacts to embody and assist the cognitive process of understanding phenomena in experimental science. EM supports the construction of computer-based artefacts for system development that have qualities similar to those of Gooding's artefacts.
- In [Tru00], Truex et al. describe the notion of 'amethodical' information system development, where the development process is not formalised. EM aims to support situated modelling for system development, but prescribes no general method for system development.
- In [Cro96], Crowe et al. advocate a constructivist approach to system development that emphasises "the mental and social nature of the construction of forms of information". EM aims to support cognitive processes in system development at both personal and interpersonal levels.
- In [Bro95], Brooks contends that *conceptual integrity* is the most important consideration in a system design. EM aims to provide principles and tools that help to maintain conceptual integrity in system development.
- In [Weg97], Wegner argues that interaction is more powerful than algorithms. EM encourages the interactive construction of models that can support human problem solving aspects of system development.
- In [Wes97], West advocates that *hermeneutic* computer science (which focuses on aspects of the informal natural world) as a complement to traditional formalist computer science (which focuses on aspects of formal deterministic worlds) in order to build complex systems. EM provides principles and tools that can be regarded as advances in hermeneutic computer science.

A key feature of EM (to be discussed in detail in chapter 2) is the construction of computer-based models that serve to embody the modeller's construal of phenomena in respect of agency, the observables that mediate this agency and the dependencies that link these observables. In developing a system, there are two complementary ingredients – the construction and configuration of system components, or of prototype models for these components, and the emergence in the minds of the developers of a system construal that reflects their commonsense explanations and expectations concerning how these components interact. These two ingredients of 'system construction' and 'system conception' which together make up 'system realisation' are both represented in EM. Model-building activity in EM is broader than system realisation in that it embraces the construction of experimental models whose purpose is to inform the modeller about the salient properties of the environment in which the system is to be established. Moreover, an EM model to represent a system component embodies the relevant observables and dependencies using a style of representation (namely a 'definitive script') that typically supports interactions beyond the scope of the actual component.

In view of the above observations, the EM approach to system development is not easy to compare directly with traditional system development methods. For this reason, in this thesis, the EM approach to system development will be discussed in association with two perspectives on systems that are identified in the thesis title, namely *before systems* and *beyond systems*. When adopting a 'before systems' perspective, we are considering the extensive preliminary activities that help to inform the system conception, and in particular with the way in which a system design with conceptual integrity may emerge from this activity. When adopting a 'beyond systems' perspective, we are concerned with exploiting the power of EM to develop computer-based models that can fulfill the functions of a system without the circumscription characteristic of traditional systems. This enables us to develop embryonic system-like configurations of components that can be customised and optimised to serve as a conventional system at the point of use. In this style of application of EM, which has been extensively illustrated in previous research (see e.g. [Run02]), the sharp distinction between a developer and a user is no longer appropriate. This theme is particularly relevant to modern ubiquitous computing environments, where modes of use are so dynamic that they cannot be captured in advance.

The primary aim can be subdivided into three more specific aims. We list these aims together with their motivations as follows.

- **Aim 1: to unify EM ideas for system development within a coherent framework.** This thesis seeks to unify ideas represented in specific applications of EM and abstract more general principles from them. The motivation behind this aim is that, over the past 20 years or so, the EM project enjoyed a steady growth of ideas and concepts about system development that are scattered in the EM literature. However, so far there has been no explicit attempt to give a coherent account of EM for system development in the most general context. This thesis aims to consolidate on and amplify EM principles for system development that can be applied to any application domain.

- **Aim 2: to expose the potentially comprehensive nature of EM as an approach to system development.** An approach to system development that is comprehensive in nature must be able to deal with its wide variety of applications and address its human aspects. With this in mind, this thesis studies the EM approach to system development in detail from several different angles. The picture of EM for system development that emerges depicts system development as a complex cognitive and social human activity that admits no formal prescription.

- **Aim 3: to evaluate the existing EM tools and explore new ways to build better tools.** This thesis aims to evaluate the existing EM tools in terms of how well they support the concepts of EM. Based on the issues identified from this evaluation, we aim to explore new ways to build better EM tools. The motivation behind this aim is that the development of EM tools has not kept up with the rapid development in research into EM concepts and principles. As a consequence, the existing tools cannot fully realise some important EM concepts.

1.3 Thesis outline

This thesis consists of nine chapters, structured as follows. Firstly, we give an overview of EM concepts (chapter 2). Secondly, we discuss this EM approach to system development in relation to traditional system development research. This includes a macro 'bird's-eye' view that gives an overall description of the EM perspective on system development (chapter 3), and a micro view that discusses the human problem solving that system development using EM entails (chapter 4). Thirdly, we discuss the themes of 'before' and 'beyond' systems. This includes the discussion of conceptual integrity as the essence of the system concept (chapter 5), and the discussion of ubiquitous computing with reference to the new perception of a 'system' that can be supported by EM (chapter 6). Finally, we turn to the investigation of EM tools to support the activities that underlie EM. This includes evaluations and prospects for EM tools (chapter 7), and description of a new visual EM tool (chapter 8). These two chapters describe tools for building systems and contribute to filling the gap between EM principles and implementations. The thesis concludes with a summary of ideas developed during the preparation of the thesis and proposes relevant further research directions (chapter 9). Figure 1.1 depicts an overview of the contents in each chapter.

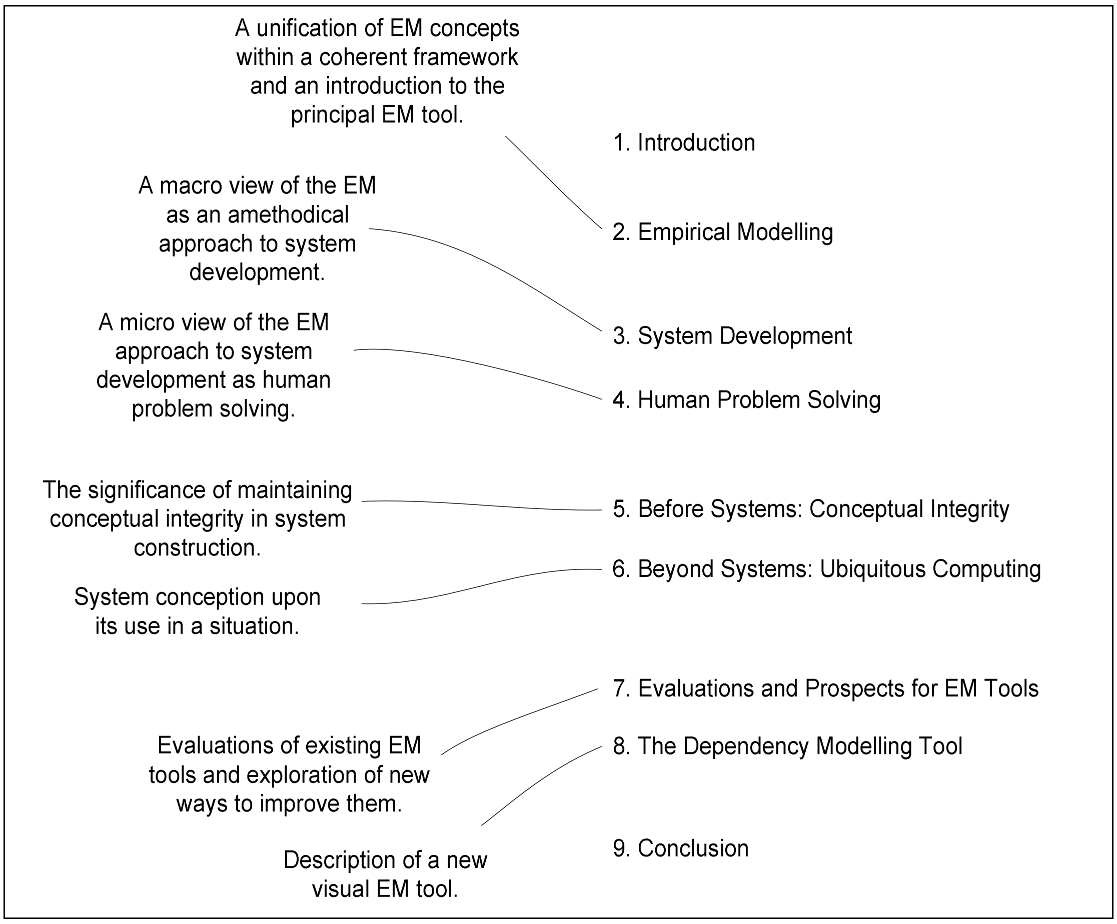

A unification of EM concepts within a coherent framework and an introduction to the principal EM tool.
1. Introduction
A macro view of the EM as an amethodical approach to system development.
2. Empirical Modelling
A micro view of the EM approach to system development as human problem solving.
3. System Development
4. Human Problem Solving
The significance of maintaining conceptual integrity in system construction.
5. Before Systems: Conceptual Integrity
6. Beyond Systems: Ubiquitous Computing
System conception upon its use in a situation.
7. Evaluations and Prospects for EM Tools
Evaluations of existing EM tools and exploration of new ways to improve them.
8. The Dependency Modelling Tool
Description of a new visual EM tool.
9. Conclusion

Figure 1.1: A summary of the thesis content

The contents of each chapter can be summarised in more detail as follows:

- **Chapter 2: Empirical Modelling.** In this chapter, we shall give the reader an overview of EM principles and tools. The content of this chapter is organised in such a way that it gradually moves from abstract concepts to concrete implementations. It starts with a discussion of the underlying philosophy of EM. Then, it introduces the modelling framework that is based on the underlying philosophy. Finally, we explain how the framework is realised in practice. This chapter should not be regarded solely as a literature review. Since the concepts of EM have themselves been evolving over the past 20 years or so, this chapter can be regarded as a consolidation of EM concepts introduced previously. In particular, we gather EM concepts into a unified and coherent framework (the Definitive Modelling Framework) that can be characterised by particular concepts of representation, and types of modelling activity associated with a distinctive interactive style of model building.

- **Chapter 3: System Development.** Promising applications for EM concepts have been identified in many research areas. The wide range of potential applications of EM gives some empirical evidence for the generality of EM concepts in respect of system development. However, there has been no attempt so far to account for this generality explicitly. In this chapter, we shall explore the generality of EM concepts and argue that the philosophical foundation of EM is domain-independent. The structure of this chapter is as follows. Firstly, it reviews some of the research strands in the literature on system development. These strands will be contrasted with the EM perspective on system development discussed in later sections. In addition, we shall specify the principal aims for the application of EM to system development. Secondly, our discussion will focus on thinking about intrinsic properties of systems and their relationship to an EM perspective on system development. Thirdly, we focus on thinking about system development activities and discuss how EM takes these activities into account. Finally, we conduct a case study of modelling a dishwasher system in both EM and another common modelling approach, object-oriented modelling using UML. Our aim is to explore the differences between these two styles of system modelling. This chapter concludes with a review of possible ways in which the aims of applying EM to system development set out earlier in the chapter can be achieved.

- **Chapter 4: Human Problem Solving.** This chapter focuses on the application of EM to support the human problem solving that is arguably the most important activity in system development. Firstly, we discuss the importance of developing principles and techniques for using the computer to support human problem solving. Secondly, we explore the relationship between problem solving and programming, with reference to some relevant research from the fields of general problem solving and psychology of programming. We shall argue that conventional programming paradigms cannot give comprehensive support for problem solving. Thirdly, we propose EM as a better approach for supporting problem solving. Finally, we present a case study based on a real life timetabling problem.

- **Chapter 5: Before Systems: Conceptual Integrity.** In his famous book, The Mythical Man-Month [Bro95], Brooks contends that “conceptual integrity is the most important consideration in system design”. This chapter endorses and elaborates this idea, and contends that obtaining conceptual integrity is essential before a coherent system concept can be formed. We believe that conceptual integrity emerges from activities that are prior to system identification or formalisation. Alongside this discussion, we shall explain how the EM approach to system development helps to address the issues of maintaining conceptual integrity. We start with a discussion of the meaning of conceptual integrity. The discussion leads to the identification of issues that are important to the maintenance of conceptual integrity. We shall discuss how EM can help to address them. At the end of the chapter, we shall also compare EM with other technologies that can be regarded as approaches to maintain conceptual integrity.

- **Chapter 6: Beyond Systems: Ubiquitous Computing.** In this chapter, we discuss how EM principles can potentially be applied to everyday practical computing as it may be in the future, where the system conception is only established upon its use in a situation. Following Weiser [Wei91] we adopt the term ‘Ubiquitous Computing’ or in short ‘ubicomp’ to refer to an era where people will use a variety of computer-based devices to support everyday activities. We firstly identify a variety of research related to ubicomp. We discuss and summarise their shared visions. We argue that these visions are hindered by the lack of a conceptual framework to encapsulate the complexity and new requirements of ubicomp. In particular, little research has so far been conducted to develop a conceptual framework that explicitly supports both the design and the use of ubicomp devices. We argue that having a coherent conceptual framework is very significant for maintaining the conceptual integrity of ubicomp systems, and that this will be fundamental to the success of ubicomp. We introduce a new conceptual framework namely SICOD (Soft Interfaces for the Control of Devices) based on EM principles and tools and illustrate this with examples. We shall discuss challenges involved in realising the framework. Finally, we shall describe some related research and make comparisons between them and our proposed framework.

- **Chapter 7: Evaluations and Prospects for EM tools.** In this chapter, we shall explore different techniques and tools that can be used to support the activities of EM. Our objectives are to evaluate existing and possible future implementations and to discuss the prospects for future development of EM tools. The chapter starts with a discussion of the essential characteristics of an ideal EM tool. This is to set the context for the evaluations described in later sections. The chapter then considers three existing technologies, Java, Excel and Forms/3, as possible EM tool implementations. Following this, we evaluate the principal EM tool: TkEden. Finally, we introduce two new EM tools (WING and EME) and highlight some of their prospective advantages based on researches described in previous sections.

- **Chapter 8: The Dependency Modelling Tool.** In this chapter, we shall describe a new EM tool, the Dependency Modelling Tool (DMT). The motivation for developing the DMT is similar to that of developing WING and EME – they aim to enhance the users' experience in the process of EM. However, the emphasis in DMT is on the ways to visualise various structures that commonly exist in EM models. We start out by identifying these structures and describe some tools developed by others in relation to visualising structures that are similar to the ones that exist in EM models. The development of DMT is partly inspired by some features of these tools. We then introduce DMT's user interface with a simple example. Followed by this, we describe how DMT helps to address two major concerns associated with an EM model: model comprehension and reuse. The final section highlights various issues relating to further research and development of the DMT.

- **Chapter 9: Conclusion.** This final chapter gives a summary of the research discussed in this thesis and discusses potential research that might be conducted in the future in relation to the work in this thesis.

1.4 Research contributions

This thesis explores EM as a distinctive approach to system development. The author's principal contribution has been to supply a comprehensive account of the essential ingredients of the EM approach to system development in the most general context. This thesis consolidates previous EM researches within a coherent and unified framework – the Definitive Modelling Framework (DMF). The DMF provides support for all the essential characteristics of EM. These include: the representation of our commonsense 'construals' of phenomena; situated model building activities that lead to the development and sharing of these construals; and an emphasis on open-ended experimental interaction.

The work in this thesis develops general principles for using EM to address the issues of system development with reference to two research strands, associated with ‘thinking about systems’ and ‘thinking about system development activities’. Where thinking about systems is concerned, the thesis identifies three intrinsic properties of systems (complexity, predictability and unity) and discussed each in association with EM principles. Where system development activities are concerned, the thesis identifies three important aspects of such activities that can be classified as cognitive, collaborative and methodological, and discusses how EM can address issues associated with all three aspects.

The thesis also contributes to linking EM to ideas of heuristic human problem solving that are arguably the most important aspect of system development. By giving illustrative examples, the thesis has given evidence that EM principles and tools offer essential support for the modeller to solve problems in an experimental trial-and-error fashion.

This thesis also makes an important contribution to analysing and exploring the notion of conceptual integrity and explains how EM can address the issue of obtaining conceptual integrity which is arguably the most important consideration in system conception.

In addition, this thesis contains the first discussion of EM in relation to ubiquitous computing, where the functional requirements of systems are so dynamic that they cannot be prescribed in advance. The thesis introduces the concept of a ‘soft interface’ as a means to support the configuration of ubiquitous devices as agents that constitutes end-user programming in this setting. A soft interface is essentially a simple EM model that facilitates the management and customisation of agents whose reliable behaviour has been identified.

Apart from these theoretical contributions, this thesis makes significant contributions to research on EM tools. In particular, it includes an evaluation of the principal EM tool (TkEden), and explores the possibilities for implementing EM tools using various existing technologies. The author has also developed three prototype tools to explore new possibilities for better tool implementations which are more suitable for novice users: the Windowing and Graphics tool (WING) – originally developed as an undergraduate project [Won98] and subsequently enhanced by the author in the preparation of this thesis, the Empirical Modelling Environment (EME) and the Dependency Modelling Tool (DMT).

During the preparation of this thesis, the author has contributed to three refereed publications [Bey00b, Bey01a, Bey01b]. The EM models built by the author as case studies in this thesis include: the Dishwasher model (chapter 3), the interface for the Temposcope (chapter 4), the Crossnumber model (chapter 4), the drink stock control model (chapter 6), the central heating control model (chapter 6), the Business Deal models (developed for the tool evaluations in chapter 7 by using TkEden, Excel, Java and Forms/3) and the ATM model (chapter 8).

2 Empirical Modelling

In this chapter, we shall give the reader an overview of EM principles and tools. The contents of this chapter are organised in such a way that it gradually moves from abstract concepts to concrete implementations. Section 1 describes the underlying philosophy of EM. Section 2 introduces the modelling framework that is based on the underlying philosophy. Section 3 explains how the framework is realised in practice.

This chapter should not be regarded solely as a literature review. Since the concepts of EM itself have been evolving over the past 20 years or so, this chapter can be regarded as a consolidation of concepts introduced in the EM project. In particular, we gather the EM concepts into a unified and coherent framework (the Definitive Modelling Framework) that has characteristic concepts for agent representation (section 2.2.1) that support modelling activities (section 2.2.2) based on a distinctive interactive style of model building (section 2.2.3).

2.1 A commonsense understanding of phenomena

The philosophical foundation of EM is based on the way we understand and interact with the world. In this section, we shall discuss the main concepts associated with it. These concepts are *not* new in the sense of having been newly invented. On the contrary, they originate from a commonsense way of understanding phenomena in the world that we experience in everyday life. EM privileges these commonplace concepts and provides a framework to embody them into computer-based artefacts – to help the cognitive process of understanding phenomena. The discussion in this section is based on Beynon's exposition of concepts of EM in [Bey02a, Bey02b].

In this context, the classical notion of concurrency is relevant only when issues associated with subjectivity, discrimination of essential entities and discretion over viewpoint can be resolved in an unambiguous way. This is the time when the observer has a full and precise understanding of the phenomena in question. Therefore, classical formal specification notations are suitable for representing the end-result of the understanding. In contrast, EM privileges powerful concepts (namely *observation*, *agency* and *dependency*) derived from the analysis of commonsense concurrency outlined above, and provides tools to assist the process of understanding before a possible end-result may eventually emerge.

We shall discuss the concepts of observation, agency and dependency respectively in the next three subsections.

2.1.2 Observation

Observation plays a key role in understanding phenomena. Observation is typically associated with our sensory experience of the environment through vision, hearing, touch, taste and smell. In EM, the concept of observation is extended to embrace determinants of state that can be directly apprehended by the observer whether or not they are sensory in nature. On this basis, observation can be purely abstract in that it involves only exploration of our own thoughts. In fact, observables are anything that can be given an *identity*. The identification of observables is a matter of the observer's interests. Therefore, *existence* of observables is subjective in nature.

From this subjective perspective, the observables that are present can be regarded as defining the observer's 'current state of mind'. In explaining this, Beynon [Bey02b] likens a state of mind to a physical location in the real-world, in which the observer can dwell, from which he can leave, and to which he can return. The current state of mind is associated with the observer's current focus of interest. A change in the current state of mind may be linked to a change in the observer's current focus of interest. A *state-change* can be the result either of an action on the part of the observer (either voluntary or involuntary and at the conscious or the subconscious level) or an action on the part of other agents in the situation.

When it comes to understanding complex phenomena with concurrency (e.g. understanding the complex behaviour of a concurrent system), the observer's state of mind can be overloaded in the sense that there are too many observables at different levels of abstraction that have to be considered at the same time. In this case, the conceptual integrity of the complex phenomena is difficult to maintain – so sense making becomes difficult. EM addresses this issue by seeking to embody and manage the state of mind of the observer using a special-purpose computer-based artefact.

2.1.3 Agency

Agency is a commonplace concept in our informal understanding of phenomena and interactions with the world. It is associated with *attributing state-changes* to what is understood to be their primary observed source. The informal nature of agency as perceived presents a challenge for conventional approaches that try to formalise agency or even factor out agency (as in a formal notation such as Concurrent Sequential Processes (CSP) that focuses on processes rather than agency as its primary abstraction in studying concurrency).

A natural question to ask at this point is: what entities in the world can be viewed as agents? This question cannot be answered without making reference to a particular context, and more importantly it can be answered only with reference to the state of mind of some observer. Agency is shaped by the subjective requirements, private interests and previous experience of the external observer. A change in the value of an observable can arise in a wide variety of ways.

In our informal interaction with the world, there is *no* single characterisation of the agency concept. In this respect, *commonsense agency* differs in nature from the two notions of agency widely accepted in the agent-oriented research community [Woo95]. In the *strong* notion of agency, an agent should be explicitly defined in terms of cognitive concepts such as beliefs, desires and intentions. In the *weak* notion of agency, an agent should be defined in terms of the observable properties that it exhibits, such as autonomy, reactivity, pro-activity and social ability. Both notions of agency try to formalise the agency concept and reduce it to a set of universal definable properties. They are arguably both inappropriate for the construal of agency in the context of commonsense concurrency.

It is difficult to discriminate between one entity as an agent and another. In fact, even a primitive observable can be an agent in a broad sense (e.g. “Ouch! This wooden spike is hurting me!”). Commonsense agency has more in common with what Lind calls the *very weak* notion of agency where indeed *anything* can be an agent [Lin00]. In the context of system design, Lind explains that:

"…the conceptual integrity that is achieved by viewing every intentional entity – be it as simple as it may – in the system as an agent leads to a much clearer system design and it circumvents the problem to decide whether a particular entity is an agent or not." [Lin00]

This quotation highlights the pragmatic advantage of adopting the very weak notion of agency. But the conceptual advantage of avoiding a prescriptive notion of agency is even more important – if we make a conscious effort to identify 'sources of state change' as we experience them in our everyday life, we find that they suggest three quite different views of agency:

- Primitive agency: agents as associations of observables. When considering the sources of state change, we commonly identify particular groups of observables as constituting a single agent. One significant characteristic that guides such grouping of observables is 'existence dependency' – sets of observables are observed to be present or absent at the same times, and so 'act as one'. Even if we have no direct experience of how such association of observables can effect state change, their presence or absence alone typically influences the perceptions of the observer. This leads us to regard such associations as agents in so far as they can potentially be the cause, cue or trigger for some action on the part of another agent. By this criterion, every observable is an agent and so is the external observer.

- Explicit agency: agents as the perceived sources of open-ended state change. A primitive agent can be viewed as an explicit agent when empirical evidence leads us to attribute state changes to it. Identifying explicit agency is typically associated with growing familiarity with a particular context for observation on the part of the observer, and with a degree of specialisation of observation towards a specific focus of interest or goal. The attribution of state change does not normally entail assumptions concerning the circumstances in which it occurs or what stimulus might be responsible. Such issues will typically be the subject of ongoing experiment.

- Circumscribed agency: agents as actors with circumscribed behaviour. In some circumstances, the attribution of agency and the context for interaction can be so precisely identified or prescribed by the observer that the agent can be interpreted as acting like the actors in a play or the components of a system. Identifying circumscribed agency is generally

> associated with a very high degree of specialisation of observational context that involves strong presumptions and discretionary constraints concerning acceptable interaction on the part of the observer. Like real world actors in a play, circumscribed agents are in significant respects no longer autonomous in the way that explicit agents can be. They are ‘virtual’ agents in the closed-world, where the context of observation is so circumscribed that (as far as the observer’s specific viewpoint is concerned) nothing new can be learnt from further study of the phenomenon that is the subject of interest.

As we understand phenomena in the world, our view of agency has a tendency to progress from the primitive through the explicit to the circumscribed. EM provides a framework to accommodate all three views and is particularly concerned with explicit agency, which is somewhere between the primitive view, where the agency is almost vacuous, and the circumscriptive view where it is very tightly constrained.

2.1.4 Dependency

Dependencies are patterns of stimulus-response between observables with some degree of persistency. Identifying dependencies among observables also plays a key role in understanding phenomena. To identify dependencies, the observer needs to have a perception of state-change and identity (e.g. to see what moves), combined with the ability to remember and so to have an *expectation* (e.g. to predict what will move), and to correlate the action with the state-change (e.g. to know what causes the move). Typically, such identification is associated with a prolonged period of observation and/or interaction with the phenomena that may be either deliberate or accidental. Activities that lead to the identification of dependencies by the observer of a phenomenon can be categorised as follows:

- Pure observation – The observer either has no way to, or intentionally does not, influence the phenomena. For example, cosmology is entirely founded on pure observation of a system over which we apparently can exercise no direct influence.

- Direct interaction – In some cases, direct and directed intervention is possible. The observer can interact with the phenomena in the role of experimenter.

- Off-line experimentation – The observer may be able to carry out ‘off-line’ experiments to test particular hypotheses about stimulus-response patterns without directly interacting with the phenomena.

In all cases, dependencies originate from past experience and are confirmed by the present observation and/or interaction. The purpose of identifying dependencies is to inform future expectations of events associated with the phenomena. In all three activities, the capacity of the observer to be surprised by responses is particularly important. Surprises typically reveal subtleties of phenomena that require further investigation and may eventually lead to the discovery of new observables, agents and dependencies. In this respect, being able to recognise explicit agency is significant.

2.2 The Definitive Modelling Framework (DMF)

In this section, we introduce the Definitive Modelling Framework (DMF). The DMF is a framework for building artefacts that embodies understanding of phenomena in terms of commonsense notions of concurrency, observation, agency and dependency. It is a ‘modelling’ framework because it is primarily associated with building artefacts; it is ‘definitive’ because the modelling process involves exploring *definitions* that form an important part of an artefact. The term ‘Definitive Modelling Framework’ rather than ‘Empirical Modelling Framework’ is adopted on the basis that there are arguably modelling activities that involve embodying observation, agency and dependency in artefacts that do not explicitly use the computer-based representations to be described below. The main aim of the DMF is to assist the cognitive process of understanding phenomena. The introduction of the DMF serves to consolidate EM concepts developed in the past 20 years or so that include the Abstract Definitive Machine (ADM) [Sla90], an EM environment for concurrent engineering [Adz94a] and Interactive Situation Models (ISMs) [Sun99a].

2.2.1 The representation of observables, dependencies and agents

In the DMF, observables and dependencies can be represented as *definitions*. A typical definition takes the form:

$$y \textit{ is } f(x_1,x_2, \dots , x_n)$$

where $y, x_1, x_2, \dots , x_n$ are *definitive variables* (or *variables* for short), *is* is a separator and *f* is a *formula*. A definition is divided into two parts separated by the *is* separator, viz. its left-hand-side (LHS) and its right-hand-side (RHS). The LHS of a definition typically contains only one variable (*y*) whose value is determined by the formula on the RHS. The formula contains algebraic operators and functions that reference a set of variables (and possibly some constants) as operands and is used to calculate the current value of the LHS variable (*y*). The purpose of the formula is similar to that of the formulae in the cells of a spreadsheet.

A definition is a representation of an *indivisible stimulus-response pattern* (or a dependency) between observables. Observables are represented by variables. The current status of observables is represented by the current values of variables. The maintenance of the stimulus-response pattern is specified by the formula – 'stimulus' is associated with any change of value in the variables referred at the RHS; 'response' is associated with an update to the value of the variable at the LHS. The term 'indivisible' is used to convey the idea that the maintenance of the relationship established by the stimulus-response pattern permits no external interruption.

In the DMF, a set of definitions provides the fundamental representation of state. The primary role for such a set of definitions is similar to that of the set of definitions underlying a spreadsheet: it supplies values and dependencies that correspond closely to the observables and dependencies in the situation it represents. A set of definitions can also be used to model behaviours. For this purpose, subsets of definitions are grouped into *entities* that correspond to primitive agents in the environment. The current state of the environment is determined by the primitive agents that are present typically together with generic persistent observables such as time and gravity; in the DMF, a set of definitions that represents state accordingly comprises groups of definitions within an entity that come and go together (in view of their existence dependency), together with isolated definitions that are typically persistent.

With this model of state, the effect of all kinds of agent actions can be expressed in the DMF. A typical action takes the form:

$$g(x_1, x_2, \dots, x_n) \rightarrow (\ \textit{variable redefinition} \mid \textit{entity creation} \mid \textit{entity deletion}\ )^*$$

where $x_1, x_2, \dots, x_n$ are variables, *g* is a *guard*, and the action consists of a sequence of operations to *redefine variables*, and *create* or *delete* of *entities*. The guard is a logical expression that references a set of variables (and possibly some constants) as operands. An action is a representation of a latent interaction or experimental action associated with some agent. When the guard evaluates to true, the agent who owns this action has the privilege to perform the sequence of redefinitions, creations and deletions specified at the right-hand-side of the action. Note that the fact that the agent has the privilege to act does not necessarily mean that the agent has to act.

Definitions and actions equip the DMF with the mechanisms needed to support our commonsense construal of phenomena. Their expressive power stems from the intelligent engagement of the human observer in their interpretation and execution. The appropriate way to represent an explicit agent in the DMF is by an entity that comprises a group of definitions together with an associated group of latent actions. By default, all action is conceived as performed by the human observer. Only the human observer can make due allowance for provisional and imperfect knowledge of the conditions governing action. Only the human observer can play the part of a super-agent, contributing actions that are not preconceived such as representing experimental interactions and random events.

When using the DMF for modelling and simulation, it is not practical or appropriate for the observer to be responsible for all actions. Some actions are typically executed automatically as in a machine. The extent to which automatic execution is meaningful depends upon the extent to which the activities of explicit agents can be circumscribed. In general construal and simulation of concurrent behaviour within the DMF, both human and machine execution are represented. What is more, whether such behaviour is to be understood as design, use or testing activity is a matter of interpretation.

Figure 2.1 depicts the *dual perspective on agency* in the DMF. From the *human perspective*, an agent can be viewed in terms of the observables it owns, identified dependencies among the observables (i.e. indivisible stimulus-response patterns), and latent interactions and experiments. From the *machine perspective*, an agent is represented by an entity that consists of a group of definitions, possibly together with a group of guarded actions.

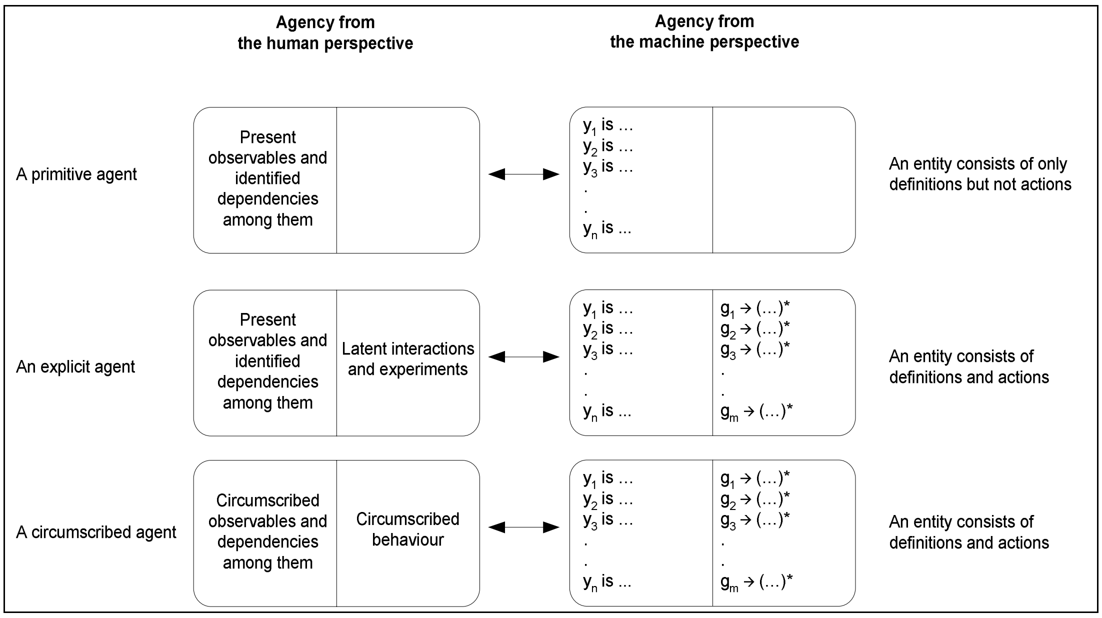

Agency from the human perspective
Agency from the machine perspective
A primitive agent
Present observables and identified dependencies among them
y_1 is ...
y_2 is ...
y_3 is ...
y_n is ...
An entity consists of only definitions but not actions
An explicit agent
Present observables and identified dependencies among them
Latent interactions and experiments
$g_1 \rightarrow (\ldots)^*$
$g_2 \rightarrow (\ldots)^*$
$g_3 \rightarrow (\ldots)^*$
$g_m \rightarrow (\ldots)^*$
An entity consists of definitions and actions
A circumscribed agent
Circumscribed observables and dependencies among them
Circumscribed behaviour
An entity consists of definitions and actions

Figure 2.1: The dual perspective on agency

In the DMF, any agent can be described from *both* the human and the machine perspective whenever they are appropriate. It is important for the external observer to keep both perspectives in mind when analysing agencies. When we consider the absolute discretion that is involved in human execution of actions, the significance of the definitions and actions attached to an agent in the machine perspective is called into question. Human execution can discount these definitions and actions entirely. Viewed in this light, the purpose of representing agency from the machine perspective is to provide a representation of what is already known but provisional; in contrast, the human perspective encourages exploration of what is still unknown. On this basis, the duality facilitates the representation of provisional knowledge about the context in which agent actions are performed and the acquisition of knowledge of a similar nature that is still unknown.

The motivation for the dual perspective is similar to that explained by McCarthy in his discussion of “ascribing mental qualities to machines” [Mcc79]. McCarthy points out that it is sometimes useful to ascribe certain mental qualities like beliefs, intentions and wants to a machine. In the context of understanding the behaviour of a program, he describes the reason for ascribing mental qualities as epistemological:

> *“… ascribing beliefs is needed to adapt to limitations on our ability to acquire knowledge, use it for prediction, and establish generalizations in terms of the elementary structure of the program.”*

[Mcc79]

Our concern is to understand phenomena in general rather than to understand the behaviour of specific programs. However, in the DMF, the rationale for being able to treat any agent as human-like is similar to McCarthy's reason for ascribing mental qualities to machines.

Figure 2.2 shows a representation of multiple agents identified in the process of understanding a phenomenon. We can see that there is a circumscribed agent whose behaviour has been well understood; there is an explicit agent that has yet to be – and may never be – circumscribed; there are primitive agents associated with sets of variables with integrity, and with definitions outside the boundaries of all agents. In the picture, a solid outline around an agent means that the boundary of the agent is fixed; a dotted outline means that the boundary of the agent is subject to further revision. This convention will be used to depict the various kinds of agency throughout the discussion of the DMF in this chapter.

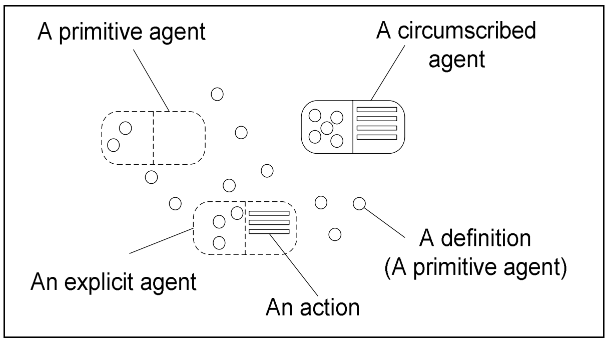


Figure 2.2: A representation of multiple agents

The significant features of the representation of multiple agents within the DMF can be summarised as follows:

- Concurrency – There are two main types of concurrency. Firstly, different agents can perform actions concurrently. Secondly, the dependencies associated with definitions are maintained concurrently.

- Indivisible stimulus-response patterns – Definitions represent dependencies among observables whose evaluations permit no external interruption.

- Dual perspective on agency – Any agent can be viewed as a human-like agent.

- Openness of privileges – The representation imposes *no* constraint on how the observables/variables owned by an agent are referenced by others.

In the next subsection, we shall discuss how modelling activities of EM can be performed using the DMF.

2.2.2 The modelling activities

The representation of multiple agents described above serves as an artefact that embodies our understanding of the phenomenon. To understand the modelling activities under the DMF, we need to conceptually distinguish between four entities: the *modeller*, the *phenomenon*, the *referent* and the *artefact*. Figure 2.3 depicts the relationships between them. The artefact embodies general experience of the phenomenon. The referent is the aspect of the phenomenon that is of particular interest to the modeller, and in general evolves to reflect the specialised ways of interacting with and interpreting interaction with the artefact that develop in the modelling process. The modeller as the external observer is the archetypal agent who makes the fundamental and essential semantic link between the referent and the understanding of it (as embodied in the artefact he builds). Only the external observer has the capacity to be surprised by responses from the referent or the artefact. Surprise can lead to a shift in perspective that prompts a point of departure from what has been previously explored, and leads to a deeper understanding of the referent and an enrichment of the artefact. We shall discuss the importance of using an artefact, typically computer-based, to represent our understanding in more detail in the next subsection.

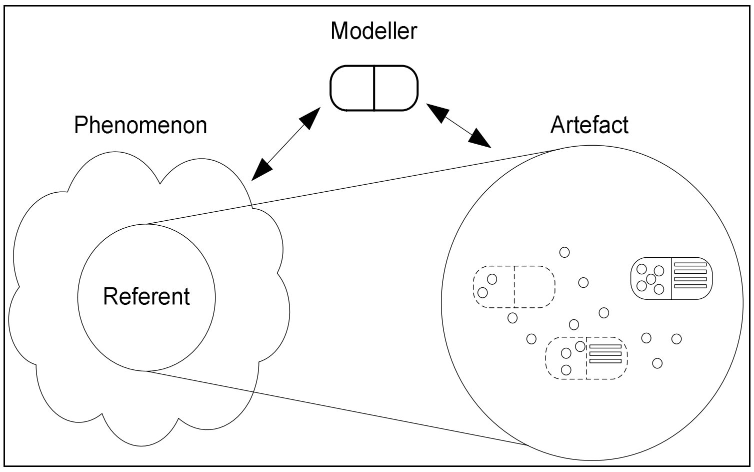


Figure 2.3: Relationships between the modeller, the phenomenon, the referent and the artefact

There are two important and interrelated modelling activities namely *agentification* and *role-playing*. Agentification involves identifying agents in the referent and analysing the observables to which they may be construed to respond. Figure 2.4 depicts the typical pattern of progress in the agentification activity over time. Initially, only a few observables and dependencies have been identified by the modeller. The view of agents is at its most primitive – all observables can be interpreted as primitive agents. As the modeller obtains more understanding about the referent, some explicit agents are identified as being responsible for state-changes to some observables. An element of shift in observational perspective on the part of the modeller is also involved here. Gradually, deeper understanding shapes some circumscribed agents whose behaviour is so predictable that they cannot surprise the modeller at all. This is a creative and experimental process. There is *no* universal generic procedure to follow through which successful representation or understanding is guaranteed. Success depends upon being able to correlate state-based observations in order to identify stimulus-response patterns by interacting and experimenting with the artefact as well as the referent.

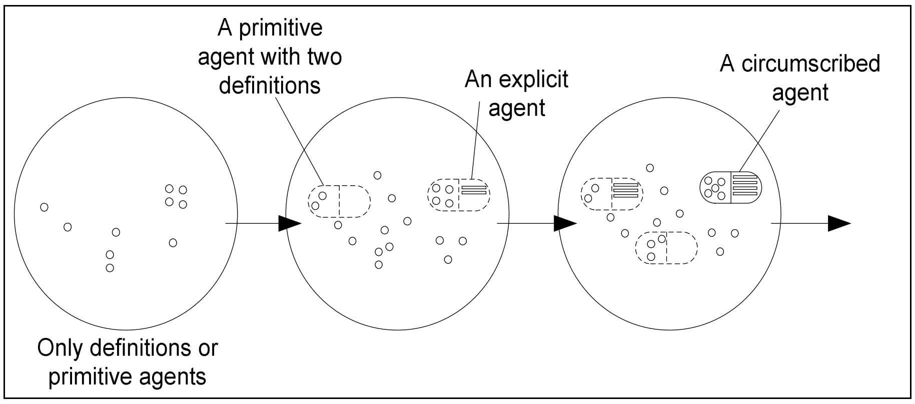

A primitive agent with two definitions
An explicit agent
A circumscribed agent
Only definitions or primitive agents

Figure 2.4: Agentification as progression from primitive to explicit and circumscribed views of agency

The concept of role-playing activity is a feature of EM that has been discussed by several previous researchers (cf. [Sun99a, Run02]). This involves a conceptual distinction between different roles that the modeller can play. The modeller can be considered as a *super-agent* who can play the roles of the *external observer*, *acto*r and *director* described as follows:

- The modeller as the external observer – The modeller interprets what is going on in the artefact with reference to his observation and perception about the phenomenon. There are many factors that can affect the modeller's interpretation which include his past experience, future expectation, knowledge of current interactions, subjective judgement, and even his physical ability. Observations and their interpretation contribute to a deeper understanding of behaviour of both the artefact and the referent.

- The modeller as an actor – The modeller construes a situation from the perspective of an internal agent who is directly interacting with other agents. When the modeller acts in the role of an actor, she is not only able to see things from the perspective of the actor, but is also in a position to reflect upon the actor's role in its overall context.

- The modeller as a director – As a director, the modeller directs the internal agents in the execution of their actions. This activity is similar to directing a play or simulating the execution of a concurrent program. In this case, the modeller can experiment by executing, modifying and interrupting existing actions, or by introducing new actions.

Figure 2.3 depicts the relationships between the modeller, the phenomenon, the referent and the artefact in an abstract and simplistic way. The figure is helpful to the reader in understanding their significance in the modelling activities, but does not always convey the full nature of the phenomena and the referents that arise in practice. When we consider the modelling context for system development in more detail, we can distinguish three scenarios (see Figure 2.5):

- Scenario 1: the artefact represents a fixed referent – In this scenario, the modeller constructs the artefact by observing a specific aspect of the phenomenon as its given fixed referent. Interactions with the phenomenon are primarily aimed at obtaining more understanding of the referent rather than changing it. The key activities involved in this scenario are *analysis* and *simulation* of an existing phenomenon that is at some level well-understood, as for example, in creating a football game simulation [Tur00]).

- Scenario 2: the artefact and its referent co-evolve – The aspects of the phenomenon that are of interest change as the artefact is developed. Thus, changing the artefact affects the referent. The modeller is not only observing the phenomenon but also shaping the referent. The key activity involved is the *design* of the artefact that contributes to the discovery of new aspects of the phenomenon (cf. [Loo98]), as for example, in the design of a digital watch [Roe01].

- Scenario 3: the phenomenon comprises model-building activities – The modelling involves treating modelling activities performed by some other modellers from a higher viewpoint. The meta-modeller can be thought of as a coordinator whose purpose is to *negotiate meanings* with other modellers within the phenomenon. The referent (not depicted in Figure 2.5) is associated with those aspects of the interaction between other modellers that concern her in the role of coordinator. These might include issues such as requirements and deadlines imposed upon modellers, work patterns for their collaboration and key characteristics of their artefacts.

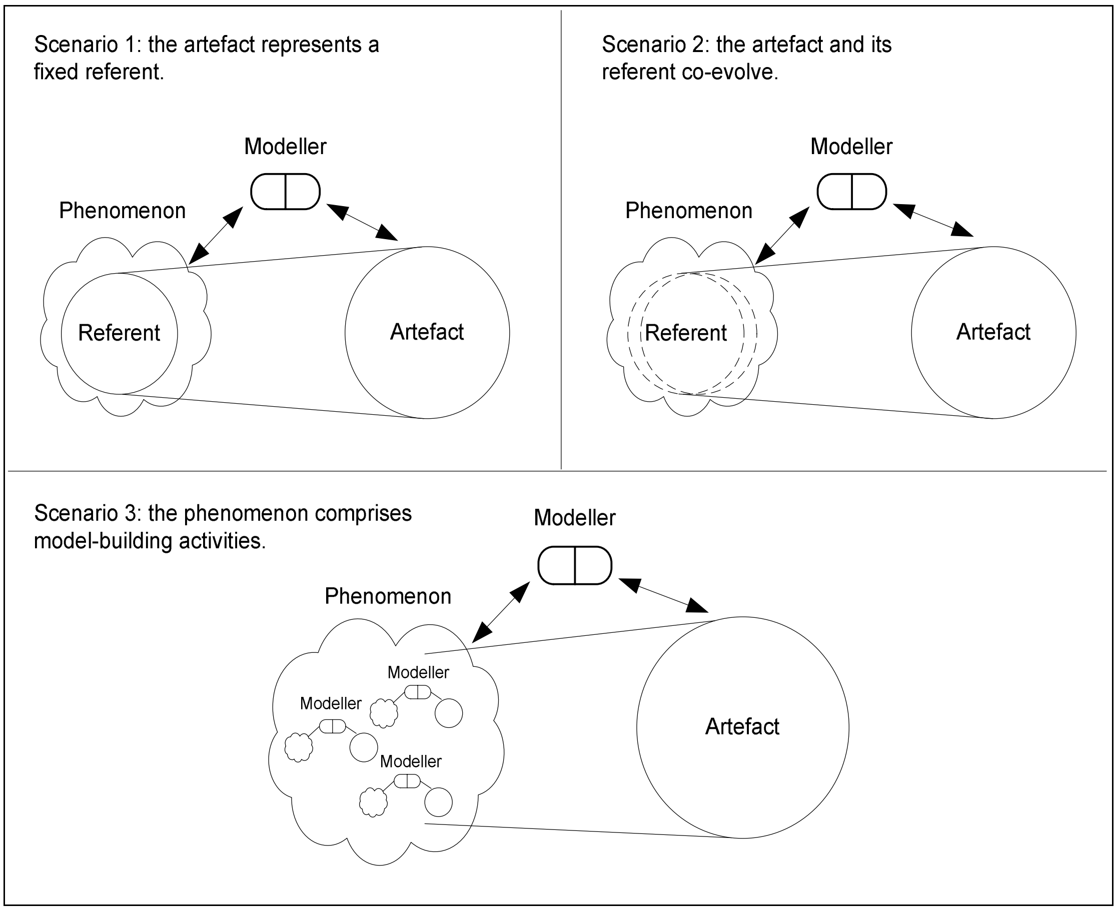

Scenario 1: the artefact represents a fixed referent.
Modeller
Phenomenon
Referent
Artefact
Scenario 2: the artefact and its referent co-evolve.
Modeller
Phenomenon
Referent
Artefact
Scenario 3: the phenomenon comprises model-building activities.
Modeller
Phenomenon
Modeller
Modeller
Modeller
Artefact

Figure 2.5: Scenarios of relationships between the phenomenon, modeller and artefact

Scenario 3 conveys the idea of *collaborative definitive modelling* in the DMF that requires some further elaboration. In the context of applying EM to concurrent engineering, Adzhiev et al. [Adz94a] introduce the concept of a hierarchy of designer agents who collaborate with each other in designing engineering products. This concept can be generalised or reinterpreted to apply to all kinds of definitive modelling that involve a team of human-like agents, i.e. a team of modellers. Collaborative definitive modelling involves developing interpretations and negotiating and sharing understanding of a referent among a team of modellers. The current status of the *consensus* can be understood with reference to a hierarchy of modellers, as shown in Figure 2.6 below.

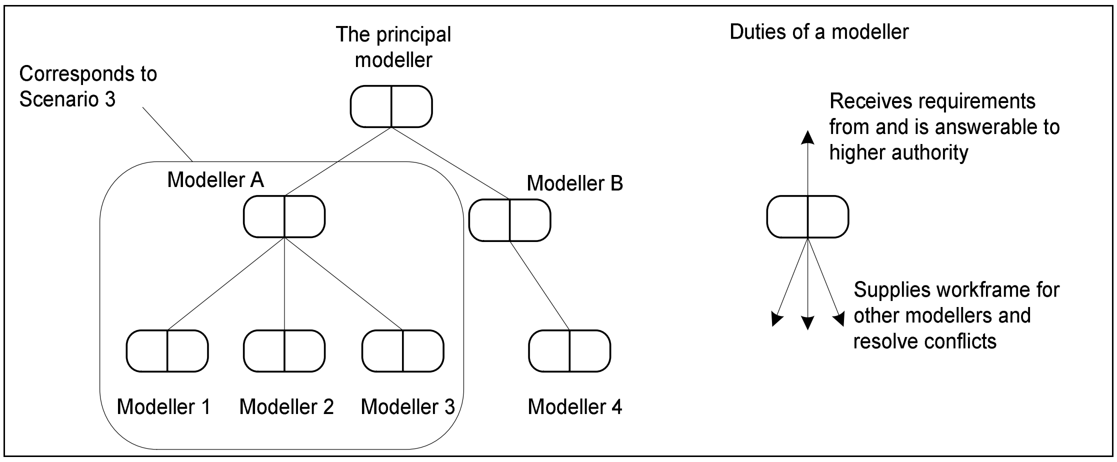

The principal modeller
Duties of a modeller
Corresponds to Scenario 3
Receives requirements from and is answerable to higher authority
Modeller A
Modeller B
Supplies workframe for other modellers and resolve conflicts
Modeller 1
Modeller 2
Modeller 3
Modeller 4

Figure 2.6: A hierarchy of modellers in collaborative definitive modelling

The hierarchy of modellers is actually a generalisation of Scenario 3 in Figure 2.5. Each modeller internal to the hierarchy has two roles. As a coordinator, a modeller supplies a frame for the work of other modellers and resolves conflicts. As a worker, a modeller receives requirements from and is answerable to a higher authority, that is, act as a modeller at a higher level of the hierarchy. At the upper-levels of the hierarchy, the modelling activity has a managerial aspect that involves framing the design objectives and patterns of interaction between the modellers at the next lower levels. At the lowest level of the hierarchy, the modelling activity focuses more on working on specialised experiment and understanding that impacts on specific aspects of the whole consensus.

The consensus in collaborative definitive modelling is evolving all the time according to emerging understanding of the referent. Each modeller builds an artefact to help his understanding of certain aspects of the phenomenon that contributes to the whole consensus. This interaction of minds is in the same spirit as Minsky's agent hierarchy in his Society of Mind [Min88]. The artefact also serves as a medium for sharing knowledge. This hierarchical framework for collaborative definitive modelling expresses the potential conceptual integrity of the intermediate understanding of the referent, which has a potential for consensus that encompasses every modeller in the team. This issue will be further discussed in chapter 5.

We summarise the significant features of the modelling activities in the DMF as follows:

- State-based observation – It is important for the modeller to be able to directly observe and experience states of the referent and the artefact so that he can correlate them and identify patterns among them.

- Evolutionary construal – Agentification is an evolutionary learning process typically starting from identifying primitive agency to shaping explicit agency and eventually but not necessarily proceeding to specifying circumscribed agency.

- Role-playing – The modeller can play several different roles (external observer, actor, director and coordinator) to effect the shifts of perspective necessary to acquire deeper understanding.

- Subjectivity – Modelling as a personal affair is represented in the DMF. Subjective interpretations and interactions are central to the concerns of understanding phenomena.

- Collaboration – Modelling as a collaborative affair is represented in the DMF. The DMF has a potential to facilitate the collaborative creation and understanding of complex phenomena that involves a team of modellers (see chapter 3 and 5 for examples).

2.2.3 Interactive Situation Models (ISMs)

This subsection discusses the importance of the *interactive* nature of the artefact in the DMF. From the discussion in the previous subsection, we know that the artefact in the DMF plays several important roles:

- in representing the modeller's personal understanding (in terms of definitions, actions and agents);
- in the activities involved in acquiring new understanding (agentification and role-playing);
- in the animation of behaviour (through potentially automatable dependency

maintenance and action triggering);

- in communication between modellers (collaborative definitive modelling).

These roles indicate that the artefact in the DMF is more than an abstract representation such as a formal specification might provide. The artefact actually facilitates the whole cognitive process of understanding and creating phenomena. To construct an artefact that can meet the demands of all the above roles, we need to use the computer. Indeed, EM is particularly concerned with *using the computer as an instrument* [Bey01a] that mediates between a phenomenon as conceived by the modeller and the phenomenon as perceived.

To emphasise that we are in fact concerned with a very special kind of artefact, we refer to the artefact built by using the DMF as an *Interactive Situation Model* (ISM), a term introduced by Sun [Sun99a]. An ISM is a computer-based artefact constructed through *situated* modelling activities. The use of term 'situated' reflects the significance of both the social and cultural context (cf. [Suc87]), and the specific physical environment in which the modelling is conceived [Bey98]. Unlike most computer models, which have a fixed interface and preconceived use, an ISM is 'interactive' in the sense that it is open to elaboration and unconstrained exploratory interaction. In something like the way that clay is suitable for modelling physical shapes, an ISM is suitable for modelling our conceptual understandings. By embodying patterns that reflect commonsense concepts of concurrency, observation, agency and dependency in an ISM, the modeller aims to establish an intimate relationship between the ISM and his or her evolving knowledge about the referent.

Building ISMs is closely related to the way in which experimental scientists use artefacts as a means of devising *construals* through observation and experiment. The term 'construal' was used by Gooding to refer to the concrete artefacts that embody insight into experimental interactions such as are described in [Goo90]. In particular, Gooding describes how Faraday, in developing his understanding of electromagnetic phenomena, constructed artefacts to represent observables such as electrical currents and magnetic fields, and dependencies such as the relationships between the polarity of a magnetic field and the direction of current. The importance of physical artefacts in supporting understanding is also endorsed by Feynman's view of what it means to be a physicist:

"a physical understanding is a completely unmathematical, imprecise, and inexact thing, but absolutely necessary for a physicist" [Fey64].

In its essence, an ISM resembles what Gooding characterises as a construal. There is ambiguity concerning the extent to which a construal should be identified with a physical object or with a mental model. As Gooding observes in [Goo01] this ambiguity presumes a philosophical stance that is at odds with the traditional dualist separation between the mind and the physical world.

Throughout this thesis, we shall make no distinction between the terms ‘ISM’, ‘EM model’ and ‘artefact’. They are synonyms that describe a model built by using the DMF but emphasise different aspects of the nature of the model.

2.3 EM in practice

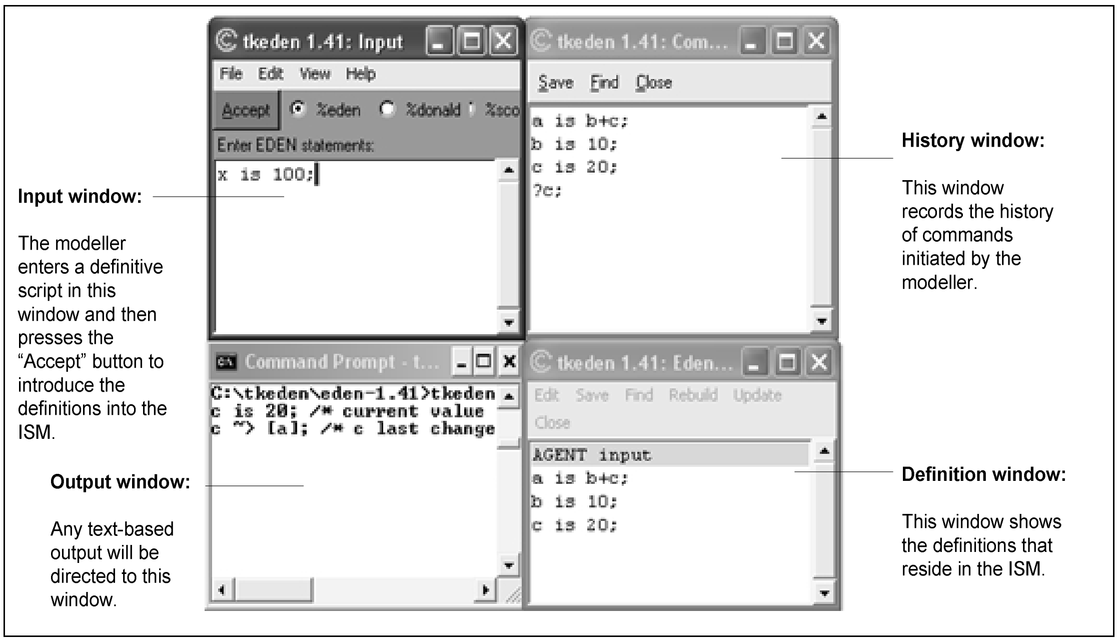
This section describes two important techniques developed to support EM in practice. Firstly, we shall introduce an implementation of the DMF that is commonly used in EM, namely ‘modelling with definitive scripts’. Secondly, we shall introduce a technique that has the potential to document the understanding gained from modelling activities, namely the LSD notation.

2.3.1 Definitive scripts

The main EM tool currently used for supporting modelling activities under the DMF is *TkEden*. As explained in more detail later in this subsection, to construct an ISM, the modeller can specify definitions by using some *definitive notations* to form a script of definitions, or *definitive script*. TkEden interprets the definitive script that is input by the modeller and introduces the definitions into the ISM. TkEden also *automatically* maintains the dependencies among definitions by keeping the value of every definitive variable up-to-date. This subsection only gives the brief explanation of constructing an ISM using definitive scripts that is needed for the reader to understand the illustrative models introduced throughout this thesis. For an extended treatise on modelling with definitive scripts, the reader can refer to [Run02].

The core of the TkEden interpreter is a simple interpreter, called TtyEden, that was originally developed by Y. W. Yung in his final year undergraduate project in 1987 [Yun90]. TtyEden has a text-based command line interface that supports interactive model construction in a single definitive notation called Eden (a general purpose language for definitive modelling). The tool was later extended by Y. P. Yung in his study of the ‘definitive programming paradigm’ [Yun93]. In addition to Eden, Y. P. Yung integrated two other definitive notations into the tool: Donald (for 2D line-drawing) and Scout (for screen display). Since the tool makes use of the Tcl/Tk library for implementing windowing and graphical drawing, it is called TkEden. Subsequently, TkEden has been maintained and enhanced by several people in the EM research group, most recently and extensively by Ashley Ward who has developed and maintained an open-source distribution [Ope02]. Enhancements include implementing a prototype distributed version of TkEden (namely DtkEden developed by Sun [Sun99a]) and introducing other definitive notations (e.g. Sasami for 3D graphics and Eddi for database modelling [EMWeb]).

Figure 2.7 shows a screenshot of the TkEden interface. It displays four windows, namely the ‘input window’, the ‘output window’, the ‘history window’ and a ‘definition window’. The ‘input window’ is where the modeller can redefine, delete and create definitions by writing definitive scripts. The ‘history window’ records the sequence of commands that have been input by the modeller. The ‘definition window’ shows some of the definitions that reside in the ISM: other windows to display complementary definitions can be accessed via the View option in the ‘input window’.



Input window:

The modeller enters a definitive script in this window and then presses the “Accept” button to introduce the definitions into the ISM.

```
x is 100;
```

History window:

This window records the history of commands initiated by the modeller.

```
a is b+c;
b is 10;
c is 20;
?c;
```

Output window:

Any text-based output will be directed to this window.

```
C:\tkeden\eden-1.41>tkeden
c is 20; /* current value
c ~> [a]; /* c last change
```

Definition window:

This window shows the definitions that reside in the ISM.

```
AGENT input
a is b+c;
b is 10;
c is 20;
```

Figure 2.7: A screenshot of the TkEden interface

A definitive script may contain definitions in several different definitive notations. The term ‘definitive notation’ was first introduced by Beynon [Bey85] to refer to a simple language that can be used to specify definitions in formulating a definitive script. The basic semantics of a definitive notation is determined by an underlying algebra of data types and operators but the syntax of definitions can take a variety of forms. For example, a definition to establish the following dependency:

"the value of the variable A is the sum of the values of the variables B and C"

can be formulated in TkEden in three different ways:

- `A is B + C;`
- `A = B + C`
- `A = B + C;`

A definitive notation can be specifically designed for a particular application domain to facilitate ease of use by using domain-specific terms and syntactic constructs. For the most part, the models illustrated in this thesis are constructed by using Eden, Donald and Scout in TkEden. The three illustrative definitions above are in Eden, Donald and Scout respectively. Some similarity with these notations is helpful in understanding the models introduced later in this thesis.

Eden

Eden (the Evaluator for DEfinitive Notations) was first designed and implemented by Y. W. Yung [Yun88]. The Eden notation is a general purpose language that implements the DMF concepts of definitions and actions. The Eden syntax and data types are similar to those in the C language. The basic programming constructs include `for`, `while` and `if`. The basic data types include `integer`, `float`, `string` and `list`. In fact, Eden is a 'hybrid' programming language that allows both definitive modelling and procedural programming to be performed in the same environment. The modeller can even specify their own *functions* that can be used in specifying definitions. This makes Eden extremely expressive but, to make most effective use of the tool, the modeller has to be conscious of the need to be as faithful as possible to the principles of EM in the DMF.

Significant features of Eden are described with illustrative examples as follows:

- syntax for definitions – the sample definitions shown below specify the dependencies between three variables. Note that each definition has its LHS and RHS separated by the `is` separator, and each definition is terminated by a semi-colon '`;`'. There is no need to declare a variable before its use. The type of a variable is determined by the interpreter automatically. The dependencies between variables specified by the definitions are also maintained automatically. In this case, if the value of `b` or `c` has changed, the value of `a` will be updated according to its formula.

```
a is b+c;
b is 10;
c is 30;
```

- difference between a variable assignment and a definition – the two statements below are semantically different. The first statement specifies a definition but the second statement specifies an assignment. The first statement specifies that the value of `a` is *always* dependent on the sum of values of `b` and `c` so that `a` gets updated every time the value of `b` or `c` has changed). The second statement specifies that, unless and until it is reassigned, the value of `a` is equal to the current value of the sum of `b` and `c` at the point of assignment. In this case, the calculation is one-off, so that subsequent changes of `b` and `c` do not have any effect on the value of `a`. We refer to `a` as a definitive variable in the first statement and a procedural variable in the second.

```
a is b+c;
a = b+c;
```

- specifying a function – an example of the definition and use of a function is given below. This function determines the greater of two numbers. After introducing this function into the model, the modeller can use it anywhere on the RHS of a definition, as, for example, in the definition: "`a is greater(b, c);`".

```
func greater {
    para x, y;
    if (x>y) result = x;
    else result = y;
    return result;
}
```

- specifying an action – an example of an action is given below. An action in Eden is a 'triggered procedure'. In this case, the action `testaction` redefines `a` whenever the variable `m` changes. The current definition of `a` depends on the value of `m`.

```
proc testaction: m {
    if (m>0) a is b+c;
    else a is b*c;
}
```

- basic means for querying a definition – examples of two different types of query are given below. The first statement asks for the current definition of variable `x`. The second statement returns the current value of `x`. In both cases, the output is directed to the ‘output window’.

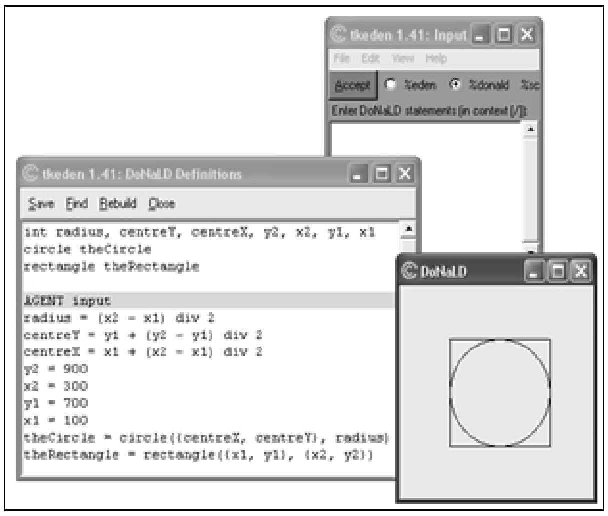
```
?x;
writeln(x);
```

Donald

Donald is a definitive notation for 2D line-drawing. Its design was first conceived by David Angier in his final year undergraduate project, and later extended by Beynon et al. [Bey86b]. Donald supports a variety of data types including `integer`, `real`, `char`, `point`, `line`, `shape`, `arc`, `circle`, `ellipse`, `rectangle` and `label`. Drawings are grouped into `viewports`. Each `viewport` is like a drawing board with independent coordinates. Variables in Donald need to be declared before their use. There is no procedural assignment in Donald. In Donald, the equals sign ‘=’ denotes a definition, rather than a procedural assignment as in Eden. Each statement in Donald is terminated by a carriage-return. For example, Listing 2.1 below shows sample Donald definitions for specifying a circle within a square on the screen (see Figure 2.8 for the result in a screen capture).

```
%donald

viewport testDrawing

rectangle theRectangle
circle theCircle
int x1, y1, x2, y2, centreX, centreY, radius

x1 = 100
y1 = 700
x2 = 300
y2 = 900
centreX = x1 + (x2 - x1) div 2
centreY = y1 + (y2 - y1) div 2
radius = (x2 - x1) div 2

theRectangle = rectangle({x1,y1},{x2,y2})
theCircle = circle ({centreX,centreY},radius)
```

Listing 2.1: Definitions for drawing a circle within a square in Donald



```
int radius, centreY, centreX, y2, x2, y1, x1
circle theCircle
rectangle theRectangle

AGENT input
radius = (x2 - x1) div 2
centreY = y1 + (y2 - y1) div 2
centreX = x1 + (x2 - x1) div 2
y2 = 900
x2 = 300
y1 = 700
x1 = 100
theCircle = circle({centreX, centreY}, radius)
theRectangle = rectangle({x1, y1}, {x2, y2})
```

Figure 2.8: Screen capture of a sample Donald drawing

Scout

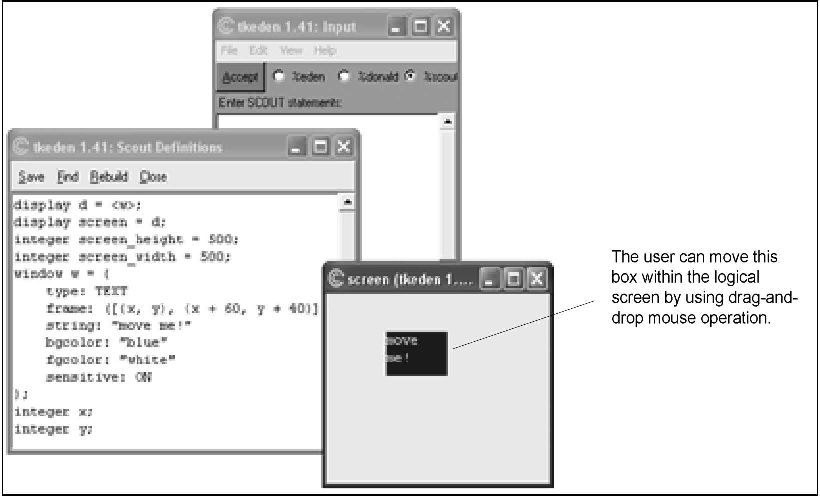
Scout (SCreen layOUT) is a definitive notation designed for specifying the contents and layout of windows on screen. It was introduced and implemented by Y. P. Yung in 1992 [Yun92]. The basic data types include `integer`, `string`, `point`, `box`, `frame`, `window` and `display`. In a Scout script, as in Donald, variables need to be declared before use, and the equals sign '`=`' denotes a definition. Each statement is terminated by a semi-colon '`;`'. Listing 2.2 shows sample Scout definitions that specify a window containing the string "move me!" that can be moved about the screen by a drag-and-drop mouse operation.

```
1.   %scout

2.   display d;
3.   window w;
4.   integer x,y;

5.   w={
6.     frame: ([{x,y},{x+60,y+40}])
7.     type: TEXT
8.     string: "move me!"
9.     sensitive: ON
10.    bgcolor: "blue"
11.    fgcolor: "white"
12.  };

13.  d=<w>;
14.  screen= d;
```

```
15.  %eden

16.  proc dragdrop: w_mouse_1{

17.  /* button pressing */
18.  if(w_mouse_1[2] == 4){
19.     logicalx=w_mouse_1[4];
20.     logicaly=w_mouse_1[5];
21.  }

22.  /* button released */
23.  if(w_mouse_1[2] == 5){
24.     x=x+w_mouse_1[4]-logicalx;
25.     y=y+w_mouse_1[5]-logicaly;
26.  }

27.  }
```

Listing 2.2: Definitions for specifying a movable window in Scout with a supporting Eden action

Lines 2-4 contain some Scout variable declarations. Lines 5-12 define the window. Lines 13-14 put the window on the screen. Lines 15-27 is an Eden action that recalculates the coordinates of the window whenever the modeller does a drag-and-drop mouse operation. The action is triggered by the variable `w_mouse_1` (line 16). Because the window `w` is defined to be sensitive (line 9), redefinitions of this variable are generated by mouse events. The variable `w_mouse_1` records the current status of the mouse and cursor position using an Eden list. A screen capture of the result of inputting the definitions and action in Listing 2.2 is shown in Figure 2.9 below.



```
display d = <w>;
display screen = d;
integer screen_height = 500;
integer screen_width = 500;
window w = {
    type: TEXT
    frame: ([{x, y}, {x + 60, y + 40}]
    string: "move me!"
    bgcolor: "blue"
    fgcolor: "white"
    sensitive: ON
};
integer x;
integer y;
```

The user can move this box within the logical screen by using drag-and-drop mouse operation.

Figure 2.9: Screen capture to illustrate the use of Scout

The way in which Eden, Donald and Scout are integrated in TkEden is depicted in Figure 2.10. The core of TkEden is the Eden interpreter that is responsible for automatic dependency maintenance and for generating output. The Donald and Scout scripts specified by the modeller are translated to Eden script and actions to be interpreted by the Eden interpreter. Other definitive notations can also be integrated into TkEden in a similar way. This is in keeping with Eden's primary role as an evaluator for definitive notations.

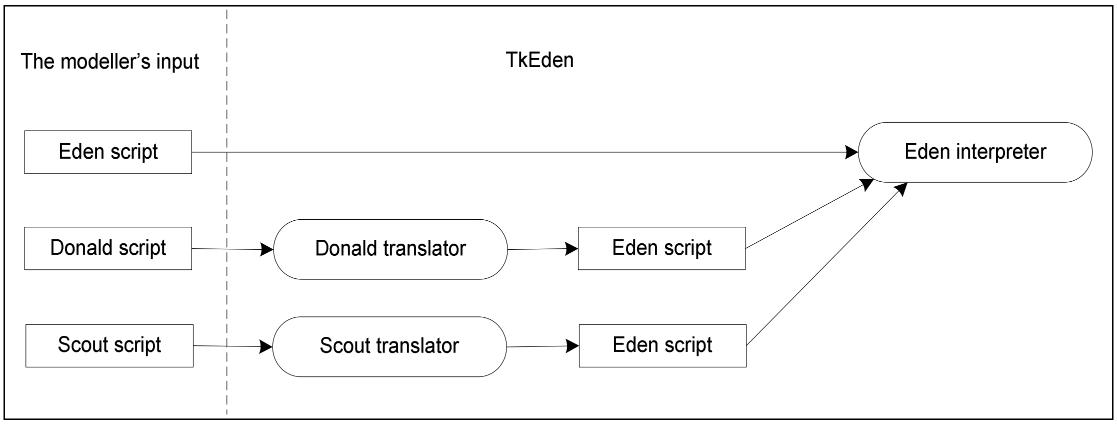


Figure 2.10: The integration between Eden, Donald and Scout in TkEden

TkEden is the principal tool for EM, and has now been used by several hundred students at Warwick University. A few hundred models have been built by using TkEden and its variants and many of these are available to download from the web [EMWeb]. Most of them include descriptions and tutorials that help users to learn about EM. Throughout this thesis, we shall use TkEden to build illustrative examples. In addition, in chapter 7, we evaluate TkEden as a tool to support modelling activities in the DMF. The prospects for introducing new EM tools will also be discussed in both chapters 7 and 8.

2.3.2 The Language for Specification and Description (LSD)

The LSD (Language for Specification and Description) notation was originally developed by Beynon in collaboration with Norris of British Telecom in 1986 to support a special form of agent-oriented analysis for concurrent systems [Bey86a, Bey88c]. Subsequently, Slade [Sla90] developed the design of LSD and investigated the scope for using LSD to generate executable models. The key activities that are performed in an LSD analysis are *identifying agents* and *classifying the observables that are deemed to govern their interaction*.

An LSD account of an agent classifies the observables associated with it into five categories, namely *state, oracle, handle, derivate* and *protocol*. Their meanings are briefly explained as follows:

- state – contains observables that the agent owns. The existence of these observables is dependent on the existence of the agent.
- oracle – contains observables that the agent may respond to.
- handle – contains observables that are conditionally under the agent's control.
- derivate – contains definitions that specify indivisible stimulus-response relationship between observables.
- protocol – contains possible actions that the agent can perform subject to certain enabling conditions being met.

For an illustrative example, consider the description of the Vehicle and Driver agents from a vehicle cruise control simulation in Listing 2.3 [Bey92a, Adz99]. Note that an observable does not necessarily have a single exclusive classification. For example, `engineStts` is both an oracle and a handle for the `Driver` agent. Also, because the same observables can be associated with different agents, an LSD account helps to express the subjective views of the agents which are not explicitly modelled in an ISM.

```
agent Vehicle{

  state:
    mass     /*total mass of car*/
    actSpeed /*actual speed*/
    accel    /*acceleration*/
    windF    /*wind resistance force*/
    brakF    /*braking resistance force*/
    ...

  oracle:
    brakePos /*brake position*/

  derivate:
    windF = windK * sq(actSpeed)
    brakF = brakK * actSpeed * brakPos
    accel = (traceF-brakF-windF)/mass
    actSpeed = integ_wrt_time(accel,0)
    ...
}
```

```
agent Driver{

  oracle:
    engineStts /*engine settings*/
    cruiseStts /*cruise settings*/
    ...

  handle:
    brakePos
    engineStts
    cruiseStts
    ...

  derivate:
    brakePos = user_input(brakPos_Type)

  protocol:
    (engineStts == off)-> engineStts = on
    (cruiseStts != off)-> cruiseStts = off
    ...
}
```

Listing 2.3: Fragments of the Vehicle Cruise Control Simulation in LSD notation

There is no prescribed role for LSD analysis within the DMF because in general we can perform modelling activities without explicitly thinking about the concepts of LSD. However, LSD provides a means to classify observables that can be used in a variety of ways in connection with the construction of ISMs. An initial LSD analysis has been the basis of several ambitious EM models (such as the five-a-side football simulation [Tur00] and the train arrival and departure model [Bey90b]), but modellers have the freedom to derive their own ways of analysis according to context. For this purpose, the LSD notation can be interpreted and used in three ways: *for construal,* *for description* and *for specification*.

For construal. An LSD analysis can be used to support the activity of constructing an ISM. In this context, LSD notation is used to frame an account that expresses the way in which the modeller construes the phenomenon of current interest. There are two aspects to this construal: the identification of agents as groups of observables (‘agentification’), and the exploration of the role that various observables play in mediating their interaction, as typically disclosed through projected or actual role-playing activity. For example, by thinking about the interaction between the driver and the vehicle, we are led to classify `brakePos` as an oracle observable for the `Vehicle` agent but a handle for the `Driver` agent. An LSD account of a phenomenon helps the modeller to acquire and document evolving understanding of the phenomenon that can be provisional, subjective and personal. It typically helps to stimulate a rich set of questions concerned with the phenomenon that would be difficult to identify without thinking about agents and their access privileges and interactions.

For description. The LSD notation can be used to provide information about the current status of an ISM. For instance, it can be very hard to know how to interact with a complex ISM. We can use LSD to document the intended user interactions, or – more generally – experimental interactions representative of the most interesting scenarios known to the modeller. In this way, an LSD description serves a useful role in communication between modellers as a complement to the sharing of an ISM.

For specification. The LSD notation can be used as a specification language to circumscribe understandings. However, the LSD notation has no formal operational semantics, and an LSD specification cannot be directly or automatically translated into an executable model (for more discussion of the issues involved the reader can refer to [Bey88a, Bey90a]). To give an operational interpretation to an LSD account additional assumptions have to be introduced.

2.4 Summary

In this chapter, we have identified the roots of EM in a commonsense way of understanding phenomena. EM is associated with the conception of concurrency as experienced in our everyday life in terms of observation, agency and dependency.

This philosophy leads to the Definitive Modelling Framework (DMF), which is characterised by an interactive style of model building based on a distinctive mode of representation and style of modelling activity. The DMF represents agents using groups of definitions and actions. Modelling activities in the DMF include agentification, role-playing and collaborative definitive modelling. The DMF supports an interactive style of model building, in which the use of the computer resembles the use of an instrument. We have also introduced the TkEden tool and its associated definitive notations that are used in practical EM, and described the role of LSD in EM.

3 System Development

Promising applications for EM concepts have been identified in many research areas. These include: software system development [Sun99, Ras01, Maa02], education [Bey97, Roe02], business process reengineering [Eva01, Che02], and product design [Fis01]. These wide ranging potential applications of EM give some empirical evidence of the generality of EM concepts in respect of system development. However, there has been no attempt so far to account for this generality explicitly. In this chapter, we shall explore the generality of EM concepts and argue that the philosophical foundation of EM is domain-independent. In other words, EM concepts can be framed and applied to the development of any kind of system.

The structure of this chapter is as follows. In the first section, we shall review two of the most important research strands in the literature on system development. These strands will be contrasted with the EM perspective on system development discussed in later sections. In addition, we shall specify the aims for the application of EM to system development. In section 3.2, our discussion will focus on thinking about intrinsic properties of systems and their relationship to an EM perspective on system development. In section 3.3, we focus on thinking about system development activities and discuss how EM takes these activities into account. In section 3.4, we conduct a case study of modelling a dishwasher system in both EM and another common modelling approach, object-oriented modelling using UML. Our aim is to explore the differences between these two styles of system modelling. In section 3.5, we review possible ways in which the aims of applying EM to system development set out in section 3.1 can be achieved.

3.1 Research on system development

System development is an area of research that has interested researchers from many different disciplines including engineering, computer science, business studies and philosophy. Despite the general interest, however, there is no agreed definition of ‘system’. For example, Jordan [Jor68] lists fifteen different definitions of the term ‘system’; the most updated version of Oxford English Dictionary Online [Oed02] also has more than ten different uses of the word. The main reason for this terminological vagueness is that researchers tend to propose their definition of a system as a basis for a prescription for how system development should proceed. This chapter makes no attempt to seek a compromise from the sea of definitions of the term ‘system’ in the existing literature or to introduce a new one. The purpose of this chapter is to investigate how EM concepts help system development and relate the general principles behind an EM approach to system development to mainstream thinking on system development.

By looking at the literature, we can find two principal perspectives from which system development is discussed. One is generally based on thinking about the nature of systems. The other is based on thinking about activities involved in system development. We shall first discuss the main ideas from both these strands of research which will be contrasted with the EM perspective on system development to be discussed in the rest of this chapter.

3.1.1 System development guided by thinking about systems

Much research on approaches to system development has been guided by thinking about systems. In philosophical terms, the focus is on the ontology and epistemology of systems. Such approaches aim to come up with guidelines about how systems should be studied and developed by first discussing what a system is and how the properties of a system can be determined. The tension between a *reductionist* and a *holistic* view of explaining phenomena in the world has had an important impact on the ways in which system development is conceived.

In the reductionist view, every phenomenon can in principle be explained through an application of a hierarchy of natural laws. Such a reductionist outlook is expressed in the following quotation from Dawkins [Daw86]: “an ecosystem is explained in terms of organisms whose behaviour is explained in terms of proteins and macromolecules and DNA code… until ultimately all behaviour is reduced to The Theory of Everything.“. Reductionism presumes that the behaviour of the whole can be understood entirely from the properties of its parts.

The counter to a reductionist view sees the whole as more than the sum of its parts. Checkland [Che93] justifies this view with reference to a simple example: carbon dioxide cannot be reduced to carbon and oxygen because the properties of carbon dioxide such as inter-atomic distance and bond angles are irreducible. The rejection of reductionism leads to another way of thinking about systems which Capra [Cap96] describes as the holistic view. In the holistic view, a system can only be defined as an integrated whole rather than simply as a collection of parts. The properties of a system *emerge* from interaction between parts that cannot be understood by just studying every part alone [Che93, Cap96].

In practical system development, there is always a tension between these two views – it is not possible to adopt one view and to ignore the other. However, different people might put more emphasis on one view rather than the other. The implications of putting the emphasis on each view can be contrasted as follows:

Abstract representation vs. embodiment. Putting the emphasis on the reductionist view, one tends to believe that knowledge about a system can be reduced to knowledge about components and expressed by using abstract representations such as formal documents. System development is mainly aimed at identifying and building components that constitute the system and creating formal documents that purport to *completely* explain these components and their interaction.

In contrast, when adopting the holistic view, one is more aware of the limitations of formal documents as a representation of knowledge about a system. The emphasis in holistic approaches to system development has been on providing a management structure for synthesising knowledge of the proposed system drawn from a variety of different viewpoints. There are two key principles of this approach to system development: the documents and artefacts generated in the system design embody knowledge about the system that is always open for interpretation; the emergence of the system is associated with management processes for organizing interpretations based on several viewpoints.

Approaches to system development that are holistic in the sense that they take account of many viewpoints on system design may nevertheless adopt representations for systems that are reductionist in spirit (e.g. object-oriented representations). Other approaches to system development, such as the use of genetic algorithms or neural nets, adopt holistic representations. Such approaches can deal effectively with issues of system optimization where the interpretation is constrained, but do not appear to be so well-suited to creative system development where open human mediated interpretation is required.

Functionality vs. realism. The reductionist view favours the idea that a system can be developed by putting together a prescribed set of functionalities. System development involves identifying and creating the components that can achieve the functionalities necessary to meet the objectives of the system (as e.g. in use-case driven system development [Jac92]). In contrast, a holistic view favours the idea that a system should emerge from modelling – or constructing artifacts within – the projected real world environment for its operation. This is consistent with the philosophy behind alternative object-oriented approaches to system development, which proceed by modelling real world objects without first prescribing the functionalities of the final system.

3.1.2 System development guided by thinking about development activities

In the system development literature, thinking about systems is complemented by thinking about the human activities that are involved in system development, especially those activities that relate to the design of a system. Research interests are along the lines of domain-independent theory of design, empirical studies of design activities, and empirical studies of designers [Log95]. The most relevant research within the scope of our discussion is on the comparison of two paradigms for describing design activities made by Dorst and Dijkhuis [Dor95]: the *rational problem solving paradigm* and the *reflection-in-action paradigm* (cf. Table 2 in Appendix E).

The rational problem solving paradigm originates in theories of rational problem solving advocated by Simon [Sim68]. It sees design as a rational problem solving process. According to Simon, problem solving can be seen as a search process over a solution space. The problem definition is assumed to be stable, and it determines the solution space where a solution lies. In this paradigm, designers are information processors in an objective reality. Here we quote some comments about the rational problem solving paradigm from Dorst and Dijkhuis [Dor95]:

> *“Seeing design as a rational problem solving process means staying within the logic-positivistic framework of science, taking ‘classical sciences’ like physics as the model for a science of design. There is*

much stress on the rigour of the analysis of design processes, 'objective' observation and direct generalizability of the findings."

It is of interest to note in passing that other researchers, such as Gooding, would take issue with this characterisation of science as rational problem solving (cf. subsection 2.2.3).

The reflection-in-action paradigm advocated by Schön [Sch83] takes a radically different view of the design process. According to Schön, every design problem is unique. The design process is seen as a "reflective conversation with the situation". The designer takes actions according to the current problem situation; these may in turn change the situation on which new actions will be based. In this paradigm, the designer is essentially constructing his or her reality [Dor95].

The implications of adopting these two different paradigms can be contrasted as follows:

Objective observation vs. subjective interpretation. The rational problem solving paradigm places much emphasis on objective observation made by the designer during the design process. This promotes the idea that system development should be done according to well-recognised or standardised methods. In contrast, the reflection-in-action paradigm places much emphasis on the subjective interpretation of the designer. This promotes the idea that system development can be guided by psychological research on cognition.

Generality vs. uniqueness. The rational problem solving paradigm emphasises the importance of identifying the commonality of problems, drawing on abstract knowledge and theories, and reuse of general principles and solutions. System development is typically seen as rationally guided by generic methods. The reflection-in-action paradigm emphasises that every design problem is unique. System development is a negotiation of meaning depending on the designer's situational experiences.

3.1.3 Aims of EM perspective on system development

The EM perspective on system development emphasises capturing the system developers' construal of the emerging system within its environment. EM is not primarily concerned with modelling the system but modelling for the construal. An EM model for system development can embody a variety of knowledge and experience of the modeller or system developer that cannot otherwise be easily expressed using conventional approaches. EM is in the spirit of constructivist approaches to system development which emphasise “the mental and social nature of the construction of forms of information” [Cro96]. The abstract merits of a constructivist approach to system development are typically discussed without reference to techniques for implementation (cf. [Ras01]). EM provides commonsense concepts and a philosophy of system development that are not only useful for academic discussion of system development but are also helpful in practical system development. Our aims in adopting an EM approach to system development are:

- To promote flexible system design
- To create more reliable systems
- To support the cognitive needs of system development
- To facilitate collaborative work

We shall explain how we believe EM can help us to achieve these aims at the end of this chapter.

In the previous two subsections, we have discussed two ways in which researchers have approached the subject of system development, namely by thinking about the intrinsic properties of a system and by considering the activities involved in system development. These two ways to approach system development are respectively associated with two cultures: research into systems theory, and research into problem solving and design. Both these cultures have an extensive literature but seem to be quite separate.

Modern computer science cannot avoid engaging with both cultures in the construction of large-scale computer-based systems. (e.g. object-oriented analysis(OOA) and design (OOD) is an example of this kind of engagement). Whilst both cultures have quite well-explored foundations, it seems to be hard to put them together except in *ad hoc* ways. The practical consequences are to be seen in the difficulties of integrating OOA and OOD, business process modelling and software development [Fer01], and project management with product design.

In the next two sections, we shall consider how the EM perspective on system development relates to both the research cultures described above (see Figure 3.1). In section 3.2, we shall explain the EM perspective on system development as it relates to thinking about systems. The discussion will focus on three intrinsic properties of systems: complexity, predictability and unity. In section 3.3, we shall explain the EM perspective on system development as it relates to thinking about activities of system development. The discussion will be focused on three aspects of the activities: cognitive, collaborative and methodological. Explaining the EM perspective from these two directions has two purposes. Firstly, it offers a convenient way to contrast it with other perspectives discussed in the previous two subsections. Secondly, we believe that the picture of EM perspective on system development can only be completed by combining complementary discussions from both research directions.

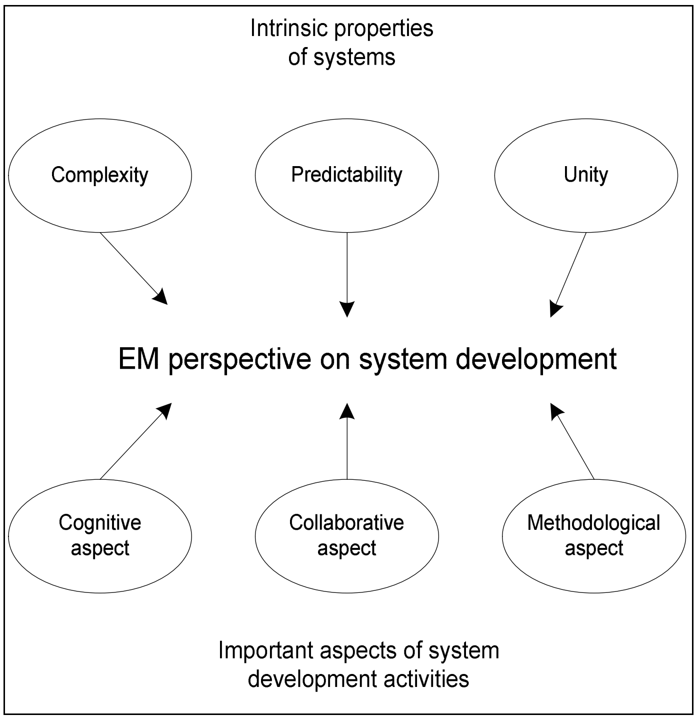

Intrinsic properties of systems
Complexity
Predictability
Unity
EM perspective on system development
Cognitive aspect
Collaborative aspect
Methodological aspect
Important aspects of system development activities

Figure 3.1: EM perspective on system development

3.2 From thinking about systems

Research on systems is generally based on thinking about the parts and the whole of a system. A reductionist view emphasises the importance of parts – it seeks a decomposition of the whole. A holistic view emphasises the importance of wholeness – it is concerned with the emergence of the parts from the whole. All discussions of systems from a traditional perspective centre upon the notions of the parts and the whole.

EM takes a radically different view. We believe that it is more fruitful to take a more relaxed view of part-whole relationships, and think about systems in terms of *observation*. That is to say, we allow the discussion of observation to shape the way we think about systems. Since observation presumes no fixed conception of the part and the whole, this naturally leads to a flexible conception of a system boundary. Systems are the products of observation. Systems emerge from observation. On this basis, we believe that observation is prior to concern for whether a system is best conceived in terms of decomposition or emergence.

Anything that we call a system has three intrinsic properties: complexity, predictability and unity. These properties are intrinsic because we would not call something without all these three properties a system. We shall study each of these in terms of observation along with the implications for system development in relationship to:

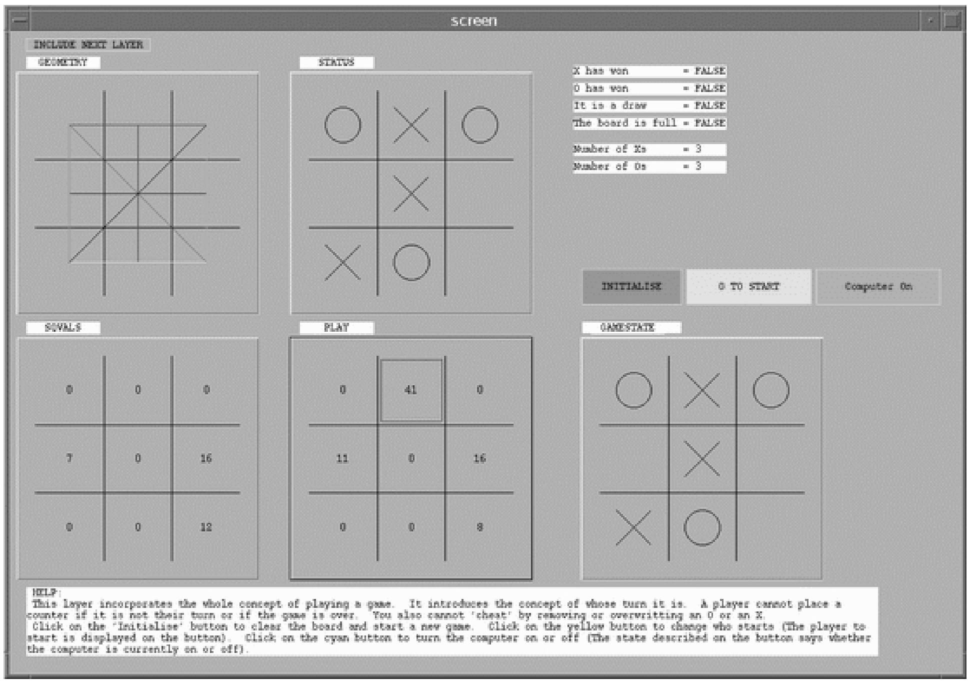
- Revealing hidden assumptions
- Exposing complexity in what is superficially simple
- Modelling unreliability
- Modelling for system development that is situated rather than conducted in isolation.

3.2.1 Complexity

Complexity is intrinsic to any system as is especially evident where system development is concerned. This is because we call something a system only when we realise or appreciate its complexity. For example, what we normally call a ‘television’ is a ‘television system’ from the viewpoint of the people who developed it or maintain it. The complexity of a given system or a target system stems from the many viewpoints from which it can be observed. The complexity of a system is reflected in the difficulty of making sense of these viewpoints (e.g. in understanding the relationship between them). Acknowledging that complexity is intrinsic to a system, our real concern in dealing with the complexity of a system in the process of development is with how to manage it and not how to reduce it.

In EM, complexity can be managed by gradually building up a series of observations and representing them within an EM model. Each observation in the EM model reflects what a developer can understand about the system. The observation, dependency and agency that are embodied in an EM model represent the construal of the system in view of the developer.

It is this construal that manages the complexity and from which the properties and behaviour of the target system eventually emerge. In this respect, EM is more in line with a holistic view than a reductionist view. However, in EM, the system emerges from observations rather than from parts. The distinction between observations and parts is an important one. Studying a system in terms of emergent parts promotes the idea that we can distinguish a specific mode of analysis and a particular viewpoint on the system. Studying a system in terms of emergent observations does not restrict the ways to analyse the system, and therefore encourages conscious management of multiple viewpoints in terms of dependency and agency. EM encourages the evolutionary development of a system with arbitrary complexity from the simplest to the deepest level of understanding. An illustrative example, the noughts and crosses (OXO) model, described in [Gar99] shows this quality. A screen capture of the model is shown in Figure 3.2 below. The purpose of building this model is to study the cognitive processes involved in playing an OXO game. The model was built in an incremental way so that the complexity of each layer of analysis (termed 'cognitive layering') is added as the modeller understands the previous layer. The end result is a comprehensive model that metaphorically represents many aspects of the cognitive processes in playing the OXO game.



```
X has won          = FALSE
O has won          = FALSE
It is a draw       = FALSE
The board is full  = FALSE

Number of Xs       = 3
Number of Os       = 3
```

```
HELP:
 This layer incorporates the whole concept of playing a game.  It introduces the concept of whose turn it is.  A player cannot place a
counter if it is not their turn or if the game is over.  You also cannot 'cheat' by removing or overwritting an O or an X.
 Click on the 'Initialise' button to clear the board and start a new game.  Click on the yellow button to change who starts (The player to
start is displayed on the button).  Click on the cyan button to turn the computer on or off (The state described on the button says whether
the computer is currently on or off).
```

Figure 3.2: The OXO model

3.2.2 Predictability

Predictability is intrinsic to any system because the essence of systematic behaviour is that what happens and can happen is predictable, at least to the extent that the states encountered are preconceived to be possible. The notion of system presumes phenomena (associated with the components of the system and their interaction) together with a system conception on the part of the observer (associated with familiar observations and expectations within a preconceived observational frame). By way of illustration, the *phenomena* that underlie a library system embrace the movement of books into, out of and around the library and the manipulation of catalogues and card indexes. The *observational frame*[1] for such system excludes the height of the chief librarian or the number of steps between floors. *Systematic behaviour* is primarily concerned with the movement of books between identifiable abstract locations such as ‘at shelf-mark Q32.4’, ‘at reception’ or ‘on loan to borrower Smith’. The *familiar observations and expectations* feature actions such as the transfer of a book from a library assistant to a borrower at the checkout. An action such as handing over the book after first tearing out some pages on the part of the assistant would be outside the scope of preconceived systematic behaviour.

The primary focus in traditional approaches to information system development, such as Jacobson’s OOSE [Jac92] is on analysing abstract patterns of interaction that prescribe systematic behaviour without explicit regard for the specific phenomena that will implement them. Critics of this style of OO development argue that the preliminary stages of the system development should involve a more comprehensive analysis of the behaviour of the objects that supports a richer model of the environment in which the system operates. Whichever approach is adopted, the principles used to specify the methods in such objects and to ensure that their combined behaviour is predictable involve an abstraction from situation. This abstraction is ill-suited to relating the system behaviour as conceived to the underlying phenomena involved in realising the system. In EM, a close relationship between the current state of an EM model and the situation it represents is maintained through conducting ‘what-if’ interactions and recognising patterns of dependency between observables that are characteristic of the system environment.

[1] Observational frame is the context of a system. Put informally, it is the ‘things one would expect’ from the system.

System predictability is closely related to system reliability. Especially in the context of system development, creating reliable behaviour is the central concern. Reliability is a measure of the exactness of the match between the intended system behaviour and the system's behaviour in use. There are two views of reliability:

- Reliability as a matter of endurance – In this context, our concern is with expected failures. For example, we need to know the extreme values of strength and stress for a car's wheels and test them at the margins. In such testing, the focus is on dealing with the failures that would be expected under extreme conditions. Reliability of this nature can be quantified by gathering measurements and statistics about failures. In this aspect, system reliability can be assessed and predicted by mathematical formulation.

- Reliability as a matter of accident – In this context, our concern is with unexpected failures. The focus is on exposing the hidden assumptions and parameters that lead to unexpected system behaviour. Reliability of this nature cannot be quantified.

To ensure system reliability, we need to keep both views in mind. The complaints about the predominance of mathematical literature in journals and at conferences in the field of system reliability [Oco00] are evidence that research on system reliability puts too much emphasis on the first view. In view of the fact that the most disastrous system failures are caused by unexpected failures, typically stemming from ignorance or neglect of some important observables, we need to put more emphasis on the second view.

EM is a particularly suitable paradigm for investigating the second view of reliability. When building an EM model, our primary concern is not with modelling the abstract observations that characterise the system but the phenomena that sustain the system operation in practice. EM leads to a model that is open-ended and extensible, and can supply a richer model of the environment in which the system is used. In such a model, hidden assumptions and parameters can be explored through intervention of the system developer as a super-agent. In principle, an EM model can simulate the target system in use and generate situations of failure quickly and economically. The reliability of the system can be improved continuously through exploring different ways of interacting with the model.

3.2.3 Unity

Unity is intrinsic to any system because in order to see a thing as a system we need to distinguish it from other parts of the world. Traditionally, this is associated with defining the ‘system boundary’, which is usually established at an early stage in system development. In that context, the aim is to fix what should be considered to be part of the system and its relationship to non-system entities based on developer’s knowledge about requirements of the target system. The term ‘system boundary’ gives an impression that there is a sharp distinction between system entities and non-system entities. This may lead to a premature focus of attention on building what lies within the boundary whilst ignoring the most significant concern of how the system interacts with the environment.

An EM model for supporting system development does not specify any system boundary. Observables in the EM model are always open for interpretation either as belonging to the system or to the environment. For this reason, the representation of a system in an EM model is very different from that in a traditional system specification. In such a system specification, a system is often represented in terms of parts that presume that appropriate preconditions will be met in the operational environment. With an EM model, the system is represented in terms of observations that presume no precondition on the environment other than – possibly – that observables preserve their integrity. An EM model serves as a description of the system and the environment. Therefore, the unity of a system can always be negotiated by the developer at any time.

3.3 From thinking about systems development activities

System development is a process that involves a variety of human activities. In this section, we shall discuss the EM perspective on system development with reference to three important aspects of development activities: cognitive, collaborative and methodological aspects.

3.3.1 Cognitive aspect

From an EM perspective, one of the most significant elements in system development is knowledge creation. Not all the knowledge created in system development contributes to ‘the system’. Significant knowledge may relate to scenarios to be avoided in the system. The knowledge generated in systems development in EM is similar in character to the empirical information gathered by the engineer through experiment prior to systems building. The activity of knowledge creation is associated with the construction of an artefact to embody experiences of the environment from which the system emerges. The process for knowledge creation comprises two important activities:

- **Emergence of knowledge** - This activity begins in the realm of the unknown, as epitomised by primitive interactions with state potentially having a subjective and private significance for the developer. The medium for exploring the unknown is an artefact with which interaction is open and whose interpretation is open-ended. An EM model has the quality of such an artefact. It can embody experience of interaction with a referent that is possibly but not necessarily ‘real-world’ (cf. Gooding’s construal [Goo90]). Through interaction with the EM model, we can identify persistent features and contexts associated with the unknown. Practical skills can be acquired to repeat the observation of these persistent features and contexts. By exercising these practical skills, we can identify dependencies and postulate independent agencies from which generic patterns of interaction and stimulus-response mechanisms are identified as knowledge.

- **Exposure of the unknown** - In this activity, the givens are the knowns. The unknown results from the organization of what is known. Tdeppically the knowledge is associated with familiar experiences, and may be perceived as having a universal relevance, being concerned with objective events and interactions. The knowledge is also associated with patterns of interaction with an EM model which are encountered by the designer in analysis. The aim of the analysis is to find conflicts and expose hidden assumptions within the knowledge. By altering the hidden assumptions, we can obtain new insight about the system. The new insight typically has implications and prompts us to make predictions that expose the unknown. The history of the Periodic Table illustrates the essential principle. In the Periodic Table, chemical elements are arranged by atomic number and the way in which their electrons are organised. Early versions of the table contained gaps between entries. The existence of elements in the gaps was predicted before they were actually discovered – often at a much later time. The prediction of missing elements is the unknown that requires further investigation.

These two activities are prior to circumscription of any system knowledge. They complement each other as shown in Figure 3.3. The source of the knowledge is the unknown, and the source of the unknown is the knowledge. The generation of the knowledge and the unknown convolve to provide the basis for learning. Therefore, building an EM Model can be seen as an evolutionary process of learning. This is in line with the reflection-in-action paradigm of design discussed in subsection 3.1.2 - in which the design process is seen as a "reflective conversation with the situation” [Sch83]. EM principles and tools provide direct support for this evolutionary design process.

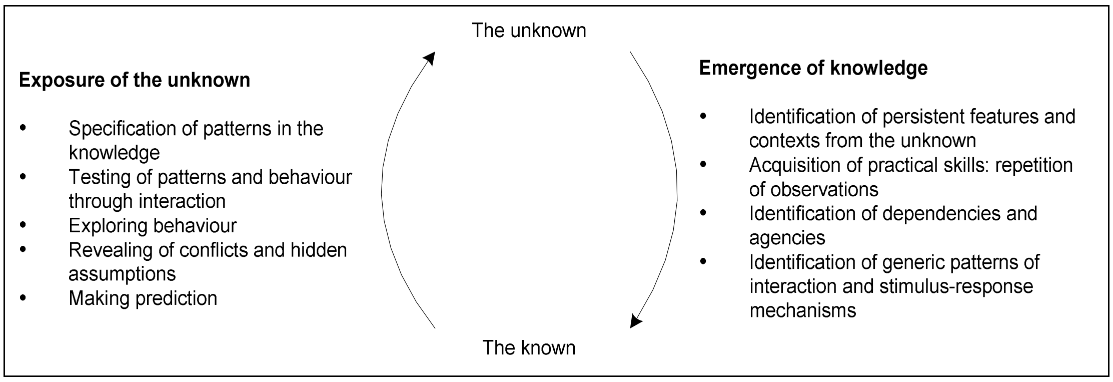

The unknown
Exposure of the unknown
• Specification of patterns in the knowledge
• Testing of patterns and behaviour through interaction
• Exploring behaviour
• Revealing of conflicts and hidden assumptions
• Making prediction
Emergence of knowledge
• Identification of persistent features and contexts from the unknown
• Acquisition of practical skills: repetition of observations
• Identification of dependencies and agencies
• Identification of generic patterns of interaction and stimulus-response mechanisms
The known

Figure 3.3: An EM perspective on knowledge creation in system development

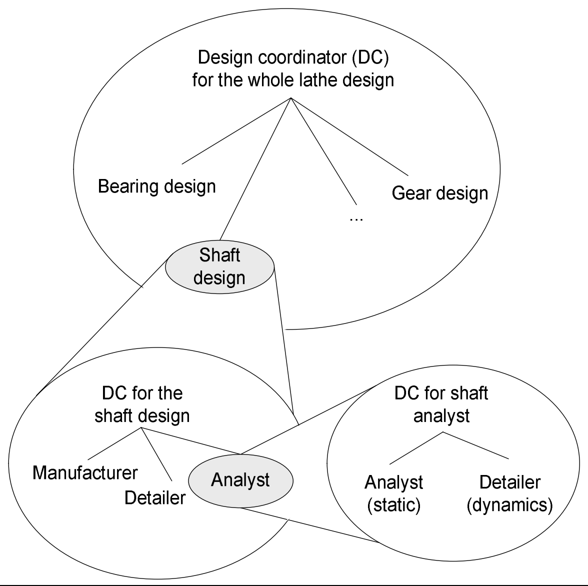
In EM, we build artefacts to assist our construal of experiences that inform the knowledge of the proposed system. The nature of experiences concerned depends on the current status of the design and purposes of the modeller. Such experiences may relate to the general phenomena that sustain the system behaviour, or the more specific environment from which the system will emerge or in which it will operate. A central activity in development of such construal is the correlation of interaction with the artefact and interaction with the relevant phenomena, environment or prototype system implementation. Such activity is similar in spirit to the ‘what-if’ interaction with the spreadsheet that is aimed at establishing a close correspondence between interaction with the artefact and interaction with its referent. Its effect is to generate experiences and knowledge representative of all the many viewpoints on the emerging system. As will be discussed and illustrated in more detail in chapter 5, the emergence of a system is associated with a process for bringing coherence to this body of experiences and knowledge. This gives conceptual integrity to the system design.

3.3.2 Collaborative aspect

System development is an activity that typically involves a group of developers rather than one individual. Collaborative system development imposes extra interrelated technological and social concerns. Sharing explanation and understanding of knowledge and experiences is the key to effective collaboration.

In current practice, documentation plays a central role in sharing knowledge. It is where most knowledge of the target system is captured. This is reflected in current groupware, in which documents and document-related processes predominantly define the logical context for collaboration [Mar98a]. However, what can be shared by documentation is only the tip of the iceberg where system knowledge is concerned. Documentation is best suited for recording explicit propositional knowledge which is derived from only a part of stable experience. The source of propositional knowledge, which is practical knowledge and experience embodied in individual developers, remains for the most part unsharable (cf. the Empiricist Perspective on Learning [Bey97]).

In EM, practical knowledge and experience of the target system can be embodied in an EM model. The distinction between conventional documentation and an EM model is not a matter of style but concerns the character of the knowledge being represented. An EM model serves as a medium for sharing practical knowledge and experiences in collaborative system development. In EM, all the activities associated with collaborative system development, including interactions within the system and the meta-interactions between the developers, can be framed as state-change associated with interaction of agents within the DMF. An agent can be a person, a technological entity (e.g. a computer) or a social entity (e.g. a company). This idea was originally developed in the context of concurrent engineering [Adz94b], but we can adapt the framework to any form of system development. The framework includes a hierarchy of agents where agents at upper levels of the hierarchy coordinate agents directly under them. This framework has been studied in [Adz94b] with reference to a case-study: the design of a lathe spindle.



	Agent		
	Analyst	Detailer	Manufacturer
Design Parameter	Typical observation procedure		
min. shaft diameter	measure loads and compute	get nearest standard diameter	measure with micrometer
step sizes, positions	consult bearing, gear sizes	check avail. Bearing, gear sizes	machine tool set up time and cost
static and dynamic characteristics	measure loads, mass, geometry: compute		
material	compute stress conditions	consult catalogue	heat treatment process & measure
spindle operation	compute use, study cutting dynamics		
manufacturing cost		get quotation from contractor	investigate cost of each manuf. Process
material cost		get catalogue cost	actual cost
assemblability	assign tolerances	check standard components fit	measure dimensions

Figure 3.4: Hierarchy of agents in design of a lathe spindle and agents associated with the shaft design (adopted from [Adz94b])

The left-hand-side of Figure 3.4 shows a hierarchy of agents involved in the design of the lathe. The design coordinator is an agent who is responsible for maintaining an EM model that integrates all EM models from the agents directly under him or her. For instance, in shaft design, there are three agents: analyst, detailer and manufacturer. Each agent has their own view of how the shaft should be designed (cf. the right-hand-side of the Figure 3.4). The three EM models of the shaft constructed by the three agents will typically contain complementary and potentially conflicting observations. The design coordinator's job is to merge the observations and maintain a coherent EM model of the shaft.

As the example shows, this framework uses a meta-EM model to describe the hierarchy of agents who are developing an EM model for the target system. The meta-EM model is accessible to all the developers involved – it allows them to maintain a sense of the whole development process and progress – helping them to maintain conceptual integrity of the system architecture. This framework can accommodate the demands of developing not only systems which once deployed usually remain unmodified (such as a dishwasher) but also systems which need constant adaptation (such as information systems). For instance, a change of personnel during system development can be directly reflected in the meta-model.

Two challenges for collaborative system development relate to the decomposition and the synthesis of a system. Where decomposition is concerned, the system is divided into components for each developer. However, after the initial decomposition, the system may sometimes need to be decomposed again in another way due to a change in the requirement. By explicitly representing dependencies and automating dependency maintenance, an EM model admits several different ways of decomposition without any need to compromise the integrity of the system. Where synthesis is concerned, the system components have to be integrated either for final deployment or for testing purpose. System synthesis involves merging concepts from different developers and resolving conflicts between their viewpoints. It involves explanation of one's own work and understanding other people's work (cf. [Gri98]). Developers can get hands-on experience of components by interacting with the EM models of components. Practical knowledge can be transferred in this way.

From the discussions in this and the previous subsections, we can see that developers can simultaneously use EM models both as a medium for reaching consensus and resolving conflicts in system development, and as a playground for individual experiment.

3.3.3 Methodological aspect

One common motivation for studying system development is to identify some universal principles to help improve the effectiveness of system development. Taking software system development as an example, different formal methodologies have been developed to tackle the so-called 'software crisis'. These methodologies usually provide structured and standardized procedures and techniques aiming at improving the software system development process (cf. SSADM [Eva92], the waterfall process model [Beo76] and the spiral process model [Boe81]). However, there is an increasing amount of research literature questioning the usefulness of formal software system development methodologies (e.g. [Bas92, Fit94, Tru00]). Their main criticism is that in practice people do not follow the formal development procedures provided by methodologies (cf. Table 1 in Appendix E). At best, people modify and adapt methodologies to suit their purpose [Rus95, Gre98b]. This argument exposes the same tension between the rational problem solving and reflection-in-action paradigms for describing design activities discussed in subsection 3.1.2. Critics of formal methodologies argue that in practice every process of software system development is unique, and questions the usefulness of standardising any development procedures.

Most experienced researchers and developers are becoming more convinced that system development should not focus on following formal methodologies. For this reason, attention has recently turned away from standardising system development processes to standardising system representations. At present, the Unified Modelling Language (UML) is the most popular system representation standard. UML provides a set of standard diagrams representing different views of a system in development. Two claims have been emphasised in connection with UML: that it is methodology-independent, and that it is suitable not only for the development of software systems but also for that of other non-software systems [Uml02]. It is beyond the scope of this thesis to discuss whether these claims are justified. Nonetheless, the influence of UML to system development in general is huge. Moving from a process-focused approach (where development is driven by applying a formal methodology) to a representation-focused approach (where development is driven by sharing standard diagrams) of system development is a significant shift in perspective in the research and practice of system development.

The EM perspective on system development is different from both process-focused and representation-focused approaches. Formal systems development methodologies are perhaps most useful as guidelines for a new developer with little experience in system development - EM prescribes no general method for system development [Bey98]. A standardised set of representations of a system cannot do justice to the dynamic variety of views of a system from different system developers – from an EM perspective, views of a system should be formed through negotiation amongst developers involved in every aspect of system development. In summary, we cannot in general standardise either the development process or the system representation. In EM, the emphasis in system development is shifted to interaction with EM models. Both process and representation can be modelled in interaction with an EM model. System building should be guided by interaction between actions and situations, and both process and representation should evolve as the development proceeds. Individual and collaborative system development activities are both mediated by interaction with EM models.

In the next section, we shall compare the application of UML and EM in system development with reference to a case study: modelling a dishwasher system.

3.4 Case study: comparing modelling with EM and UML

In this section, we compare and contrast two styles of modelling that might be used in developing a dishwasher system. The aim is to explore the difference between system development based on EM and on modelling using UML. For EM, we use TkEden as our supporting tool; for UML, we use Rhapsody from I-Logix. Both tools provide mechanisms for creating a simulation of the target system interactively. A model of the dishwasher system has been built by using each tool. The processes of building the models have been recorded in subsection 3.4.1. Observations and comparisons are discussed in the subsection 3.4.2. Our simple case study has been chosen with comparison of the two modelling approaches in mind – our main purpose is not to conduct a detailed analysis of both two styles of modelling but to capture major fundamental differences.

A detailed description of UML and Rhapsody is beyond the scope of this thesis. We assume that the reader has a reasonable amount of understanding and hands-on experience of using UML with any one of its supporting tools (e.g. I-Logix Rhapsody [Ilo02], Rational Rose [Rat02], TogetherSoft ControlCenter [Tog02]). We use Rhapsody in this case study mainly because it is more accessible under academic license, and it contains a simple tutorial of how to build a dishwasher model. However, we believe that Rhapsody includes all the common features available in most of the UML tools in the market – these include support for drawing standard UML diagrams, mechanisms for navigating through the model, code generation for the major programming languages, dialogs for filling in properties using forms, etc. On this basis, our findings are also applicable to other UML tools.

The initial requirement of the dishwasher system is adapted from Rhapsody's online tutorial [Ilo02]:

> *The Acme Company requires software for use in the control of a dishwasher. Acme supplies three custom components to control: a tank, jet, and heater. The dishwasher washes dishes by spraying the water stored in the tank using the jet. The jet sprays the water to rinse the dishes and sends pulses of water to wash them. After the dishes are clean, the heater is activated to dry them. One cycle through the dishwasher consists of washing, rinsing, and drying. You can select*

three different cycle modes:

- *Quick – used when guests are on the way and the dirty dishes need to be cleaned quickly.*
- *Normal – used under normal circumstances.*
- *Intense – used when the dishes are extra dirty, and there is more time to let the dishwasher run.*

It should be possible to switch modes at any time. The dishwasher can be started only if the door is closed. If you open the door while the dishwasher is running, operation halts. When you close the door, operation resumes from the point that it was interrupted. To ensure that the dishwasher continues to work properly, it gives an indication for scheduled maintenance after completing a predetermined number of cycles.

The EM model is generated in its entirety starting from this initial requirement. The UML model is built by following Rhapsody's online tutorial. Comparisons are made based on our experience of building these two models.

3.4.1 Two processes for modelling a dishwasher

Development of the EM model

In this section, we shall briefly describe how we built an EM model of the dishwasher system. The definitions shown in Listing 3.1 and 3.2 below are extracted from the model to illustrate our discussion. For a complete listing of all definitions in the model, the reader is directed to Appendix A.

In EM, we can start modelling with whatever observations we deem to be significant in mind. In the case of the dishwasher, we may first observe a door that can be in two states: open and closed (lines 5-8). We know that the dishwasher can be in three modes: quick, normal and intensive (lines 9-13). The rinsing, washing and drying time are dependant upon the mode selected (lines 14-16). For example, at line 14, the rinsing time is 1 second for quick mode, 2 seconds for normal mode and 8 seconds for intensive mode. We can check that the rinsing, washing and drying time is automatically updated whenever the mode is redefined. Such testing of the model as it is being built is characteristic of EM.

Having introduced the door and the dishwashing modes, we turn to making definitions for the tank, jet and heater. We will only describe the making of the definitions for the tank here (lines 17-29). In addition to being a water container, the tank has two valves: one for filling water and the other for draining water. Lines 18-22 define some states that we can use for defining the water level and valves. Line 23 defines the tank capacity. Lines 24-27 define some interdependent observations about the water level. For example, the water level status can be empty, full or in between empty and full (line 27). Lines 28-29 define the behaviour of the valves. Their definitions show that the washing progress and water level status determine whether the valves are open or closed. We can check that the definitions of the tank are working properly by altering observables they depend on. For example, on redefining water level to 20000 (line25), we should expect the value of `waterLevelStatus` (line 27) to be `tankFull` (line 19).

```
1.   /***********************
2.   AS AN ENGINEER
3.   ***********************/
4.   %eden
5.   /* door */
6.   open is 0;
7.   close is 1;
8.   door is close;

9.   /* wash mode */
10.  quick is 1;
11.  normal is 2;
12.  intensive is 3;

13.  mode is normal;
14.  rinseTime is (mode == quick)?1:((mode
     == normal)?2:8);
15.  washTime is (mode == quick)?1:((mode ==
     normal)?2:8);
16.  dryTime is (mode == quick)?1:((mode ==
     normal)?2:8);
```

```
17.  /* tank */
18.  tankEmpty is 0;
19.  tankFull is 1;
20.  tankSomewater is 2;
21.  valveClose is 0;
22.  valveOpen is 1;

23.  tankCapacity is 20000;
24.  waterFlowPerSecond is 50;
25.  waterLevel is 1;
26.  waterPercent is float(waterLevel) /
     float(tankCapacity) * 100;
27.  waterLevelStatus is
     (waterLevel==0)?tankEmpty:((waterLe
     vel>=tankCapacity)?tankFull:tankSom
     ewater);
28.  drainValve is (progress==washed &&
     door==close &&
     waterLevelStatus!=tankEmpty)?valveO
     pen:valveClose;
29.  fillValve is (progress==go &&
     door==close &&
     waterLevelStatus!=tankFull)?valveOp
     en:valveClose;
```

Listing 3.1: Definitions extracted from the EM dishwasher model

The model is built up incrementally by introducing definitions one by one. In introducing the definitions considered so far, the modeller is acting in the role of an observer who resembles a component engineer. During the modelling process, we notice that there are other roles we can play. We can play the role of a user and define what he or she can observe and act on. Lines 30-49 in Listing 3.2 define three actions for the user – the user can change the washing mode, close the door and open the door. For example, to change the washing mode to intensive in the role of a user we input "`changeMode('I');`" – this redefines the definition of mode as "`mode is intensive;`". The duration for rinse, wash, and dry will be automatically changed to 8 seconds according to the definitions in lines 14-15.

Another role we can play is that of an interface designer who designs the layout of the dishwasher interface. Lines 50-59 define the appearance and position of the quick-mode button in Donald. For example, line 57 defines that if the machine is in quick mode, the quick-mode button lights up in yellow. In order to test or evaluate the model, we can change the washing mode to quick by inputting "`changeMode('Q');`" to see the quick-mode button light up as a result of automatic dependency maintenance.

```
30.  /**********************
31.          AS A USER
32.  **********************/
33.  %eden
34.  proc changeMode{
35.     para char;
36.       if(char=='Q'){
37.         mode is quick;
38.       } else if(char=='N'){
39.          mode is normal;
40.       } else if(char=='I'){
41.          mode is intensive;
42.       }
43.  }
44.  proc openDoor{
45.       door is open;
46.  }
47.  proc closeDoor{
48.       door is close;
49.  }

50.  /******************************
51.  AS AN INTERFACE DESIGNER
52.  ******************************/
53.  %donald

54.  #mode buttons
55.  circle quickButton
56.  quickButton = circle({80,290},5)
57.  ?A_quickButton is
     (mode==quick)?"color=yellow,fill=sol
     id":"";

58.  label qLabel
59.  qLabel = label("Q",{80,300})
```

```
60.  /****************************
61.  DISHWASHER COMPONENTS VISUALISATIONS
62.  ****************************/
63.  %donald

64.  # tank visualisation
65.  rectangle theTank
66.  theTank =
     rectangle({100,50},{200,50+waterPer
     cent!})
67.  ?A_theTank =
     "color=blue,fill=solid";

68.  line theTankTop, theTankLeft,
     theTankRight, theFillValve,
     theDrainValve
69.  theTankTop = [{100,180},{200,180}]
70.  theTankLeft = [{100,70},{100,180}]
71.  theTankRight = [{200,50},{200,160}]
72.  ?theFillValveX is
     (fillValve==valveOpen)?180:200;
73.  ?theDrainValveX is
     (drainValve==valveOpen)?80:100;
74.  theFillValve =
     [{theFillValveX!,160},{200,180}]
75.  ?A_theFillValve =
     "color=red,linewidth=2";
76.  theDrainValve =
     [{theDrainValveX!,50},{100,70}]
77.  ?A_theDrainValve =
     "color=red,linewidth=2";

78.  label theTankLabel
79.  theTankLabel = label("water tank",
     {150,200})
```

Listing 3.2: Definitions extracted from the EM dishwasher model

To make it easier to check the states of the tank and its interaction with other components, we can build up a simple Donald visualisation for it. The definitions in lines 60-79 are all we need to visualise the tank. For example, lines 65-67 define the appearance of the water level in the tank as a size-changing blue solid rectangle. The size of the rectangle changes according to the percentage of water in the tank. Figure

3.5 below shows three kinds of visualisation. The top-left segment shows the layout of the dishwasher interface. The top-right segment shows visualisations of the three major components of the dishwasher. The bottom-left is an animated state-chart whose current state is synchronised with the actual state of the dishwasher.

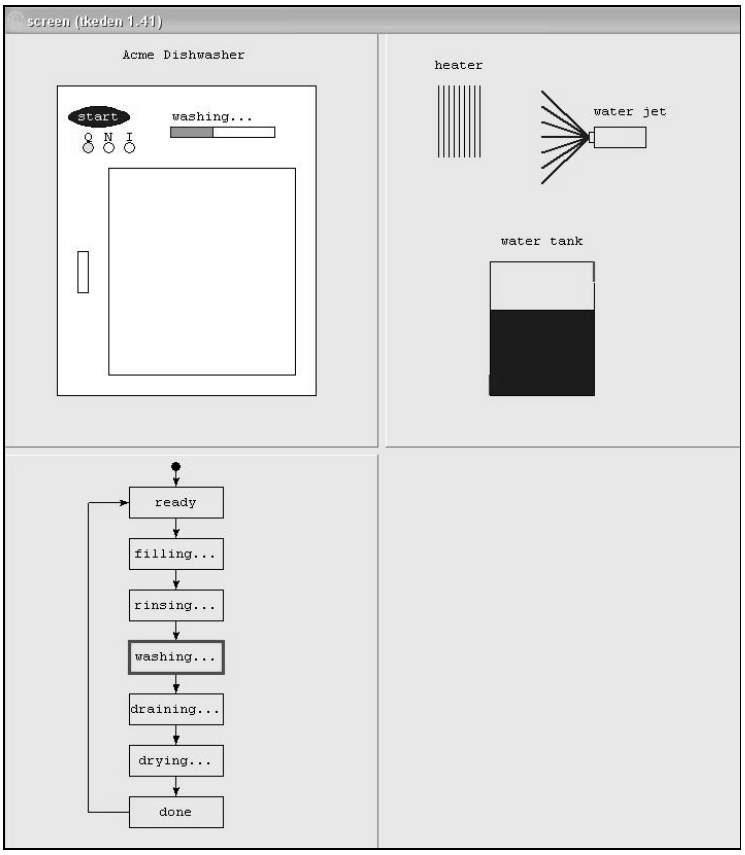


Figure 3.5: Visualisations of the EM Dishwasher model

Appendix C contains an LSD account for the EM Dishwasher model. It illustrates what was in the modeller's mind when constructing this model. It is based on the script model of the dishwasher described above. It can be used as a specification for the actual implementation of the dishwasher.

Development of the UML model

In developing the UML model, we have followed most of the steps in the online tutorial of Rhapsody [Ilo02]. The UML diagrams created are shown in detail in Appendix B. There are four major steps to creating the dishwasher model. Each step creates UML diagrams of a particular type. Altogether we have created five types of diagram: the Use Case diagram, Class diagrams (Object Model diagrams), Sequence diagrams and State Chart diagrams.

The first step is to create the Use Case diagram. This is based on the dishwasher requirement statement. The diagram shows interactions between the system and external actors. It also defines a system boundary (depicted by the bold rectangle in Figure 3.6). Three use cases are identified. The Wash Dishes and Configure Washing Mode use cases belong to the actor Cleansing. The Service Dishwasher use case belongs to the actor Service. This diagram provides a high level view of how the dishwasher will be used.

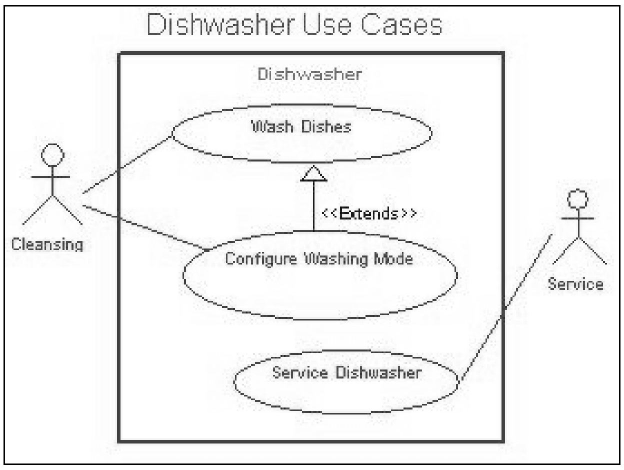
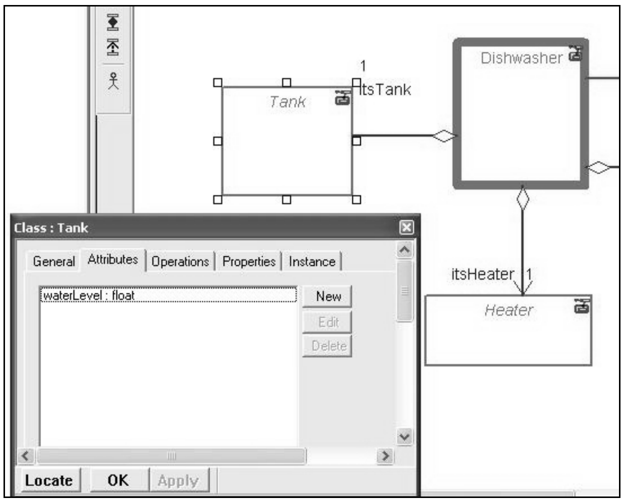


Figure 3.6: The Use Case diagram

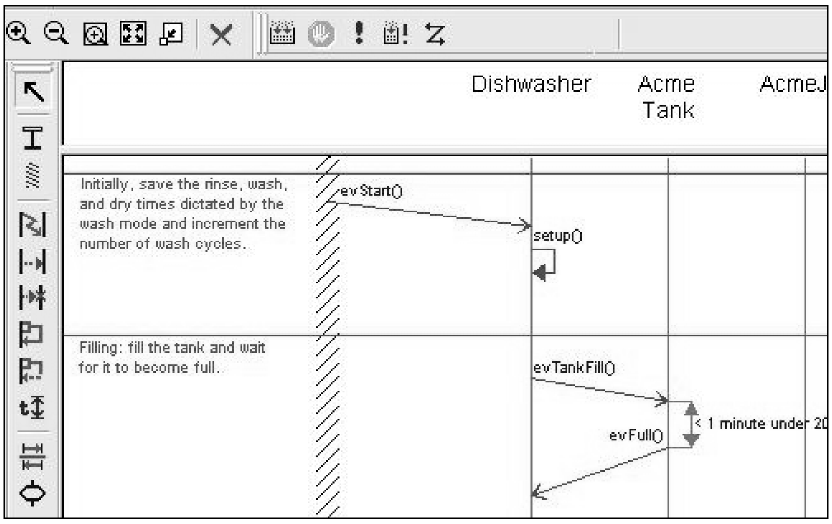
The second step is to create Class diagrams. A Class diagram shows the static structure of the system including classes, attributes, operations and relationships like aggregation and inheritance. Figure 3.7 shows a part of a class diagram – we can see that a `Dishwasher` class owns a `Tank` class, `Heater` class and other classes. We can edit all properties of a class by bringing up a class edit dialogue. For example, the display at the bottom-left corner of the figure shows that we are editing the properties of the `Tank` class.



waterLevel : float

Figure 3.7: Part of a dishwasher Class diagram and a class edit dialogue.

The third step is to create Sequence diagrams. A sequence diagram shows a scenario in the execution of a use case. Typically, it shows a sequence of message passing between different participating objects. Figure 3.8 depicts a part of a sequence associated with the Wash Dishes use case. The arrows in the sequence diagram depict message passing. The time line runs from the top to the bottom of the diagram. In Figure 3.8, we can see two objects, `Dishwasher` and `Tank`, participating in the sequence. First, the sequence is initiated by a event called `evStart()` which is received by `Dishwasher`. The Dishwasher initialises itself by calling `setup()`. Then, the `Dishwasher` sends an event by calling `evTankFill()` to the `Tank` for filling up the Tank. After filling up, the `Tank` confirms it is full by calling `evFull()` back to the `Dishwasher`. All the function calls we specified during drawing the diagram are registered in their corresponding classes. For example, the function `setup()` will be registered in the `Dishwasher` class.



Dishwasher
Acme Tank
AcmeJ
Initially, save the rinse, wash, and dry times dictated by the wash mode and increment the number of wash cycles.
evStart()
setup()
Filling: fill the tank and wait for it to become full.
evTankFill()
evFull()
< 1 minute under 20

Figure 3.8: A part of a Sequence diagram for Wash Dishes use case

The last step is to create State Chart diagrams for each class in the model. A State Chart diagram defines the behaviour of an object by explicitly specifying all the states the object can be in and the conditions for transition from one state to another. Figure 3.9 shows a State Chart diagram for the `Tank`. There are four states: empty, filling, full and draining. Initially, the `Tank` is in the empty state. When an event `evTankFill` has been generated, the `Tank` goes into a filling state. In the filling state, the `Tank` fills water until the water level equals the tank capacity. The `Tank` then enters the full state. An event `evFull` is generated by the `Tank` to notify the `Dishwasher` that the `Tank` is full. Similarly, the Tank can go on to draining and back to the empty state as a response of some system events.

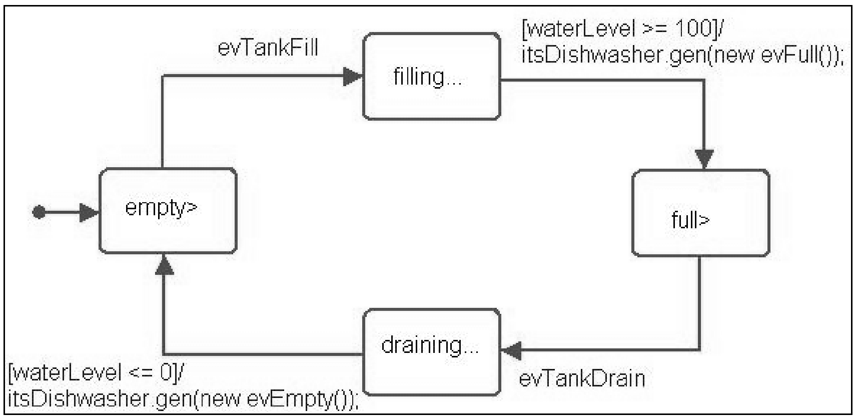


Figure 3.9: A State Chart diagram for the Tank class

After specifying all the UML diagrams, we created the user interface by using Java and linked the interface to the dishwasher system by adding the interface as an `Observer` class. We then compiled the UML diagrams into Java source code and compiled the Java source code into byte code ready for execution. Figure 3.10 shows a snapshot of the dishwasher system in execution. At the right-hand-side of the figure, we have an animated sequence diagram that shows all message passing between objects as it occurs.

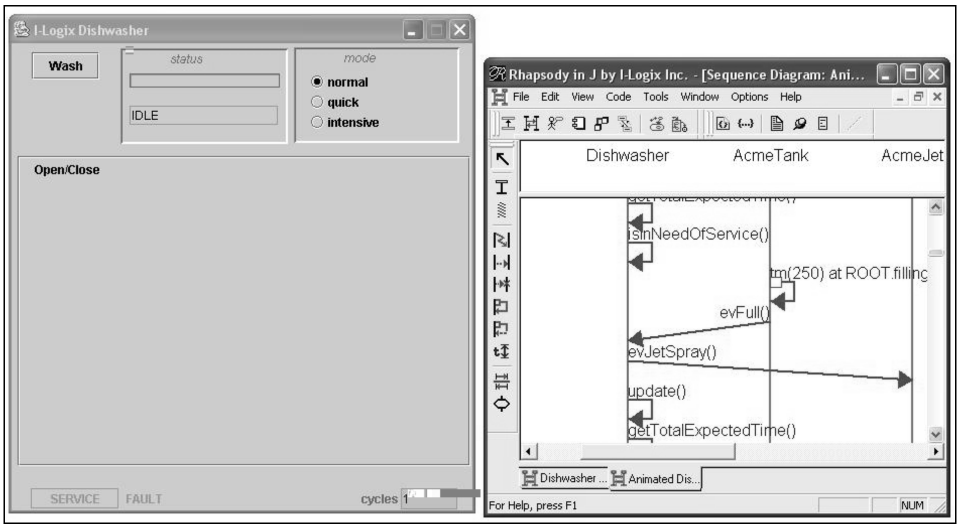


Figure 3.10: A snapshot of running the UML model

3.4.2 Comparisons

In this subsection, we shall make comparisons between EM and object-oriented modelling using UML under five headings: Modelling focus, Interactiveness, Comprehension, Openness and Interfaces.

Modelling focus

We notice that one fundamental difference between the two approaches concerns what is being modelled. EM focuses on modelling the potentially subjective interpretations of the modeller. Observations recorded in an EM model are based upon the modeller’s imagined interaction in the roles of agents acting within or on the target system. In EM, the model-building demands more awareness of the situation in which the target system operates. The design of the system emerges from accounting for different aspects of state as viewed by a variety of agents. UML is focused on modelling the structure and behaviour of the system. All the UML diagrams represent views that are most relevant to the system designer.

Subjective use is outside the scope of modelling with UML. Consequently, once the system boundary is established, the system design is typically constructed in a setting that is isolated from its operating environment and use.

Interactiveness

In EM, the representation of specific states is an intrinsic feature of any model. The modeller can always get immediate feedback on enquiries about any particular states for investigation throughout the development of the EM model. The scope for experimental interactions is only constrained by the modeller's construal. Such interactions may involve adding observables without conceptual change of state, or the consideration of states and behaviours outside the scope of normal use. A model offers a real-time response to an experimental interaction. Further experiments can be conducted immediately after changing the model.

UML diagrams are abstract representations of a system that are not primarily intended to be interpreted with reference to a particular system state. The main role of UML is to specify system behaviour. When the modeller makes an experimental change to a UML diagram, it affects the specification of system behaviour. Such experimentation is quite different in character from open-ended interaction with a specific system state. The modeller may be able to change particular system parameters (e.g. the tank capacity or the length of the rinsing phase in the dishwasher) that affect the system behaviour as whole, but experimental interaction with specific system states is constrained by the specified system behaviour (only initial states that lie in the path of a system execution are accessible). Moreover, every time a UML diagram is changed, it has to undergo two phases of compilation before the modeller can get feedback about system states and behaviours. One phase involves translation from diagrams to the source code of a target programming language, and the other involves compiling the source code into an executable program. In addition, code generation from UML diagrams is not fully automatic – the modeller has to manually resolve programming language specific issues (e.g. the declarations of language specific types). As a result, interactive experiments are more difficult to conduct in UML than in EM.

Comprehension

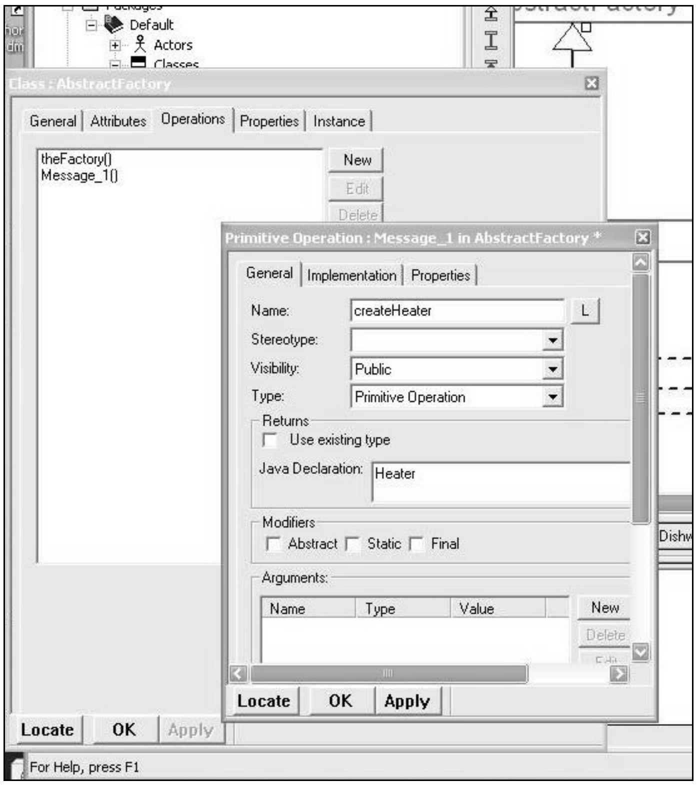
Experimental interaction plays an important role in understanding real world phenomena. The scope and quality of the possible interaction with an EM model facilitates comprehension where specific details of structure, behaviour and performance are concerned. For instance, interaction with the EM dishwasher model allows us to explore the relationship between the behaviour of the fill valve and the filling of the tank. The mechanism can be such that the opening and closing of the fill valve is directly controlled by the dishwashing cycle or by an autonomous stimulus-response mechanism that switches off the valve automatically when the tank is full. Exploration of this nature has no counterpart in the UML model of the dishwasher. Where comprehension is concerned, the function of UML diagrams is to display the predetermined relationships between components rather than to allow the modeller to explore possible alternatives.

A complementary aspect of system comprehension is concerned with understanding the integrity of the system as a whole. In our case study, the modeller cannot grasp this integrity easily when modelling with UML for three reasons:

- The confusing interface – information about a class is scattered throughout different diagrams and interfaces. For example, Figure 3.11 shows two dialogs, one on top of another, that contains information about one class. Bits and pieces of information about the class are scattered throughout tabs and fields whose positions are semantically irrelevant to the model.

- Code generation – In translating UML diagrams to source code, the tool automatically generates additional code, that is not directly relevant to the UML diagrams, making it difficult – if not impossible – for the modeller to understand the source code.

- Consistency between diagrams – There are a lot of dependencies between different UML diagrams. There are few cues to enable the modeller to trace these dependencies easily. This is a well-recognised problem of UML (e.g. [Kim99]). Some suggest introducing automatic consistency checking to the supporting tools [Rob00]. However, even if the consistency can be automatically maintained by a tool, the dependencies cannot be easily comprehended by a human interpreter.



Class : AbstractFactory

theFactory()
Message_1()

Primitive Operation : Message_1 in AbstractFactory *

Name: createHeater
Stereotype:
Visibility: Public
Type: Primitive Operation
Use existing type
Java Declaration: Heater

Figure 3.11: Dialogs showing information about a class

Openness

Openness here is related to two principal questions – to what extent does the modelling style support the modeller in:

- exploring hidden problems relating to the requirements as currently framed,
- adapting the model to deal with changing requirements.

In UML, a fixed set for diagrams, representing different viewpoints on the system under development, guides the system modelling [Smo01]. Modelling activity starts with use-case diagrams that define the system boundary. The next step is to draw a class diagram that defines the main components of the system. Classes define what we need to observe by way of attributes of component in order to develop the system.

Other diagrams are constructed based on the use-case diagram and class diagram. These introduce additional properties and constraints governing the interaction between components until the model is specific enough for implementation. Therefore, modelling with UML resembles a top-down approach to analysis and design. The system is built as a result of *circumscription*. The circumscription takes place from the beginning of the modelling process – use-case diagrams specify the system boundary of interest; class diagrams specify the available observations. These preliminary commitments affect the ability of the model both in accommodating changing requirements and in exploring hidden requirements. The openness of the model is also constrained by the limited viewpoints that a system designer can use.

EM presumes no fixed set of viewpoints. The modelling process involves the identification of observables that may or may not belong to any agents initially. Attributing observables to agents is very different from assigning attributes to classes in UML. The former is a process of discovery and invention whereas the latter is a process of analysis. In EM, agents emerge as groups of observables. The emergence is an ongoing rather than one-off process. It is for this reason that there is no formal syntax for defining an agent for an EM model. In EM, the system is built as a result of *emergence* rather than circumscription. Throughout the process of EM, the system boundary is undefined. Throughout the modelling process, the model always remains open for the exploration of hidden requirements and to accommodate changes in requirements (cf. [Loo98, Loo01, Gog94, Gog96]). Furthermore, EM regards the modeller as a super-agent within the model. The modeller can conduct interaction with any parts of the model to simulate different unexpected exceptional events or conditions in operating the system.

Interfaces

UML does not provide support for specifying interfaces of the target system. Neither can dependencies between states and interfaces of the system be specified by using UML. This means that developers need to use other techniques to design and test the interfaces of the system. In our case study, the interface is created by hard-coding it in Java. EM supports interface design and model-building within the same framework. Dependencies between the states and interfaces are automatically maintained. This feature specifically leads to models for system prototyping that are more stable.

3.5 Aims revisited

As we mentioned in subsection 3.1.3, there are four principal aims in applying EM to system development. In this section, we shall discuss how these aims can be achieved.

- **To promote flexible system design –** An EM model serves as a construal of the system and its situation that is always open to revision. There is no fixed structure within an EM model that is resistant to change - observables in an EM model can be regrouped to reflect structural change without affecting the underlying dependencies between them. As a result, an EM model can typically be easily adapted to accommodate new requirements.

- **To create more reliable systems –** In EM, the target of modelling is not only the system but also the external environments and situations that associated with the system. As a result, system developers can be more aware of the external factors that may affect the operation of the system in actual use. They can simulate various situations of use quickly and economically with the support of the computer-based EM tool. This can help to find identify hidden assumptions that are crucial to the correct operation of the system. The reliability of the system can be tested through interaction with the model. In EM terms, reliability of the system stems from the system developer's experience of stable patterns of observation and interaction.

- **To support the cognitive needs of system development –** An EM model is particularly suitable for recording provisional knowledge and experience that is subjective and unstable in nature. This provisional knowledge and experience is the primary source of creativity and innovation that is important in successful system development. An EM model helps to maintain conceptual integrity in the face of the inherent complexity of the system design. As a result, the system developer can tackle system design problems heuristically.

- **To facilitate collaborative work –** The concepts of EM are domain independent and originate in commonsense. In principle, developers from different disciplines can learn EM easily. An EM model can be used as a medium for supporting communication. Every developer can build EM models of the system within his or her own area of expertise. Different EM models for the same aspect of the system represent different views with potential conflicts clearly exposed. Resolving these conflicts can lead to new system knowledge.

In addition, research into EM provides not only a philosophy for system development but also tools that give practical expression to this philosophy. There are several ways to use EM in system development:

- For the study of existing systems - EM can be used for study existing systems. The product of the study will be an EM model that embodies deep understanding about the system in question. Based on that understanding, we can proceed to improve the existing system or create a new better one. Relevant modelling techniques for this purpose are discussed in [Bey00a].

- For the design of new systems – EM principles and tools can be use to design new systems. The EM model created by the process of design can be circumscribed by producing a LSD specification. The LSD specification can be used as a blueprint for implementing the system.

- For the implementation of new software systems – If a software system is to be developed, we can create an EM model either to serve directly as the new system or as a prototype system from which a suitable conventional system can be generated semi-automatically by translation. In either case the EM model that underlies the software system is very flexible and adaptable to change.

3.6 Summary

In this chapter, we have discussed the EM perspective on system development from two different directions. The first direction involves thinking about the relationship between EM for system development and the intrinsic properties of systems. The second direction involves thinking about EM in relation to important aspects of system development activity. We have also described a case study of building a dishwasher model by using both EM and UML. As a result, we have compared the differences between EM and UML styles of modelling systems. We have discussed the aims of applying EM to system development and how we can achieve these aims.

4 Human Problem Solving

In this chapter, we shall focus on discussing the application of EM to support the human problem solving that is arguably the most important activity in system development. In the first section, we shall discuss the importance of developing principles and techniques to promote computer support for human problem solving. In the second section, we shall explore the relationship between problem solving and programming. We shall introduce some relevant researches on general problem solving and the psychology of programming. We shall argue that conventional programming paradigms fail to give comprehensive support for problem solving. In section 4.3, we shall propose EM as a better approach for supporting problem solving, and illustrate this with the reference to solving a particular type of recreational puzzle. In section 4.4, we shall describe a case study based on a real life timetabling problem.

4.1 Human problem solving using computers

In the early development of computer science, much research focused on formalising the capability of computers to solve problems automatically. For example, Computability and Feasibility research (e.g. [Gar79]) focused on formalising what can and cannot be solved algorithmically by computers in theory and practice. Such research is commonly regarded as the foundation of computer science. University computer science courses traditionally start with teaching the mathematics and logic that underlie this formalised notion of computing.

However, this mainstream conception of computer science has recently been challenged by a number of researchers. The main argument is that the traditional formalist theories of computer problem solving do not take the environmental and human aspects into account (e.g. [Bey92b, Weg97, Wes97]). The agenda of computer science should not only be considering *computer problem solving* but the broader topic of *human problem solving using computers*. Notice that, in this shift of perspective, the emphasis changes from the computer to the human. This implies a change of focus from research on abstract concerns such as proofs and reducibility to research on more concrete situated activities that reflect the dynamic nature of the world and human cognitive processes. Of course, we agree with West [Wes97] that, whilst we must acknowledge the usefulness of formalism, we also need to be more conscious of its limitations.

Human problem solving is described by Martinez [Mar98b] as "…the process of moving toward a goal when the path to that goal is uncertain". He further explains that "we solve problems every time we achieve something without having known beforehand how to do so". This definition of problem solving excludes the process of achieving goals with prescribed steps. For the purposes of our discussion, this is the most appropriate characterisation of human problem solving. Too often traditional computer science research focuses on knowledge representation and proofs associated with executing prescribed recipes that are far removed in spirit from human problem solving.

The question then becomes: what support can computers give humans in problem solving? This chapter will propose an answer based on EM principles. Edmonds et al. [Edm95] acknowledge that there is a tension between 'automating expertise' and 'amplifying human creativity' in computer-supported problem solving. Ideally, both aspects are equally important. To automate expertise, the human problem solver is provided with a set of predefined functionalities that are common in solving a particular class of problems. To amplify human creativity, the problem solver is provided with features for customising predefined functionalities or even for defining new functionalities. Too often research tends to stress the automation of expertise, rather than promoting human involvement.

Gallopoulos et al. use the term "problem-solving environment" (PSE) to describe a computer-based system that "provides all the computational facilities necessary to solve a target class of problems" [Gal94]. The facilities include predefined solution methods, automatic or semiautomatic selection of solution methods, and ways to add new solution methods. Many PSEs are widely used by industries already. Examples are SPSS for statistical analysis, LabView for electronic engineering, Matlab for science and engineering, Mathematica for symbolic manipulation and visualisation, spreadsheets for finance and word processors for publishing. PSEs are usually domain-specific. Their usefulness is bound to their intended domain with the arguable exceptions of spreadsheets and word processors.

By contrast, the purpose of this chapter is to discuss support for general human problem solving using the computer. It is more difficult to provide problem solving support for general computer users than for domain experts because we cannot presume a particular model of the intended users.

There seem to be several options for a general computer user to solve a problem using a computer. An obvious option is to use a software package that is specifically designed for solving the problem at hand. This is the best choice provided that the user can find what they are looking for, and provided that the problem is common enough for software to be already constructed by others. Unfortunately, this is not always the case in reality. Since the notion of problem is subjective and changes over time [Hoc90], it is hard to find a software package to suit for a given purpose. Even if the user can actually find one, the dynamic nature of a problem context may quickly render the software package unsuitable without introducing modifications. In addition, as has been discussed in [Ped97], traditional user interfaces of conventional software packages are usually so rigid that they hinder the ability of the user to approach a problem differently.

Apart from finding and using existing software packages, general computer users still have two other options: to use one of the conventional end-user programming paradigms such as the spreadsheet paradigm, or to construct a program using a conventional programming languages such as Java. However, we shall argue that even these two options are not satisfactory for supporting human problem solving using computers.

In the following section, we shall discuss the complexity of providing adequate support for problem solving using the computer. We shall explore the fundamental difficulties from both traditional spreadsheet and programming paradigms for supporting general human problem solving. The nature and relationship of programming and human problem solving is also explored with reference to some recent research in psychology of programming and problem solving.

4.2 Problem solving and programming

4.2.1 Programming as search in problem spaces

Programming can be viewed as *fulfilling a requirements specification* generated from requirement analysis as part of a conventional software development cycle. For the purpose of our discussion, however, we shall take the broader view that programming is *a form of problem solving*.

Following Newell and Simon [New72], human problem solving can be thought of as navigation from an initial state to a desired goal state in a *problem space*. A problem space represents all possible states of knowledge of the problem solver related to the problem. Problem solving is viewed as a transformation from the initial state to the goal state using a set of cognitive operators. This idea of searching in a problem space has proved to be useful and widely accepted by many researchers. It has also been extended for describing various phenomena of human problem solving. For example, Simon and Lea [Sim74] characterise an induction task that involves developing general rules from particular instances as a search in two interrelated problem spaces: *rule* space and *instance* space (a 'dual-space search'). The problem solver generates possible rules from instances, and tests the rules with new instances; this in turn may invalidate the rules, and result in the generation of new rules. This process goes on and on until the desired rules are found. The concept of dual-space search was later adapted by Klahr and Dunbar [Kla88] in their explanation of scientific discovery, where scientists search in hypothesis (rule) space and experiment (instance) space.

In their research on scientific discovery, Schunn and Klahr [Sch95] suggest a 4-space search. This research was adapted by Kim et al. [Kim97a, Kim97b] for exploring the strategies used by programmers in problem solving. Kim et al. characterise programming as a search in 4-spaces: the *rule*, *instance*, *representation* and *paradigm* spaces. In programming terms, searching in rule space is related to writing program statements; searching in instance space is related to designing and conducting test cases for checking the correctness of the program; searching in representation space is related to maintaining a mental model of the programmer's understanding of the problem; and searching in paradigm space is related to selecting a suitable paradigm for navigating other spaces. Empirical studies with programmers conducted by Kim et al. revealed that the difficulty of programming can be attributed to programmers' intensive searches in representation and paradigm spaces.

Kim et al. has explained the difficulty of programming as a particular kind of problem solving activity. To discuss the challenges of providing computer support for human problem solving more fully, we also need a wider investigation into how people solve problems with and without the support of computer programming.

4.2.2 Difficulties of supporting problem solving by programming

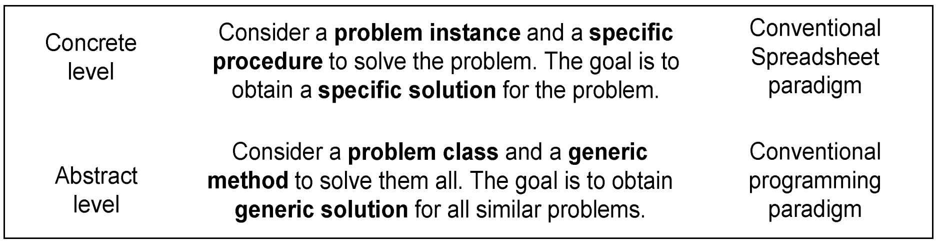
Traditional programming paradigms are suitable for problem solving at the abstract level. They provide a high degree of expressiveness and generality. Programming language designs usually give much support to abstract thinking and may even enforce programmers to think abstractly (cf. ‘abstraction hunger’ described in [Gre98a]). For example, it is generally considered to be more useful to develop a program to sort an arbitrary number of names than to sort a fixed number of names in alphabetical order, even though the problem at hand is to sort just a given number of names.

Hoc and Nguyen-Xuan [Hoc90] point out that in normal problem solving situations the goal is specific, whereas in most programming situations the goal is usually generic. Learning programming therefore has two difficulties [Hoc90]. Firstly, there is “a shift from value to variable processing”. In other words, there is shift from thinking in concrete terms to thinking in abstract terms, and from dealing with specific to generic problems. Secondly, beginner programmers have to consciously transform natural procedures to machine procedures, and search for representations that they are not aware of in normal problem solving situations. The difficulty of programming therefore can be viewed as stemming from a huge gap between natural languages and programming languages [Hon90, Mye98, Pan01].

By contrast, much end-user programming research promotes programming at a concrete level. Examples are programming by demonstration [Cyp93] and spreadsheet programming [Nar93]. This is illustrated by considering the traditional spreadsheet. The advantages of spreadsheet programming are as follows [Nar93, Geh96]:

- it provides task-specific programming primitives that are already in the user’s problem domain. The primitives include a set of predefined functions that are commonly useful for the task at hand.
- it supports a trial and error style of programming. The underlying automatic dependency maintenance between cells allows the user to explore different settings quickly.
- it supports concrete display of data. The user does not have to worry about abstract data types.

All the above benefits make learning spreadsheet programming a lot easier than learning traditional programming. However, the major drawback of traditional spreadsheet programming is the lack of an abstraction mechanism. This makes reuse of a spreadsheet difficult. Some recent spreadsheet programming research are attempting to introduce abstraction mechanisms into spreadsheet programming. Examples are the structured spreadsheet Basset [Tuk96] and the visual programming language Forms/3 [Bur01]. However, these mechanisms are usually difficult to use. Figure 4.1 depicts problem solving at two different levels of abstraction. The conventional spreadsheet paradigm provides facilities for the user to solve a problem at a concrete level. It usually supports the user in considering a problem instance and a specific method to solve the problem. The goal is usually to obtain a specific solution to the problem. On the other hand, a traditional programming paradigm encourages the user to think abstractly. It supports the user in considering a problem class and in finding a generic method to solve them all. The goal is to obtain potential generic solution for all similar problems.



Concrete level	Consider a **problem instance** and a **specific procedure** to solve the problem. The goal is to obtain a **specific solution** for the problem.	Conventional Spreadsheet paradigm
Abstract level	Consider a **problem class** and a **generic method** to solve them all. The goal is to obtain **generic solution** for all similar problems.	Conventional programming paradigm

Figure 4.1: Programming for problem solving at two different levels of abstraction: concrete and abstract

However, the process of problem solving usually involves thinking in both concrete and abstract terms at the same time. Therefore, neither the spreadsheet nor a traditional programming paradigm gives comprehensive support for the natural demands of human problem solving. In the next section, building on pioneering research in problem solving [New72] and the psychology of programming [Hoc90], we shall investigate an EM approach to problem solving using the computer.

4.3 Empirical Modelling for problem solving

4.3.1 Construal of the problem solving situation (CPSS)

As we discussed in the last section, problem solving not only involves searching in multiple problem spaces but also thinking at different levels of abstraction. Figure 4.2

depicts our new understanding of human problem solving. For simplicity, the figure shows only two levels of abstraction: *concrete* and *abstract*. There are three problem spaces associated with each level. At the concrete level, they are the problem instance space, the method space and the specific solution space. At the abstract level, they are the problem class space, the generic method space and the generic solution space. Problem solving can be viewed as building up a construal of the problem solving situation as informed by the searching of all the problem spaces together with the problem solver's past knowledge and experience.

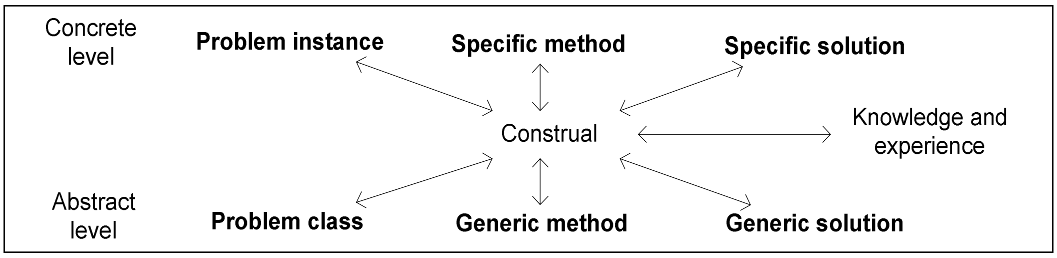


Figure 4.2: Problem solving as search in problem spaces at two different levels of abstraction

Martinez described human beings are "problem solvers who think and act within a grand complex of fuzzy and shifting goals and changing means to attain them" [Mar98b]. This implies that there is no predetermined order of search in the problem spaces. Usually problem solving starts off by a quick analysis of the problem. However, after the initial analysis, the problem solver can choose to search in any problem spaces according to their subjective judgment about what will make the best progress. A complete path from initially posing a problem to finding its solution involves arbitrary navigation between the problem spaces.

This new perspective motivates a shift from *supporting problem solving by programming* to *supporting problem solving by modelling*. With a conventional programming approach to problem solving, the role of the computer is mainly to provide means to specify and automate the procedural knowledge obtained by searching the method and generic method spaces. With a modelling approach to problem solving, the role of the computer is extended to provide means to represent the construal which consists not only of knowledge obtained from searching the method and generic method spaces but also the other problem spaces. By applying EM principles, we can construct an Interactive Situation Model (cf. chapter 2) of the problem solving situation. This model, which will be referred to as a Construal of the Problem Solving Situation (CPSS), embodies knowledge obtained by searching all the problem spaces. In a CPSS, knowledge is represented in terms of observation, dependency and agency. Integrated development and use of a CPSS can give powerful support for problem solving. In particular, in order to obtain a comprehensive solution, the problem solver needs to be able to account for all his or her partial knowledge of potential methods and solutions and how these are related to context and previous experience. The CPSS supports this kind of problem solving activity.

We shall illustrate the use of a CPSS with reference to a class of simple recreational problems, called Crossnumbers. Figure 4.3 shows a Crossnumber problem. The task is to place the numbers given in the list into the blank cells of the given 5 by 5 grid. Crossnumbers are adopted as a case study in this section because they are a convenient vehicle for illustrating the general principles behind CPSSs even if in practice a human problem solver would be unlikely to consider semi-automatic approaches to their solution.

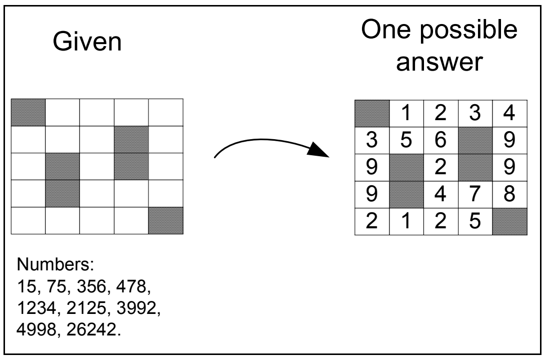


Figure 4.3: A Crossnumber problem with 5 x 5 grid (left) and its solution (right)

With reference to Figure 4.2, at the concrete level, the particular 5 by 5 Crossnumber problem is in problem instance space; any particular solution to solve the problem sits in the method space; whereas the solution shown on the right of the figure sits in the specific solution space. At the abstract level, the problem class space may contain all possible Crossnumber problems with grids of different sizes; search in generic method space represents development of generic program to solve all similar Crossnumber problems; and the generic solution space contains all possible solutions for similar Crossnumber problems.

Appendix D contains the complete listing of a CPSS for solving the Crossnumber problem. Solving problems like Crossnumbers involves free navigation of all spaces at different levels of abstraction. In putting the emphasis on semi-automatic human problem solving rather than fully automatic problem solving, we are motivated to consider well-recognised heuristics that can be used to guide problem solving. We shall show how the construction of a CPSS gives support in applying these heuristics.

4.3.2 Supporting heuristic problem solving

The process of problem solving involves understanding the problem, exploring the methods, and designing solutions to test the understanding of the problem and methods. The process can be guided by heuristics. Martinez describes a heuristic as “a strategy that is powerful and general, but not absolutely guaranteed to work” [Mar98b]. There are two kinds of heuristic: heuristics that are specific to a particular problem situation; and heuristics that are generic enough to apply in any problem solving context. We shall consider some generic heuristics discussed in the literature [Pól45, Hew95, Mar98b] and see how the CPSS of Crossnumber supports their application. They are External Representation, Problem Reinterpretation, Suspending Evaluation, Ends-means Analysis, and Successive Approximation.

External Representation. External Representation relates to finding ways to represent what is in our mind externally. By using an external representation, the problem solver can lay out complex information that cannot be considered internally in our mind at once [Mar98b]. It also forces the problem solver to clarify his or her thinking. For example, Pólya [Pól45] encourages the solver to draw a figure and to introduce a suitable notation in order to understand mathematical problems before solving them. External representation is sharable with other people who might try to consider the problem and might offer help. Good external representation renders visible what is actually in the problem solver’s mind. However, it is commonly agreed that it is not easy to achieve this in practice (e.g. [Hoc90, Pan01). The use of observables, dependencies, and agents can arguably help us to construct more natural external representations. By way of example, consider the external representation of the grid extracted from CPSS of the Crossnumber problem.

```
grid =   [['x',' ',' ',' ',' '],
          [' ',' ',' ','x',' '],
          [' ','x',' ','x',' '],
          [' ','x',' ',' ',' '],
          [' ',' ',' ',' ','x']
        ];

a1 is grid[1][1]; a2 is grid[1][2]; a3 is grid[1][3]; a4 is grid[1][4]; a5 is grid[1][5];
b1 is grid[2][1]; b2 is grid[2][2]; b3 is grid[2][3]; b4 is grid[2][4]; b5 is grid[2][5];
c1 is grid[3][1]; c2 is grid[3][2]; c3 is grid[3][3]; c4 is grid[3][4]; c5 is grid[3][5];
d1 is grid[4][1]; d2 is grid[4][2]; d3 is grid[4][3]; d4 is grid[4][4]; d5 is grid[4][5];
e1 is grid[5][1]; e2 is grid[5][2]; e3 is grid[5][3]; e4 is grid[5][4]; e5 is grid[5][5];
```

The two blocks of script above represent two different conceptualisations of the grid in the problem solver's mind. The first block is a list of lists. The second block defines each cell of the grid as a separate observable. Which representation is closer to the problem solver's conception depends on context. But by using definitive script, the problem solver can have many different representations at the same time. The consistency between representations can be automatically maintained. For example, in this case, values of the second representation are always consistent with the corresponding values in the first representation by definitions.

Suspending Evaluation. It is sometimes useful to temporarily stop evaluating the usefulness of the observations we introduced. This heuristic facilitates the discovery of new methods to solve a problem. For example, all the following definitions were initially conceived simply by observation of the grid structure with no specific purpose or goal in mind.

```
blocks  is [a1,b4,c2,c4,d2,e5];
position1 is [a2,a3,a4,a5];
position2 is [b1,b2,b3];
position3 is [d3,d4,d5];
position4 is [e1,e2,e3,e4];
position5 is [b1,c1,d1,e1];
position6 is [a2,b2];
position7 is [a3,b3,c3,d3,e3];
position8 is [d4,e4];
position9 is [a5,b5,c5,d5];
```

In a definitive script, introducing extra observations does not affect the existing observations. Therefore, there is no harm in introducing observations which may never be used to solve the problem. Eventually, such apparently redundant observations may lead us to a reinterpretation of the problem, such as will now be discussed.

Problem Reinterpretation. The way in which a problem is interpreted may affect the difficulty of solving the problem [Hew95]. Therefore, when we have found the problem is not as easy as expected, it is useful to reinterpret the problem in other ways. This is usually accompanied by a change of problem representation. In the case of the Crossnumber problem, the problem is transformed when we choose to represent the 9 positions where the given 9 numbers are to be filled in as follows:

```
numbers is ["1234","2125","26242","3992","4998","356","478","15","75"];
position1 is [a2,a3,a4,a5];
position2 is [b1,b2,b3];
position3 is [d3,d4,d5];
position4 is [e1,e2,e3,e4];
position5 is [b1,c1,d1,e1];
position6 is [a2,b2];
position7 is [a3,b3,c3,d3,e3];
position8 is [d4,e4];
position9 is [a5,b5,c5,d5];
```

The problem becomes: assign digits to cells so that each of the observables from `position1` to `position9` contains one of the numbers in the observable `numbers`. After this reinterpretation of the problem, the criteria for the correctness of the solution can be specified as follows:

```
ok1 is containsString(numbers, digitsToString(position1));
ok2 is containsString(numbers, digitsToString(position2));
ok3 is containsString(numbers, digitsToString(position3));
ok4 is containsString(numbers, digitsToString(position4));
ok5 is containsString(numbers, digitsToString(position5));
ok6 is containsString(numbers, digitsToString(position6));
ok7 is containsString(numbers, digitsToString(position7));
ok8 is containsString(numbers, digitsToString(position8));
ok9 is containsString(numbers, digitsToString(position9));
solved is ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8 && ok9;
```

The definitions `ok1` through `ok9` monitor the correctness of numbers filled in from `position1` to `position9` respectively. The overall correctness is defined by the definition `solved`. This can be regarded as a big step towards solving the problem because the goal is clearly represented by definitions. We can also monitor the status of any purported solution by querying the related definitions.

Ends-means Analysis. Ends-means Analysis involves “form[ing] a subgoal to reduce the discrepancy between your present state and your ultimate goal state” [Mar98b]. In other words, try to do something which seems to be making progress towards the ultimate goal state. This heuristic is useful when the problem is too complex for a solution to be found at once. In constructing the CPSS of Crossnumber, the introduction of the following definitions enables a semi-automatic way of solving the problem (‘achieving a subgoal’), which may also eventually lead to an automatic solution.

```
set1 is findNumberWithConstraints(numbers, digitsToString(position1));
set2 is findNumberWithConstraints(numbers, digitsToString(position2));
set3 is findNumberWithConstraints(numbers, digitsToString(position3));
set4 is findNumberWithConstraints(numbers, digitsToString(position4));
set5 is findNumberWithConstraints(numbers, digitsToString(position5));
set6 is findNumberWithConstraints(numbers, digitsToString(position6));
set7 is findNumberWithConstraints(numbers, digitsToString(position7));
set8 is findNumberWithConstraints(numbers, digitsToString(position8));
set9 is findNumberWithConstraints(numbers, digitsToString(position9));

stuck is set1#==0 || set2#==0 || set3#==0 || set4#==0 || set5#==0 || set6#==0 || set7#==0
|| set8#==0 || set9#==0;
```

The initial values of `set1` to `set9` are:

```
set1: ["1234","2125"]
set2: ["356","478"]
set3: ["356","478"]
```

```
set4: ["1234","1215","2992","4998"]
set5: ["1234","1215","2992","4998"]
set6: ["15","75"]
set7: ["26242"]
set8: ["15","75"]
set9: ["1234","1215","2992","4998"]
```

We can immediately observe that `set7` contains only one number `26242`. This means that only number `26242` can go into position 7. This is the number to be assigned into the grid first, which may further constrain the choices of numbers in other positions. One strategy may be to assign numbers to the positions where fewest choices are available. The observable `stuck` indicates when we should stop assigning numbers and backtrack to make an alternative choice.

Successive Approximation. Successive Approximation involves initially constructing a less than satisfactory solution and then making iterations to improve the solution until a satisfactory one is developed. The whole development and use of CPSS for Crossnumber problem can be viewed as Successive Approximation. Initially the support of CPSS is limited in terms of actually solving the problem. Over time, semi-automatic or even automatic methods for solving the problem may emerge from the CPSS.

4.3.3 Integrating development and use

Problem solving skills can be regarded as a form of tacit knowledge. Heuristics provide guidance for us in learning and applying these skills. Constructing a CPSS of a problem helps the problem solver to apply these skills. We have shown how the development and use of CPSS supports the application of a variety of generic problem solving heuristics. Being able to develop and use a CPSS at the same time is very important to its ability to help problem solving. There are three reasons. First, by integrating development and use, the problem solver can get immediate feedback on the progress in terms of the understanding of the problem, the reliability of proposed methods and the validity of the solution. The problem solver can directly take this information into account to improve the usefulness of CPSS. Second, the problem solver is more able to deal with the changes made to the requirement of the problem by constantly reviewing the situation of the problem. Finally, the separation of development and use is artificial. The interaction between development and use is the key to problem solving. CPSS guides a more natural way of human problem solving. CPSS is a medium to support thinking.

4.4 Case study: a timetabling problem

In this section, we describe an application of EM principles and tools to solve a real life timetabling problem. The problem involves scheduling the time and location of undergraduate final year project oral presentations in the Computer Science Department at Warwick University. A CPSS called the Temposcope was built to support the timetabling process. The name 'Temposcope' stands for 'An instrument for Timetabling with Empirical Modelling for Project Orals' [Bey00b]. A departmental administrator has been successfully using the Temposcope over the last two years as a support for solving the timetabling problem. This case study is a practical demonstration of the concepts of EM for problem solving discussed so far in previous sections of this chapter.

4.4.1 Timetabling problem for project oral presentations

The initial situations of this timetabling problem are briefly described as follows. There are about 125 students who are in the final year of two similar undergraduate courses, Computer Science (CS) and Computer and Business Studies (CBS). One of the requirements for their final year project is to have a project oral in the last week of the first semester. Each student's supervisor and an assessor from the members of staff are expected to attend the oral. The oral week lasts from Monday to Friday, 9 am to 6 pm. Each oral is allocated a 40-minute timetable slot. Each day has 13 timetable slots available. Each oral is held in one of up to 5 departmental rooms that are available. The Temposcope is used to support timetabling process which was previously being done by hand with pen and paper, and took about a whole week. The problem involves assigning projects to slots while taking account of the availability of staff, students and rooms.

Solving timetabling problems on this scale is usually non-trivial for either a human timetabler or a computer program. A huge number of choices and constraints make it difficult for a person to take all factors into account at once. Determining whether there is a feasible solution of a timetabling problem is a well-known NP-complete problem [Gar79]. Conventional techniques for computer-supported timetabling usually involve extensive searches of a huge solution space that involves evaluating an assigned weight or penalty to each decision made according to constraints [Cor94]. To construct the timetable reasonably efficiently, to achieve a good quality result and maintain flexibility, a high degree of co-operation between human and computer is required. Research on computer-based support for solving timetabling problems are no longer solely about finding optimum computer algorithms but more about supporting human problem solving as a process that consists of a number of activities such as data capturing, data modelling, data matching, report generation and storage of timetabling results [Opt00, Sch00b].

EM principles support both the development and use of the Temposcope but do not impose the order of them. This makes the Temposcope highly adaptable to change of situations. By integrating development and use, the quality of the Temposcope is constantly improving. As the timetabler’s knowledge and experience increases, better strategies may be found to produce quality timetables in a shorter period of time. In the next subsection, we shall describe how the development and use of the Temposcope helps to solve the problem.

4.4.2 Integrated development and use of the Temposcope

Construction of the timetabling EM model, Temposcope, started off from a state-based analysis of the problem in terms of observable, dependency and agency. It is an informal analysis of the particular problem at hand as perceived by the problem solver. This includes identifying key observables of the problem and finding a representation for them. For example, the basic observation might be: there are 5 rooms, 2 staff with their availability, and 2 students with their final projects. They can be represented by the following definitions:

```
room = ["104", "327", "313", "LL1", "444"];
WMB_AV = [1,2,3,7,8];
SBR_AV = [7,8,9,10,11,12];
data1 = ["Al-Khaburi", "Ali", "How secure is a secure website? ", "CBS", "DA", "AB",
"SBR"];
data2 = [" Andand", "Aradhana", "Impact & utilisation of the internet in Indian companies",
"CBS", "YM", "MCK", "AB"];
```

Choosing the right representation is very important to problem solving (as mentioned in previous section). Representing a problem in one way might make the problem easier to solve than representing in another way. One important feature of representing a problem using definitions is that the problem solver has freedom to specify different representations within the model. Different representations can exist together in the model without introducing problems of inconsistency. This is because dependencies between the data and representation are automatically maintained. For example, there are two different styles to represent staff availability in the Temposcope:

```
WMB_AV is [1,2,3,7,8];
WMB_AVBinary is makeBinary(WMB_AV); // the value would be [1,1,1,0,0,0,1,1]
```

The above two definitions demonstrate the co-existence of different representations of the same observable. The consistency between two representations of the availability of a particular staff WMB is maintained.

Also note that the observables introduced so far are at the concrete level specific to the current situation of the problem. For example, `WMB_AV`, `SBR_AV`, `DA_AV` are observations of the availability specific to three staff. This may be considered to be poor style for conventional programming paradigms, where the emphasis is on generality and abstract variables are used. However, the ability to specify observables at the concrete level (or ‘concrete variables’) in the Temposcope empowers the problem solver to be more engaged with the situation of the specific problem. The flexibility of representation also has a profound effect on mediating the interaction between human and computer. One example is the definition:

```
staffAV is [WMB_AV, SBR_AV, DA_AV…];
```

This definition relates to observation at a more abstract level than the previous ones. `staffAV` represents a list of availability for all staff. This supports different kinds of agency: facilitating the computer to iterate through the availabilities of staff, whilst also supplying a specific definition of the availability of each staff member that is more understandable by the human problem solver.

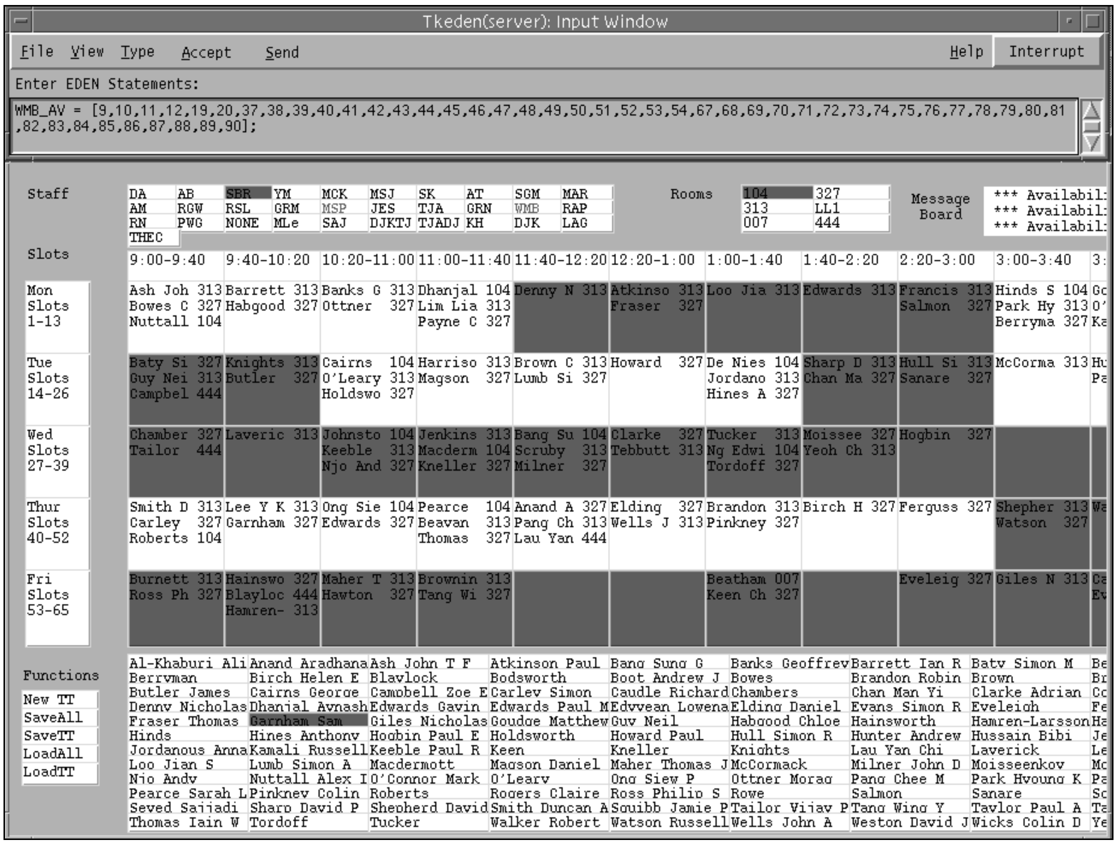
In the Temposcope, dependencies are specified:

```
// example definition showing explicit dependencies here!
class is makeclass(data);
ataff is makestafflist(data, [5,6,7]);
AVSTAFF is makeAVSTAFF(staff, avail);
avx is map(proj2_1, [avstud], class);
// definitions showing some joint observations here:
DJKTJA_AV is union(DJK_AV, TJA_AV);
```

When compared with a conventional programming paradigm, where dependencies are implicitly scattered around the procedural code, explicit dependency specification helps to make the model more flexible to change and comprehensible in use.

The definitions introduced so far are all intrinsic to the problem – they are not associated with any specific method for solving the timetabling problem. The emphasis on state-based rather than behavioural-based analysis enables the problem solver to take richer observables into account with deeper insight into the dependencies amongst them. This helps the problem solver to gain more insight into the problem, with which eventually strategies or even efficient algorithms may emerge. However, even without further extending the model, the problem solver can already use the model to support decision making in timetabling. For example, the joint availability of staff can be easily obtained. Subsequent enhancement of the model will make the model more useful, but even at present it is already usable (cf. the Successive Approximation heuristic).

Recall that one of the heuristics we have discussed is External Representation. The script itself, as an interactive textual artefact, can be regarded as a form of external representation of our understanding of the problem. In addition, the Temposcope incorporates visual representations to help the problem solver to make decisions in the process of development and use. Figure 4.4 is a screen capture of the visualisation of the Temposcope.



Enter EDEN Statements:

```
WMB_AV = [9,10,11,12,19,20,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81
,82,83,84,85,86,87,88,89,90];
```

Staff

DA	AB	SBR	YM	MCK	MSJ	SK	AT	SGM	MAR
AM	RGW	RSL	GRM	MSP	JES	TJA	GRN	WMB	RAP
RN	PWG	NONE	MLe	SAJ	DJKTJ	TJADJ	KH	DJK	LAG
THEC									

Rooms

104	327
313	LL1
007	444

Message Board

*** Availabil:
*** Availabil:
*** Availabil:

Slots	9:00-9:40	9:40-10:20	10:20-11:00	11:00-11:40	11:40-12:20	12:20-1:00	1:00-1:40	1:40-2:20	2:20-3:00	3:00-3:40	3:
Mon Slots 1-13	Ash Joh 313 Bowes C 327 Nuttall 104	Barrett 313 Habgood 327	Banks G 313 Ottner 327	Dhanjal 104 Lim Lia 313 Payne C 327	Denny N 313	Atkinso 313 Fraser 327	Loo Jia 313	Edwards 313	Francis 313 Salmon 327	Hinds S 104 Park Hy 313 Berryma 327	Gc O’ Ka
Tue Slots 14-26	Baty Si 327 Guy Nei 313 Campbel 444	Knights 313 Butler 327	Cairns 104 O’Leary 313 Holdswo 327	Harriso 313 Magson 327	Brown C 313 Lumb Si 327	Howard 327	De Nies 104 Jordano 313 Hines A 327	Sharp D 313 Chan Ma 327	Hull Si 313 Sanare 327	McCorma 313	Hu Pa
Wed Slots 27-39	Chamber 327 Tailor 444	Laveric 313	Johnsto 104 Keeble 313 Njo And 327	Jenkins 313 Macderm 104 Kneller 327	Bang Su 104 Scruby 313 Milner 327	Clarke 327 Tebbutt 313	Tucker 313 Ng Edwi 104 Tordoff 327	Moissee 327 Yeoh Ch 313	Hogbin 327		
Thur Slots 40-52	Smith D 313 Carley 327 Roberts 104	Lee Y K 313 Garnham 327	Ong Sie 104 Edwards 327	Pearce 104 Beavan 313 Thomas 327	Anand A 327 Pang Ch 313 Lau Yan 444	Elding 327 Wells J 313	Brandon 313 Pinkney 327	Birch H 327	Ferguss 327	Shepher 313 Watson 327	Wa
Fri Slots 53-65	Burnett 313 Ross Ph 327	Hainswo 327 Blayloc 444 Hamren- 313	Maher T 313 Hawton 327	Brownin 313 Tang Wi 327			Beatham 007 Keen Ch 327		Eveleig 327	Giles N 313	Ca Ev

Al-Khaburi Ali	Anand Aradhana	Ash John T F	Atkinson Paul	Bang Sung G	Banks Geoffrey	Barrett Ian R	Baty Simon M	Be
Berryman	Birch Helen E	Blaylock	Bodsworth	Boot Andrew J	Bowes	Brandon Robin	Brown	Br
Butler James	Cairns George	Campbell Zoe E	Carley Simon	Caudle Richard	Chambers	Chan Man Yi	Clarke Adrian	Cc
Denny Nicholas	Dhanjal Avnash	Edwards Gavin	Edwards Paul M	Edyvean Lowena	Elding Daniel	Evans Simon R	Eveleigh	Fe
Fraser Thomas	Garnham Sam	Giles Nicholas	Goudge Matthew	Guy Neil	Habgood Chloe	Hainsworth	Hamren-Larsson	Ha
Hinds	Hines Anthony	Hogbin Paul E	Holdsworth	Howard Paul	Hull Simon R	Hunter Andrew	Hussain Bibi	Je
Jordanous Anna	Kamali Russell	Keeble Paul R	Keen	Kneller	Knights	Lau Yan Chi	Laverick	Le
Loo Jian S	Lumb Simon A	Macdermott	Magson Daniel	Maher Thomas J	McCormack	Milner John D	Moisseenkov	Mc
Njo Andy	Nuttall Alex I	O’Connor Mark	O’Leary	Ong Siew P	Ottner Morag	Pang Chee M	Park Hyoung K	Pa
Pearce Sarah L	Pinkney Colin	Roberts	Rogers Claire	Ross Philip S	Rowe	Salmon	Sanare	Sc
Seyed Sajjadi	Sharp David P	Shepherd David	Smith Duncan A	Squibb Jamie P	Tailor Vijay P	Tang Wing Y	Taylor Paul A	Ta
Thomas Iain W	Tordoff	Tucker	Walker Robert	Watson Russell	Wells John A	Weston David J	Wicks Colin D	Ye

Figure 4.4: A screen capture of the Temposcope

The visualisation shown in the Figure 4.4 was developed incrementally throughout the development and use of the Temposcope. Its development involves defining hundreds of Scout windows on-the-fly, which is a tedious job if every window has to be defined manually. This task was assisted by using the mechanism of virtual agents in DTkEden (a distributed version of TkEden [Sun99a]). Scout supports building up visualisation and interface quickly without possibly losing track of the problem. The visualisation also reflects the conception of the final timetable. This enables the problem solver to 'work backwards' by modelling a virtual solution for the problem.

As the modelling of a variety of aspects of the Temposcope goes on interactively, the problem solver's knowledge about the problem is gradually increased. This knowledge enables the problem solver to develop possible methods to tackle the problem in an incremental way. Immediate feedback given by the Temposcope helps the problem solver to assess whether he or she is heading in the right direction, and enables the application of the Means-ends Analysis heuristic. The initial strategy employed was "place-and-seek" [Pae94]. This involves assigning a project to a slot and checking whether there is conflict or not. If there is a conflict, other slots will be tried until the conflict is resolved. Even with this simple strategy, the Temposcope helps the problem solver to make decisions by automatically checking for conflict on-the-fly.

As we gather experience in using the Temposcope, we can develop more advanced strategies. One strategy is to count the number of possible slots for each project after each assignment of slots. This statistic can be used to find out the most tightly constrained projects. The timetabler tries to schedule these projects first before the others. Counting the number of possible slots for more than one hundred projects is a tedious job for human but the computer can do it quickly and accurately. Human-computer co-operation makes solving the timetabling problem easier than solving by either human or computer alone.

Another important feature of the Temposcope is its openness to the problem solver: it provides free access to all its definitions at any time without restrictions. This facilitates integrated development and use that enables the quality of the Temposcope to be improved over time with possible adaptation to new situations.

4.4.3 Distinctive qualities of the Temposcope

The distinctive qualities of the Temposcope stem from its integrated development and use, based on EM concepts of observation, dependency and agency. They can be summarised as: support for a heuristic approach to problem solving; comprehensive problem analysis; adaptation and extension to a dynamic problem situation; and deeper engagement of human agents with the computer.

Support for a heuristic approach to problem solving

In the last subsection, we have shown that the Temposcope facilitates the use of a variety of human problem solving heuristics that include Means-ends Analysis, Successive Approximation and External Representation. The freedom to adopt any approach and heuristic at any time in problem solving is essential for solving problems whose solutions are uncertain.

The Temposcope also enables more flexible human-computer co-operation than conventional timetabling applications. The problem solver can conduct exploratory experiments. The computer automatically maintains dependencies between observables that are central to giving immediate feedback from the experiments. The process of conducting experiments resembles the process of making a scientific discovery [Kla88], in which people make hypotheses and test them by conducting experiments. As a result of experiment, the hypotheses are validated or invalidated. Based on the new results, new hypotheses can be made which in turn direct new experiments. In particular, the Temposcope represents a construal in which provisional knowledge about the project timetabling problem is embodied and ready to be refined. The results of experiment may eventually lead to the discovery of better methods to solve the timetabling problem.

Support for comprehensive problem analysis

In the solution of a timetabling problem, there are two aspects to be considered. On the one hand, there are generic algorithms and strategies for timetabling that are not specifically related to the particular problem. On the other hand, we need to develop a rich understanding of the specific timetabling situation. In a traditional timetabling program, the abstract algorithms and strategies are built into the program itself. This means that they are determined by the developer in isolation from the problem solving context. The timetabler’s task is to make the best possible use of the given timetabling mechanisms to tackle the specific problem.

In using the Temposcope, it is possible to consider both the development of a timetabling method and the exploration of the specific timetabling problem in one and the same environment. In other words, timetabling using the Temposcope can be regarded as an integrated study of the timetabling problem and methods for its solution. In principle, this can lead to a deeper understanding of every aspect of the timetabling problem, out of which strategies for the best use of the power of both human and computer may emerge.

It should be noted that the integration of development and use envisaged here is more intimate than can be achieved simply by interleaving conventional development and use. By way of analogy, allowing arbitrary interleaving of conventional development and use is similar to allowing a car driver to return her car to the factory to be redesigned whenever she encounters unexpected problematic road conditions. In EM, there is no ontological distinction between the states in which development and use take place; it is as if the car driver can invoke car redesign in the context in which problematic conditions arise. This is significant for a variety of reasons: because (in the traditional redesign scenario) problematic road conditions may not be reproducible, making testing difficult; because there is a semantic issue concerning whether the redesigned car can return to 'the same context'; because it is hard for the driver to communicate her experience in a remote situation to the designer at the factory.

As the above analogy suggests, EM involves a blurring of the roles of designer and user. This means that, when using the Temposcope, the timetabler can not only conduct problem analysis in systematic ways but also in *ad hoc* ways. In particular, the problem solver can change definitions in the Temposcope and observe the consequences. This helps the problem solver to comprehend the full implications of dependencies in the model.

The definitions introduced as the result of problem analysis are intrinsic to the problem. We can regard them together as a model of the problem from which new strategies of solving the problem can be tested.

Support for adaptation and extension to a dynamic problem situation

Conventional timetabling applications may have algorithms that are optimised to solve a particular timetabling problem. The flexibility of these applications is limited to the situations that can be conceived during the development process. Timetabling problems like the one we have discussed are dynamic in nature. The requirements and situations are changing according to the external environment in unpredictable ways. The observables of the Temposcope are direct reflections of the observables in the external environment. Changes in the external environment can be directly related to a need for change in the model. The problem solver's job is to maintain the relationships between the model and the external environment by modifying existing definitions or adding new definitions.

Since adding new observables to the Temposcope does not affect existing observables in general, new modes of observation can be introduced independently of existing observations. If new observations are based on existing observation, their dependencies are automatically maintained. Redefinition of an observable is also easy with automatic update of other observables that are dependent on it. All these features make extension to the model relatively easy. And, most importantly, new extensions can be tested quickly to allow a proper evaluation of their usefulness. The Temposcope has been already extended to allow online submission of staff availability and to estimate workload for each staff (workload weighting).

Support for deeper engagement between human and computer

Problem solving is related to subjective cognitive processes that originate from the problem solver's intuition, knowledge and experience. Thinking in terms of observables, dependencies and agents is arguably more natural than thinking in variables, procedures and functions. The gap between our natural language and machine language can be kept to a minimum. The problem solver can be more engaged with the problem at hand rather than worry about language translations.

The Temposcope allows the incorporation of subjective views of the problem solver. These views may reflect unresolved issues and unpredictable circumstances. They are provisional knowledge about the problem and its solution. Extensive interaction with the model may convert this provisional personal knowledge into more stable knowledge that can be shared with others.

4.5 Summary

In this chapter, we have explored the use of the computer to support human problem solving. Based on research on general problem solving and the psychology of programming, we have proposed a better explanation of the phenomenon of human problem solving using a computer that emphasises cooperation at two levels of abstraction in searching many different problem spaces. We have discussed the difficulties of using a conventional programming paradigm to support problem solving. We have explained how integrated development and use of a Construal of the Problem Solving Situation (CPSS) can give powerful support for problem solving. In particular, we have shown how a CPSS supports the application of well recognised problem solving heuristics with reference to a simple example: the Crossnumber problem, and to a real life application: the use of the Temposcope.

5 Before Systems: Conceptual Integrity

In his famous book, The Mythical Man-Month [Bro95], Brooks contends that "conceptual integrity is the most important consideration in system design". This chapter endorses and elaborates this idea, and contends that obtaining conceptual integrity is essential before a coherent system conception can be established. We believe that, in successful system design, conceptual integrity emerges from activities that are prior to system identification. Along with the discussion we shall explain how the EM perspective on system development helps to address the issues of obtaining conceptual integrity.

This chapter is organised as follows. Section 5.1 explores the meaning of conceptual integrity. Section 5.2 discusses the importance of conceptual integrity in system development, and identifies issues that arise in maintaining the conceptual integrity of a system design. Section 5.3 describes how EM can help system developers to maintain conceptual integrity by addressing the issues identified in section 5.2. Section 5.4 illustrates the discussion with reference to a TkEden model of a railway. Section 5.5 compares EM with other technologies that aim to help system developers to maintain the conceptual integrity of a system design.

5.1 What is conceptual integrity?

In the context of system development, the term 'conceptual integrity' was first introduced by Brooks in the 1975 edition of his book The Mythical Man-Month [Bro95]. Drawing on much experience of system development, Brooks contends that:

> *"Conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas…Conceptual integrity in turn dictates that the design must proceed from one mind, or from a very small number of agreeing*

resonant minds." [Bro95]

Our discussion in this chapter aims to endorse and elaborate the importance of conceptual integrity in the design of systems, and offers EM to help developers to maintain conceptual integrity in a system design.

Brooks associates 'conceptual integrity' with the 'unity of design' but offers no further satisfactory explanation on what conceptual integrity is. To appreciate the importance of conceptual integrity, we need to take a closer look at its meaning. The word 'conceptual' is associated with the cognitive process of concept forming that involves the conscious recognition and identification of elements of our experience; and the word 'integrity' is associated with the idea of 'being integrated' or 'being one'. The idea of a coherent whole is reflected in the way that 'having integrity' is used to describe something that always conforms to one's expectations – there is an implicit reference to future events. We describe something as having conceptual integrity, if the concepts formed from our *experience* of the thing can reliably determine the future events which are associated with the thing (there is no surprise). In *obtaining* conceptual integrity, we are concerned with the emergence of concepts (representing the rational world) from experience (representing the empirical world). As the American philosopher, William James, points out in his Essays on Radical Empiricism, first published in 1902:

"Experiences come on an enormous scale, and if we take them all together, they come in a chaos of incommensurable relations that we cannot straighten out. We have to abstract different groups of them and handle these separately if we are to talk of them at all." [Jam96]

Raw experiences are potentially confusing or incoherent as "they come in a chaos of incommensurable relations". To make sense of them, we need to abstract groups of relations between them and study them separately. Abstracting groups of relations from experiences is the embryonic concept-forming activity. The concepts formed may conflict each other. Obtaining conceptual integrity is a process of removing conflict. To remove conflict, we need to access the raw experiences that originally informed our concepts – to explore them, to understand, to compare and to analyse. This idea is depicted in Figure 5.1. At the left-hand-side of the figure, two groups of concepts are formed by abstracting relations from raw experiences. To obtain conceptual integrity, we need to combine the concepts into one group of unified concepts. This involves the resolution of potential conflict between concepts so that they acquire coherence.

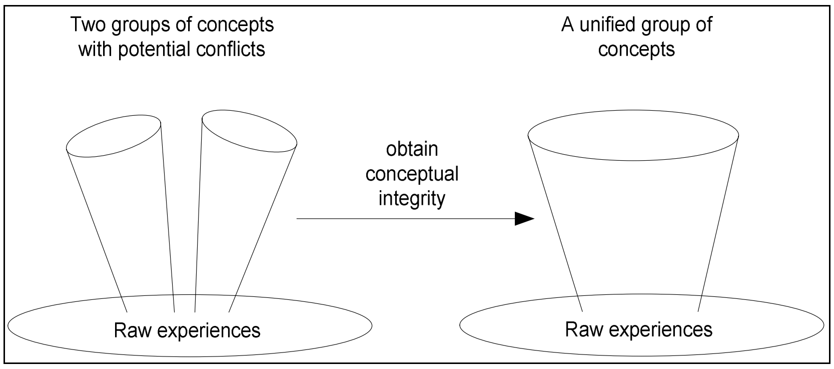

Two groups of concepts with potential conflicts
A unified group of concepts
obtain conceptual integrity
Raw experiences
Raw experiences

Figure 5.1: Obtaining conceptual integrity from raw experiences

Significant features of the above observations that relate to the nature of conceptual integrity include:

- Conceptual integrity is something appreciated by an observer. This is because we cannot talk about experience without reference to the observer.
- To obtain conceptual integrity, we need to resolve conflicts between concepts. This involves access to the raw experiences that inform these concepts.
- A central issue in obtaining conceptual integrity is the potential incoherence of raw experiences.

The process of obtaining conceptual integrity is closely associated with heuristic cognitive activities that are hard to formalize (cf. Naur's characterisation of a science, which regards "coherent description as the core of the scientific/scholarly acitivity" [Nau01]). Since, as mentioned in the last chapter, EM has the potential to support heuristic problem solving, we believe that EM can help in obtaining conceptual integrity. An illustration of this process can be found in section 5.4

5.2 Conceptual integrity of system design

System development is located at the intersection of formal and informal, objective and subjective, and technical and non-technical activities (cf. [Sun99a, Wes97]). It is difficult for developers to maintain the conceptual integrity of the system design. By 'system design', we mean the current state of the design of the system under development. A system design evolves throughout the system development process. Changes to the system design are usually made concurrently by different members in the system development team. This makes maintaining conceptual integrity of the system design even more difficult. But conceptual integrity is surely necessary because it is essential for:

- innovation – with deep understanding of the current state of the system design, the developers can relate concepts in the system design more easily to new ideas;
- flexible system design – a system design with conceptual integrity can be more easily adapted to possible change of requirements;
- effective project management – better knowledge about the status of the design makes it easier to adjust available resources.

Conceptual integrity is of the essence in system development in that what lacks conceptual integrity cannot be easily perceived as a coherent system. The difficulty of maintaining conceptual integrity can be understood in relation to a tension involved in system development depicted in Figure 5.2 below. This tension exists between distinguishing and synthesising different viewpoints at both personal and interpersonal levels.

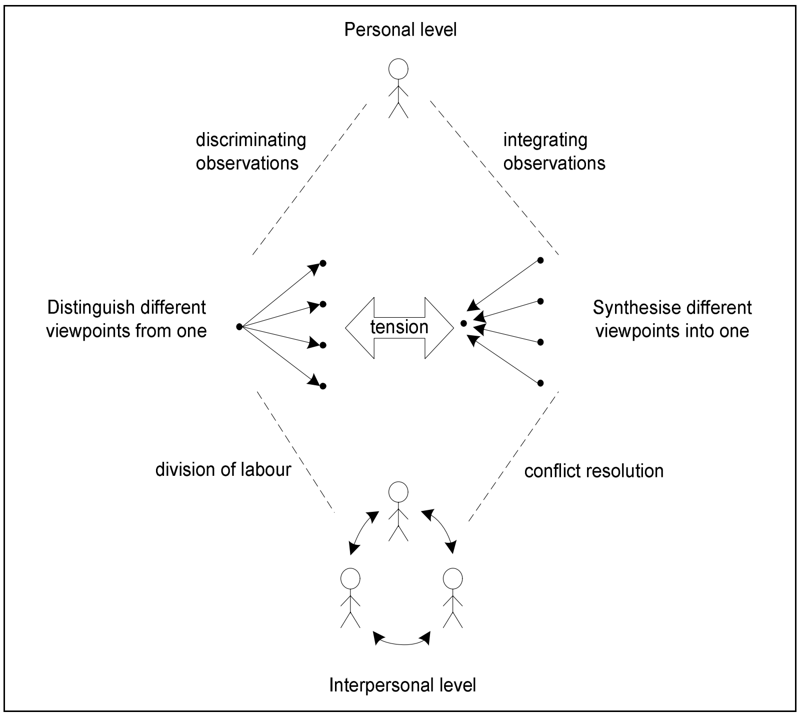

Personal level
discriminating observations
integrating observations
Distinguish different viewpoints from one
tension
Synthesise different viewpoints into one
division of labour
conflict resolution
Interpersonal level

Figure 5.2: A major tension in system development

At a personal level, distinguishing viewpoints involves discriminating different observations from apparently the same experience; synthesising viewpoints involves integrating different observations as a unified experience. At the interpersonal level, distinguishing viewpoints is the key to the division of labour; and synthesising viewpoints involves conflict resolution. The tension between distinguishing and synthesising viewpoints always exists in system development – we want to distinguish viewpoints on an experience but at the same time see them as a unified whole. The potentially incoherent experience that originates from trying to reconcile these viewpoints is the main obstacle to obtaining conceptual integrity. The key questions that need to be answered in relation to obtaining conceptual integrity in system design are:

At a personal level,
- How can we represent experience that is potentially incoherent?
- How can we effectively abstract concepts from experience?
- How can we resolve inconsistency in concepts?

At the interpersonal level,
- How can we ‘share’ experience and concepts?
- How can we resolve conflicts and reach consensus?
- How can we represent alternative views?

These are difficult questions to answer because, as mentioned above, the process of obtaining conceptual integrity is hard to formalise. For this reason, conventional formal methodologies have limited applicability. To address the questions, we need to have a human-centred approach to system development that emphasises the creative nature of the cognitive process. We believe that the principles and techniques of EM offer a plausible solution that can satisfactorily address these questions.

5.3 EM for maintaining conceptual integrity

The central idea applying EM to help developers to maintain conceptual integrity in system development is to construct Interactive Situation Models (ISMs). ISMs can be used to represent situated experience acquired through a variety of system development activities. An ISM serves a role in knowledge representation in the sense that ‘one experience knows another’ [Bey99]. In introducing the concept of an ISM, Beynon and Sun [Bey99] cite writings on philosophy [Jam96], linguistics [Tur96] and experimental science [Goo00] that reflect the importance of negotiating meaning through interaction with artifacts where processes of explanation and knowledge creation are concerned. They point out that “[w]here formalisms aim at freedom from ambiguity and independence of agency and context, experiential representations rely essentially upon the engagement of the human interpreter”. This highlights the limited role that formalisms can play in system development. The EM approach to overcoming this limitation is to use an ISM to embody experience (potentially incoherent) that human interpreters can abstract and share by interacting with it, and that may eventually lead to conceptual integrity in a system design.

In EM, there is a direct correspondence between observables, dependencies and agents in the external world and variables, definitions and actions in an ISM. It is this faithful metaphorical representation of real world entities that makes an ISM a powerful representation of what we observe and experience. Specifying variables, definitions and actions in an ISM is *not* primarily a rational process – because the modeller does not have to give a logical justification for their existence in the model – rather it is an empirical process in which observations are faithfully recorded. On this basis, even incoherent experience can be embodied in an ISM. By analogy, an ISM resembles a draft sketch for a painting (cf. [Ras01]). Its purpose is primarily concerned with *forming concepts* rather than *specifying concepts* –not all the lines in the sketch will eventually remain in the final painting; nevertheless, every line in the initial sketch contributes – even if only indirectly – to the final painting at some stage in the painting process. EM is concerned with identifying and exploring relevant observations prior to formulating concepts from which the system emerges. By its nature, this process of identification demands that we explore things that are eventually deemed to be outside the final system.

As mentioned earlier, James’s prescription for making sense of experience is “to abstract different groups of [relations] and handle these separately”. In EM, activity of this nature can be performed by extracting groups of definitions from an ISM and studying them separately. An effective way to study concepts is to experiment with them in the manner that is similar to the use of a spreadsheet. A typical use of a spreadsheet involves correlating the real world situation with the state captured in the spreadsheet cells. For instance, a spreadsheet can represent the financial situation of a company. A change in the value of a cell may reflect a change in the real world situation of the company. An accountant can perform many ‘what-if’ type of experiments to explore a variety of situations in which the company will be affected. By interacting with a spreadsheet, one can obtain experience that reflects the experience of changing the real world financial situation of the company. Based on this experience, one can analyse existing concepts and form new concepts. Using an ISM resembles using a spreadsheet in this respect, but provides more general functionalities that are suitable for studying concepts in potentially any domain. For this reason, an ISM is a test-bed within which the concepts of a system design can be studied, which is essential for obtaining conceptual integrity of the system design.

Since an ISM embodies experience and concepts, a developer can share experience and concepts with others by sharing an ISM. Unlike a static document, an ISM can be shared and studied by others interactively. For this reason, an ISM is like a concrete physical prototype of a product which one can study by directly experiencing it rather than merely in an abstract fashion.

Where collaborative work is concerned, the collaborative definitive modelling in the DMF provides an agent hierarchy structure within which ISMs can be shared and conflicts can be resolved. The detail of the structure and mechanisms involved has been already discussed in chapter 2, and chapter 3 illustrates the idea with reference to the design of a lathe spindle.

To show that the EM perspective on system development is highly relevant to the conceptual integrity of a system design, we can reinterpret the concerns of system development discussed in chapter 3 in terms of conceptual integrity.

- Complexity – In system development, a system design will be expressed with reference to many different ideas and viewpoints. The viewpoints are not necessarily or typically consistent when taken in conjunction. In other words, they all come together with such a degree of complexity that is difficult to make sense of them. If we are to achieve conceptual integrity, we need to be able to represent complex experiences of this sort that lack conceptual integrity. At present, we ‘represent’ such experiences in a rather incoherent way using documentation in both natural and formal languages, possibly supported by prototypes, and manage sense making through documented meetings between developers. If we cannot represent such experiences effectively, we cannot begin to resolve the conflicts that inhibit sense making and are tempted to resort to simplifying compromises. By using EM, we aim to achieve conceptual integrity of a system design *without compromising complexity*. There is evidence that achieving conceptual integrity without compromising complexity is possible. For instance, Brooks [Bro95] cites the cathedral at Rheims as an example of a complex structure having conceptual integrity.

- Predictability – The perception of conceptual integrity is associated with confirmation of expectation. Things have conceptual integrity if they do not surprise us, in the sense that what we know about the system is a reasonable guide to what we do not know. It should be noted that this predictability, that is fundamental to conceptual integrity, manifests itself in observation and interaction. The fact that all concepts are expressed by using a set of common modelling abstractions (as e.g. in the proposals for 'seamless' software system development discussed in [Lid94, Pai01]) does not necessarily mean that a system will exhibit conceptual integrity. For example, representing a set of programs using a functional programming paradigm arguably does not guarantee their conceptual integrity. Functional programming is well-suited to the representation of simple programs whose entire significance is captured in their input-output relation. It cannot easily meet the need to embrace other viewpoints on programs, such as arise where complex interaction or visualisation is involved. For this reason, where programming is concerned, the use of the functional paradigm entails trading complexity for conceptual integrity.

- Unity – The coherence of things that have conceptual integrity means that we can experience them as 'being one thing'. Grouping observables into objects as in OOA is one way to try to bring unity to a complex phenomenon, but the coherence and integrity of an object is achieved by extracting it and viewing in isolation from a single viewpoint. On the other hand, the coherence needed for conceptual integrity in a system design needs to embrace many viewpoints. This is similar to the situation in everyday experience, where the integrity of observables is not simply confined to object associations but extends to more subtle dependencies. As explained and illustrated with reference to modelling a door in subsection 4.2.2 of Rungrattanaubol's thesis [Run02], "in modelling with definitive scripts, dependency is the feature that gives integrity to [the] diverse representations of a 'single' object". The key feature of modelling with definitive scripts is that it allows us to add observables within conceptually the same state. In other words, adding a definition to the script need not mean changing the state to which it refers, but rather enriching our perception of the state to which it refers. With such use of definitions, dependencies between observables are seen to be the mediators of unity, expressing 'what belongs to what'.

- Cognitive aspect – In the convolving generation of knowledge, systematic elements emerge. These elements sometimes make sense individually but are incoherent when combined. Potentially, subject to suitable compromises and shifts in perspective, all the requisite systematic elements can be made coherent (cf. Naur's idea of science as 'coherent description of aspects of the world'

[Nau01]). Conceptual integrity emerges in system development through bringing coherence to groups of systematic elements taken together – as an incremental and evolving process.

- Collaborative aspect – Even when each individual developer's view has integrity, we still need to bring them together to realize a coherent overall design in the spirit of Brooks's 'one mind'. The DMF provides collaborative definitive modelling in which a hierarchy of agents/developers interact with each other by sharing ISMs to achieve a coherent and consistent representation.

- Methodological aspect – Conceptual integrity cannot be imposed by formal methods or business processes. Obtaining conceptual integrity is a heuristic cognitive process. EM provides an observation-led approach that is not associated with preconceived ideas about how the observables in the domain should be organized. For instance, whereas OOA obliges the modeller to identify and maintain object boundaries throughout the system development process, EM does not favour any specific whole-part decomposition of a system under development unless or until one emerges. The motivation for such decomposition may come from the properties of the domain ('thinking about the system') or from the demands of distributed development ('thinking about system development activities'). EM does not separate the phases of the development activities according to a preconceived pattern. This promotes conceptual integrity through working practices that are in sharp contrast to a classical development process, where the contributions made by separate design participants at the various stages are largely independent.

We summarise this section by offering answers to the questions asked at the end of section 5.2:

At a personal level,

- How can we represent experience that is potentially incoherent? – by using ISMs raw experience can be faithfully represented. EM is not primarily concerned with modelling a system but facilitating construals of situations in the environment from which the system emerges.
- How can we effectively abstract concepts from experience? – in EM, observation, interaction and experiment are the key to the development of concepts.
- How can we resolve inconsistency in concepts? – by extensive interaction and

experiment with the model, inconsistency can be resolved.

At the interpersonal level,

- How can we ‘share’ experience and concepts? – an ISM resembles a concrete physical prototype in that it can be shared amongst developers, but offers a far richer level of conceptual support through interaction than a conventional prototype. By interacting with a shared ISM, developers can share experience and concept.
- How can we represent alternative views? – the modeller is able to add observables to an ISM without changing its conceptual state. Observables or groups of observables representing alternative views can be linked together with underlying dependencies.
- How can we resolve conflicts and reach consensus? – the hierarchical collaborative definitive modelling structure in the DMF helps developers to resolve conflict and reach consensus in a manageable way.

In short, EM aims at enhancing the scope for interaction in system development by allowing experience and concepts to be embodied in computer-based models that can be shared, explored and discussed dynamically to facilitate the maintenance of conceptual integrity in a system design.

5.4 Case study: the Railway model

In this section, we shall describe the Railway model and use it to illustrate the ideas discussed in earlier sections. The model was developed by Y. P. Yung in the EM research group, and has been used to illustrate various aspects of EM research in other contexts [Bey90b, Adz94a, Adz94c]. It is a TkEden model with scripts of a variety of definitive notations. The model contains a number of submodels which will be described one by one as the discussion develops.

The Railway model contains a model for designing track layout. Figure 5.3 shows the visualisation of the track layout model. The track segments are arranged to form two circuits one inside the other. The two circuits are connected by the pair of sets of points located at the top of the figure. Although the track layout is simple, the definitions for track segments are based on standard track segments from a track catalogue - so that the length and curvature of each piece is based on a standard specification. Listing 5.1 shows an extract from Donald definitions that defines one of the segments in the track layout. The openshape ST226 defines a standard prototypical segment with the product code ST226. The openshape A9 defines a segment on the track that is based on a transformation of the segment ST226.

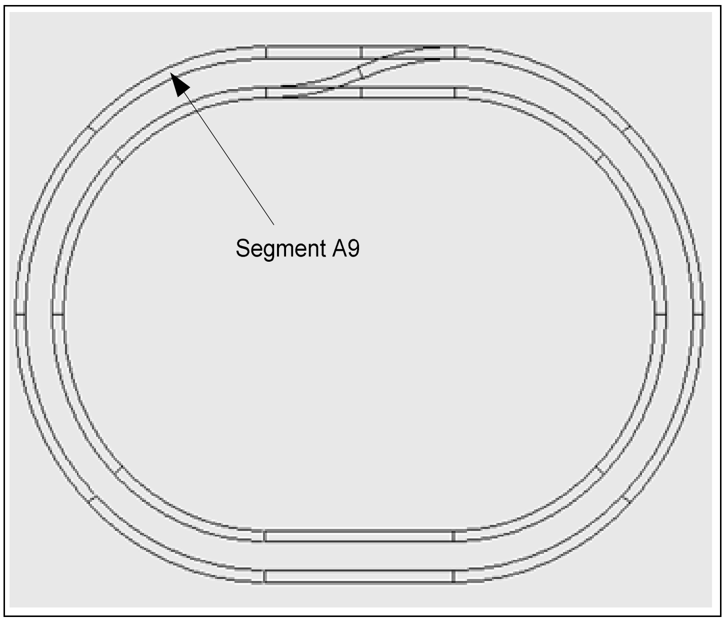


Figure 5.3: The track layout model

```
# standard prototypical curved track segment
openshape ST226
within ST226 {
     real angle
     angle = pi div 4
     real x1, y1, x2, y2
     x1 = (~/Radius2 - ~/TrackWidth div 2) * sin(angle)
     y1 = x1 * tan(angle div 2)
     x2 = (~/Radius2 + ~/TrackWidth div 2) * sin(angle)
     y2 = x2 * tan(angle div 2)
     line f, e
     f = [{0, ~/TrackWidth div 2}, {0, -~/TrackWidth div 2}]
     e = [{x1, y1 + ~/TrackWidth div 2}, {x2, y2 -~/TrackWidth div 2}]
     arc l, r
     l = [{0, ~/TrackWidth div 2}, {x1, y1 + ~/TrackWidth div 2}, \
          -angle * 180 div pi]
     r = [{0, -~/TrackWidth div 2}, {x2, y2 -~/TrackWidth div 2}, \
          -angle * 180 div pi]
}
# A9 track segment based on transformation of ST226
openshape A9
within A9 {
     point start, end
     real inDir, outDir
     start = ~/A10/end
     inDir = ~/A10/outDir
     shape track
     track = trans(rot(~/ST226, {0,0}, inDir), start.1, start.2)
     end = rot(start, start + {~/Radius2 @ inDir + pi div 2},~/ST226/angle)
     outDir = inDir + ~/ST226/angle
}
```

Listing 5.1: Definitions of a track segment based on transformation from a standard prototypical part. An extract from the script producing Figure 5.3.

The use of standard track pieces in Figure 5.3 resembles the use of physical prototypes. In this case, the designer chooses to use small-scale replicas of real standard track segments. The task involves selecting the right pieces and connecting them together in order to experiment with possible layout options. By its nature, the process is interactive and informal. The merit of a physical model over a mathematical model is that the designer can get concrete experience from the model by interacting with it - this helps the cognitive process of understanding and stimulates creativity (cf. Gooding's construals [Goo90]). The EM model of the track layout has the same quality as a physical model. The designer can interact with the model by making definitions. The Donald visualisation of the model gives immediate feedback to the designer's actions. Of course, the physical model has many attributes and behaviours that are not represented in the EM model, such as the weight of each segment and the rate of track expansion on heat under sunlight. Identification of additional attributes and behaviours is part of the design process. In EM, the designer can always extend the model by adding extra definitions to reflect his understanding and to embody practical experience obtained through interacting with the model. In this respect, an EM model has an significant advantage over a physical model. For example, if there is a budget for the cost of the track, the designer can introduce a price for each standard segment and add definitions for maintaining a total cost for the design. In the case of a physical model, it is necessary to manually maintain a separate cost model in parallel. The synchronisation between the two models has to be performed manually.

In a real situation, the designer typically has to consider integrating a wide variety of such different viewpoints. It is difficult for the designer to maintain the conceptual integrity of the whole design. In EM, such models can be integrated into one model in such a way that the selection of segments in the layout is automatically reflected in the total cost.

The model depicted in Figure 5.3 and the associated Donald definitions only specify the geometric appearance of the track. To further appreciate the potential power of integration, we can consider the segment connectivity model depicted in Figure 5.4. This model is a part of the Railway model that describes the combinatorial relationships between track segments. It defines a topological layout of the track. Such a layout is essential because the connectivity of track affects the possible paths that the trains can travel. We need a representation that expresses the relationship between adjacent track pieces more explicitly than physical coincidence of endpoints on the display. The topological layout depicted in the left-hand-side of the Figure 5.4 serves this purpose.

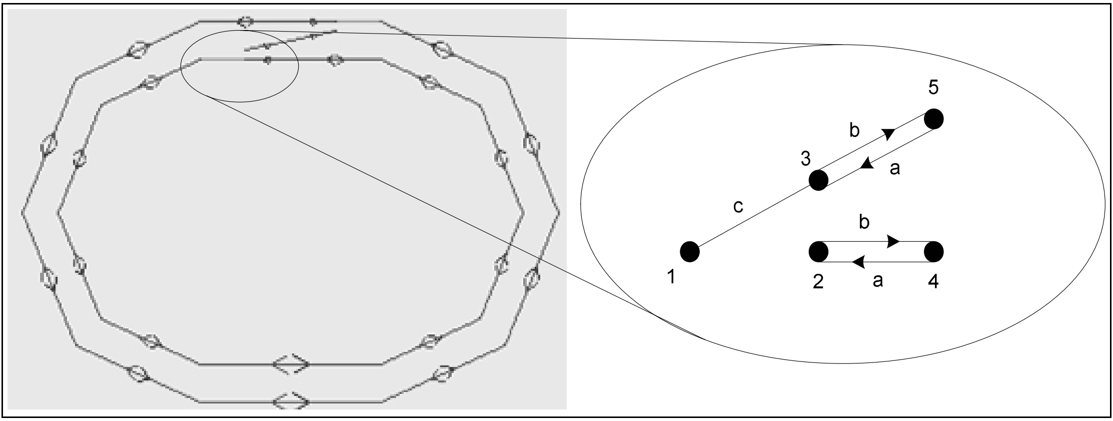


Figure 5.4: The connectivity model

```
1.   mode Point1 = 'abc'-diag 5   #points have 5 vertices and 3 colours

2.   Point1Stts = 1   #status of the points, 1 = closing the loop
3.   #defining the edges

4.   a_Point1{1}=1; b_Point{1}=1; c_Point1{1}=if(Point1Stts==1) 2 else 3
5.   a_Point1{2}=2; b_Point{2}=4; c_Point1{2}=if(Point1Stts==1) 1 else 2
6.   a_Point1{3}=3; b_Point{3}=5; c_Point1{3}=if(Point1Stts==1) 3 else 1
7.   a_Point1{4}=2; b_Point{4}=4; c_Point1{4}=4
8.   a_Point1{5}=3; b_Point{5}=5; c_Point1{5}=5

9.   Scale = 100                 #scaling factor for locating vertices
10.  Point1!1=A!1                #defining locations of the vertices of Point1 in
11.  Point1!2=A!1-[Scale/2,0,0]  #terms of the location of the first vertex of
12.  Point1!3=A!1-[Scale/2,Scale/10,0] #A, the graph representing the outer loop
13.  Point1!4=A!1-[Scale,0,0]
14.  Point1!5=A!1-[Scale,Scale/5,0]
```

Listing 5.1: Definitions that specifies the digraph at the right of Figure 5.4

The topological layout uses a directed graph with coloured edges to describe the connectivity between segments. Each vertex of the graph represents a track piece and each edge represents a join of connected track pieces. In the graph, the direction of an edge represents a direction for the traversal of the associated track segments. The right-hand-side of Figure 5.4 depicts one of the sets of points that link the two circuits - it contains 5 vertices (numbered from 1 to 5) and edges of 3 colours (colour a, b and c). The corresponding definitions are shown in Listing 5.1. The a-coloured edges represent the path that a train can travel in an anti-clockwise direction in the circular track. Similarly, the b-coloured edges represent the path in a clockwise direction. The c-coloured edge is an undirected edge that conditionally links to one of two vertices depending on the current status of the points (i.e. vertex 1 connects to either vertex 2 or vertex 3). The definitions in lines 2-6 of Listing 5.1 serve this purpose.

The track layout model and the segment connectivity model complement each other as two alternative views for the track design. Together they also provide the necessary context for adding the third submodel: a train simulation model. One feature for this model is that it introduces two trains as autonomous agents. Figure 5.5 depicts an integrated view of three models. In the figure, the positions of two trains are represented as bold lines on the track layout model. The current direction of a train can be inferred from the circle at the tail of the train. The bottom of the figure shows the interfaces for the signalman who controls the points and the two drivers of the trains. As the interface shows, each train can travel in a clockwise or an anti-clockwise direction, and at any time the train can be stopped by pressing the stop button. Moreover, the speed of train movement in the simulation is governed by a clock agent.

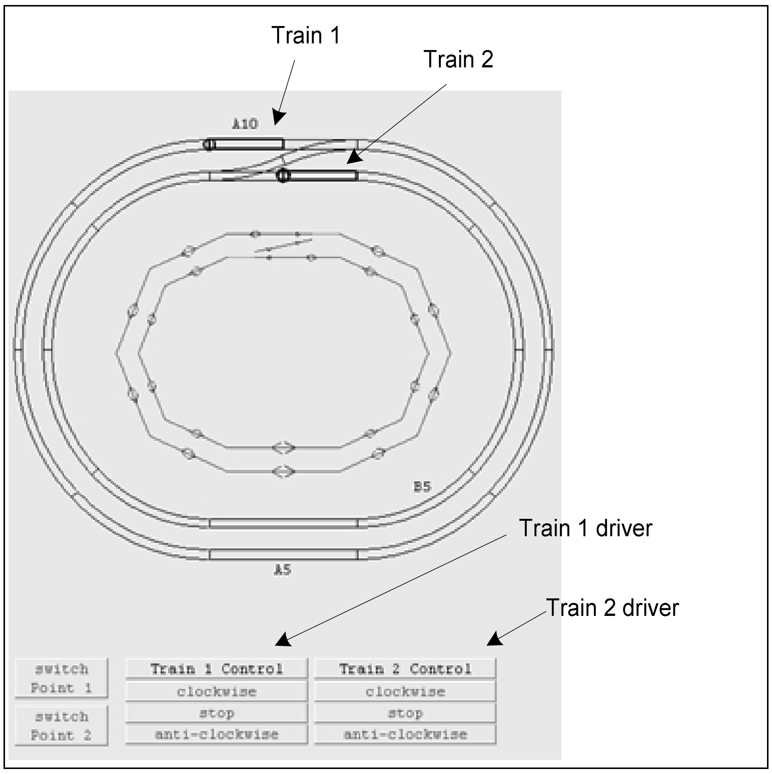


Figure 5.5: An integrated view of the track layout model, the connectivity model and the train simulation model

Adding a train simulation model to a track design potentially enables us to study it in a very rich context. For example, we can explore the consequences of track failure. Where normal operation is concerned, we can also use the integrated model to study the co-operation between the drivers and the signalman.

The last submodel integrated into the Railway model is the train station model. This model simulates the train arrival and departure protocols with reference to one of the trains (namely Train 1 in Figure 5.5) and the three stations (namely A10, B5 and A5 in Figure 5.5.). Figure 5.6 shows the entire visualisation of the Railway model. The right-hand-side of the figure depicts two stations: the most recent departure station for Train 1 (namely A10 as displayed at the top of Figure 5.6) and its next scheduled arrival station (namely B5 as displayed below). Passengers on Train 1 and at the stations A10 and B5 are also depicted. Each station has a stationmaster who communicates with a guard on the train to manage the train arrival and departure. The protocols that govern the interaction of the various agents within the train station model were derived from an LSD account listed in Appendix J.

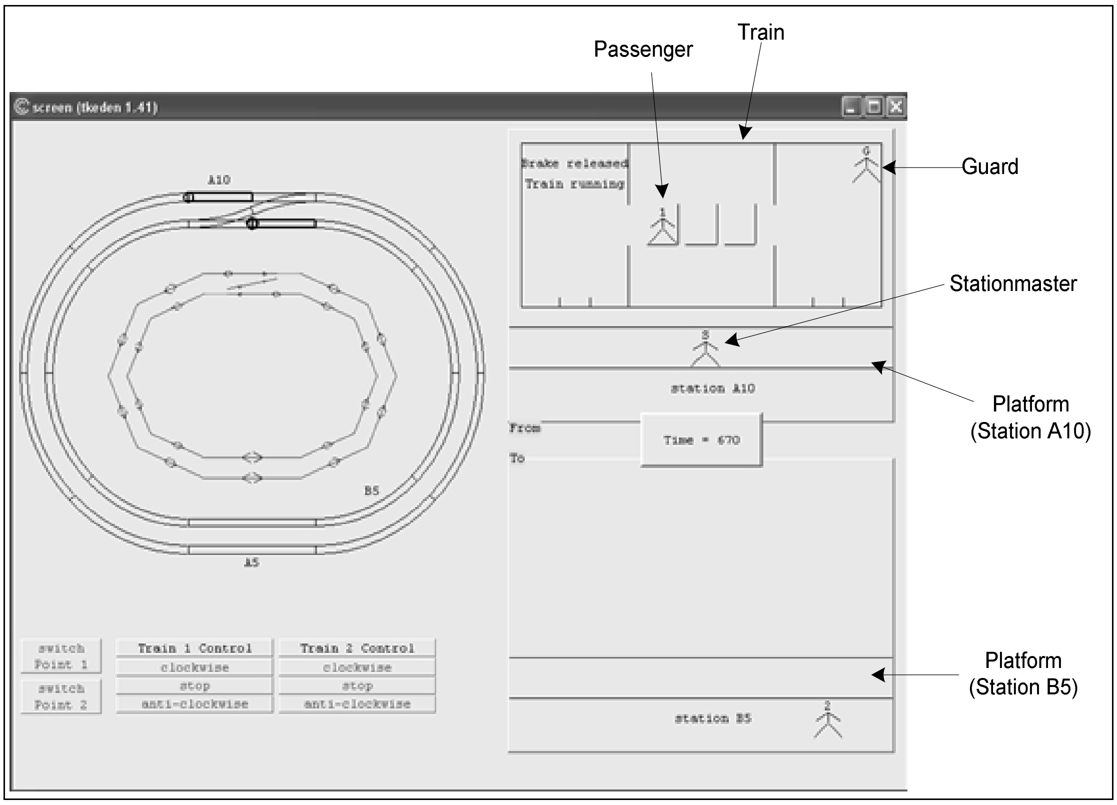


Figure 5.6: Visualisation of the entire Railway model

The Railway model itself has no clear purpose. It is an EM model that illustrates the possibility for embodying incoherent experience that lacks conceptual integrity. Bits of the model make sense independently but not when taken as a whole – the model can partially sustain a whole variety of interpretations, but every interpretation is problematic in some respect if the entire model is taken into account. For example, one obvious interpretation for figure 5.5 is that it is a model railway operated via the interface supplied by Train 1 and Train 2 controls. On the other hand, the visualisation of the train itself resembles the visualisations that are used in practice in electronic signal boxes to indicate which real life 'track segments' are currently occupied. In this case, there is a conflict between whether we interpret a track segment in Figure 5.5 as a standard piece of model railway track or as representing a segment of a real track possibly several miles long that can (according to normal real-life conventions for railway operation) be occupied by at most one train at any time. To resolve this conflict, we should need *either* to develop a track layout that more closely resembles a context for real world railway operation (e.g. match the layout to part of a particular real world railway network) *or* to visualise the trains as spanning several track pieces, as would be realistic in the model railway scenario.

Another example of a conflict relates to agency. In the model railway model in Figure 5.5, the trains themselves do not ‘really’ have drivers. It is the modeller or user who drives the train through operating the simplified control interface. Whether this is appropriate depends on the purpose for using this model. A track layout designer will probably find that a simple control interface is all he needs for testing different design layouts. However, a railway safety supervisor might find that it is crucial to model the drivers of the train in more detail. For instance, a driver’s reaction to signals is very significant where the safety of the railway is concerned (see e.g. the EM railway accident simulation model described in [Sun99b] which involves extensive exploration of a historical railway accident involving three drivers and two signalmen).

In developing a complex railway system, division of labour is unavoidable. There are many people involved in different aspects of the system. This diversity of concerns leads to conflicts in the design. A railway model such as this one facilitates the sharing of understanding across different members involved in the development. In this respect, the railway model acts as a medium for communication between developers.

When the train arrival and departure model was originally developed, there was no visualisation for the actual stations and the physical track between them. In an early prototype of the integrated model, passengers were observed to get off the train in unexpected contexts. This illustrates that some conflicts between aspects of the system under development can only be discovered in the process of integrating different aspects of the system. The concept of collaborative definitive modelling in the DMF provides an effective way to integrate different aspects of the system, to discover and resolve hidden conflicts, and thereby helps developers to maintain the conceptual integrity of the whole system design.

5.5 Enabling technologies for maintaining conceptual integrity

EM can be viewed as an enabling technology that helps to maintain conceptual integrity in the design of systems. It uses observations, dependencies and agencies as abstractions that can potentially bring unity to diverse human interpretations and viewpoints. Where software systems are concerned, the most closely related technologies are associated with the search for paradigms for data modelling and programming that are well-matched to human cognitive processes and everyday interaction. In this section, we review key data modelling and programming paradigms, and briefly consider their connection with EM.

5.5.1 Data modelling

In the early years of computing, all persistent state was stored in program and data files. The data in such files was recorded in many different formats, using data types specific to the programs used to process the data. As applications became more complex, the diversity of data representations and structures made an integrated data processing strategy problematic. The lack of conceptual integrity in data models lay at the root of this problem. The introduction of the first databases was associated with new techniques for data modelling aimed at maintaining conceptual integrity.

The central idea behind databases is to separate the presentation of data from how the data is stored at the physical level [Car95]. This is intended to achieve machine-independent representation of data. Database users do not have to worry about how the data is stored on the physical media before manipulating the contents of a database. Early database technology included hierarchical and network databases. These were then replaced by relational databases. Relational databases succeeded because of their foundation in the mathematical theory of relations, and because their query languages allowed even end-users to manipulate the database easily. At that time, by modern standards, the applications of computers were rather limited. Relational databases helped users to maintain the conceptual integrity of systems by providing a unified way to represent and manipulate data.

In recent years, as computers have become pervasive and popular, the applications of computer-based technology seem to have outgrown the set of primitive and predefined uses that relational databases support. Modes of data representation and manipulation are now far more diverse and complex, as are the interfaces between data, the users and the environment.

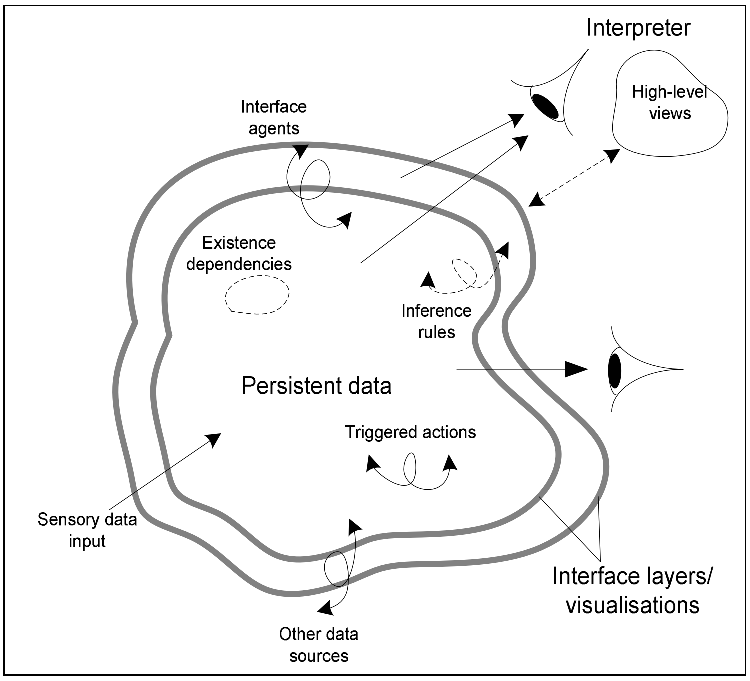


Figure 5.7: Modern context for data generation, access and manipulation

Figure 5.7 depicts the modes of data generation, access and manipulation represented in a typical modern computing application. The interactions between the persistent data depicted in the middle of the figure and the various different agents are complex. Many different forms of data input and sources of data are represented in the applications. The multi-user environment urges the consideration of different visualisations and customisations. The manipulation of persistent data is in part automated by transient processes that are hidden from the user using triggers, inference rules and data dependencies. It is then very difficult for an individual to comprehend the whole system as a whole entity. The conceptual integrity of the system is threatened. We need a better approach to data modelling that should be sufficiently expressive to meet the challenge of representing such an application in an intelligible way.

The principles of EM potentially offer an approach to data modelling that can bring conceptual integrity to modern computing applications. Observation, dependency and agency provide more general abstractions for data modelling than conventional data modelling paradigms can provide, and potentially enable data generation, access and manipulation to be managed in a unified framework. To help the reader to understand what EM can offer, we briefly compare the qualities of an EM model with those of other data modelling paradigms (cf. [Bey94b]).

EM versus relational data modelling. The main similarity between EM and relational data modelling is that both use dependency relationships to organise observables. In relational database design, tables are created according to the patterns of functional dependency observed amongst sets of attributes. In an EM model, observations are also structured with reference to the functional dependencies between observables. However, in a relational database, functional dependency is only used for designing tables. When it comes to the use of a relational database, changes to the table structure are assumed to be rare. In an EM model, functional dependency is used to maintain potentially much more dynamic relationships amongst observables.

The wide acceptance and powerful influence of relational databases in business computing over the last 20 years can be attributed to its major contribution to meeting two research challenges: that of end-user programming (as addressed by the development of relational query languages) and that of representing real world state (as addressed by the pervasive representation of business data in relational tables). It is interestingly to note that, since Codd’s conception of the relational model of data in 1970 [Cod70], the researches on the two themes of ‘end-user programming’ and ‘representing real world state’ have diverged, so that they are now conducted by-and-large separately. Where end-user programming is concerned, the modern emphasis is on visual programming. Where representing real world states is concerned the modern emphasis is on virtual reality (VR) and augmented reality (AR). The integration of these two research strands is no longer comprehensively supported by relational data modelling. The principles of EM potentially offer more scope for the reintegration of these two themes. Evidence to support this claim can be found in the fact that the Information Systems Base Language (ISBL [Tod76]), an early prototype for a relational query language, is essentially a definitive notation. This means that, by introducing the Eden Definitive Database Interface (Eddi) – a definitive notation based on ISBL – it is possible to subsume the functionality of a relational database within the EM tool TkEden. The emerging use of Eddi in conjunction with agent-related abstractions (e.g. as implemented by TkEden triggered actions) and definitive notations that offer more than tabular representations and a relational query interface illustrates the scope for EM to generalise relational data modelling.

EM versus rule-based data modelling. The need for data modelling to support rule-based activity underlying ‘intelligent systems’ motivated the deductive database as an extension of the relational database. In addition to all the relational database features, the database model stores mechanisms for reasoning about the stored information expressed using constructs of logic programming [Ram94]. Deductive databases can perform sophisticated inferences and draw conclusions from the data stored by using a predefined set of logical rules. In EM, the possible forms of user interaction are typically subject to fewer logical constraints. Moreover, in principle, EM can represent the ideas of triggering and deduction in ways that are safer in that they give the user more conceptual control, albeit at the cost of building explicit mechanisms that are less efficient.

EM versus object-oriented data modelling. Storing data organised as objects with relevant operations is conceptually very different from storing data organised as observables within definitions. In particular, indivisible interactions between real world entities are very well represented by definitions specifying dependencies between observables. In an OO data model, the indivisibility of interactions can only be modelled by message passing among objects. Whereas dependency maintenance in EM is automatic, in an OO model the integrity of updates can only be guaranteed by explicit specification of communications between objects. In the real world, the characteristics of an entity are determined by how it is observed. How an entity is perceived and how it can be transformed is dependent what agents are present in a system. In some circumstances, this means that what constitutes an entity is very ambiguous and changes over time. An EM model facilitates this dynamic view of entities by allowing the user to change what constitutes an observation on-the-fly. In contrast, the OO paradigm, objects have fixed boundaries, and changing the boundaries is usually difficult. Moreover, many observations may be associated with a context supplied by more than one object – as when we consider the attributes of a relation. It is difficult to model this kind of observation in an OO data model. The reason for this is that objects are good at representing circumscribed patterns of observation and transformation but not the degree of interconnectedness and fuzziness that characterises change in the real-world.

Features of the EM approach to data modelling can be summarised as follows:

- An EM model has the fundamental quality of a database that data sources are conceptually integrated but offers more scope for rich representation and interaction. Definitive notations support more general observables for the metaphorical representation of data of interest.

- Definitions provide a direct and dynamic mechanism for representing dependencies amongst observables.
- The concept of agency in EM is general enough to embrace all activities that are associated with data generation, access and manipulation.
- EM supports the integrated design and use of data modelling applications within the same framework (cf. the ‘factory’ analogy in section 4.4.3).

5.5.2 Programming paradigms

In this subsection, we shall consider the efforts made in computer science to help developers to maintain conceptual integrity in software systems. The discussion will centre around how programming research is converging towards a generic modelling concept that helps to achieve conceptual integrity.

Almost every paradigm of programming represents a shift in perspective on how we do programming. Most of the well-developed paradigms have a relatively small and coherent set of concepts. For example, logic programming sees everything in terms of logic rules; functional programming sees everything as functions; and in OO programming we are encouraged to see things in object terms.

The variety of programming paradigms reflects the nature of computer programming. Software system development resembles general problem solving in that there is more than one way of solving a problem. But why is one programming paradigm more popular than another if they all seem to have a set of coherent concepts? In particular, why is the OO paradigm popular but not the functional programming paradigm? One possible explanation is that the OO paradigm is more in line with the way we think about problems. Identifying objects is one of the tasks we usually do when solving problems in the real world. We also invoke the concept of inheritance when we recognise one problem as a special case of another. The success of the OO paradigm reflects a deeper concern that relates to conceptual integrity – we are not only seeking to design a coherent language around simple and consistent programming constructs, but also a coherent as well as a commonsense way of thinking about the world. True conceptual integrity of software systems can only be achieved by finding ways to represent our natural way of thinking about the world. We can find more evidence in support of this claim by considering the extensions of the OO paradigm discussed below.

One criticism of OO concerns the hierarchical structure of class inheritance. A class inheritance diagram depicts a relationship among objects in the real world. It is a mechanism for classification. As a commonsense analysis shows, a set of concepts can usually be classified in different ways in the real world, but class inheritance only represents one of the classifications. Real classification is dynamic whereas class inheritance in OO is static. This leads people to argue that OO cannot capture real world concepts faithfully [Jac02]. Recent research into 'subject-oriented' [Har95] and 'aspect-oriented' [Elr01] programming are trying to address this problem. The term 'cross-cut' is used to describe situations in which there are common concerns amongst a set of classes that cannot be captured by a class inheritance diagram.

Other critics are challenging the concept of object itself. Agent-oriented programming can be viewed as an extension to OO. In this context, Jennings [Jen00] cites the following definition: "An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.". This area of research recognises that there are in general two kinds of object: passive and active. The current OO paradigm favours the modelling of passive objects: most objects respond to requests by means of message passing and cannot initiate actions on their own behalf. The identification of passive and active objects is also a commonsense feature of everyday life situations.

OO and its extensions reflect a trend in programming research towards finding ways of representing real world concepts as naturally and faithfully as possible. In shifting the focus from objects to agents, we are moving towards a more 'commonsense' kind of modelling. If we regard Agent-oriented programming as a 'natural extension' of OO, then EM can be thought of as a 'natural extension' of Agent-oriented programming [2]. In fact, some researchers in Agent-oriented programming are proposing concepts that are in line with the principles of EM. This will be illustrated in the following discussion, in which we compare EM with other approaches to programming.

[2] Roughly speaking, classes in OO can be represented by EM agents; inheritance in OO corresponds to prototype inheritance in EM.

EM versus object-oriented paradigm

Typically, OO software development creates three artefacts: the object-oriented analysis (OOA) model – the model of the real-world problem; the object-oriented design (OOD) model – the model of a software solution of the problem; and the program – the implementation of the OOD model. Although all three models are represented in terms of objects and their relationships, they are inherently different. As Kaindl [Kai99] observes, the transitions from model to model are not smooth. Although the graphical notations used to represent OOA and OOD models are very similar, and this gives a feeling of smooth transition, in reality it causes more confusion. Also, in practice, the transitions between different OO models are very *ad hoc* (see Figure 5.8) – updating a particular model usually triggers a chain reaction of changes in the other two models resulting in time-consuming consistency maintenance and potentially more proneness to errors. In EM, we typically have only one model to deal with. This is because, in practical situations, we do system analysis, design and implementation incrementally in parallel. The fact that there is no clear distinction between different models from different kinds of software engineering activities helps to make the system design more adaptable and eliminates potentially time-consuming transitions between different models.

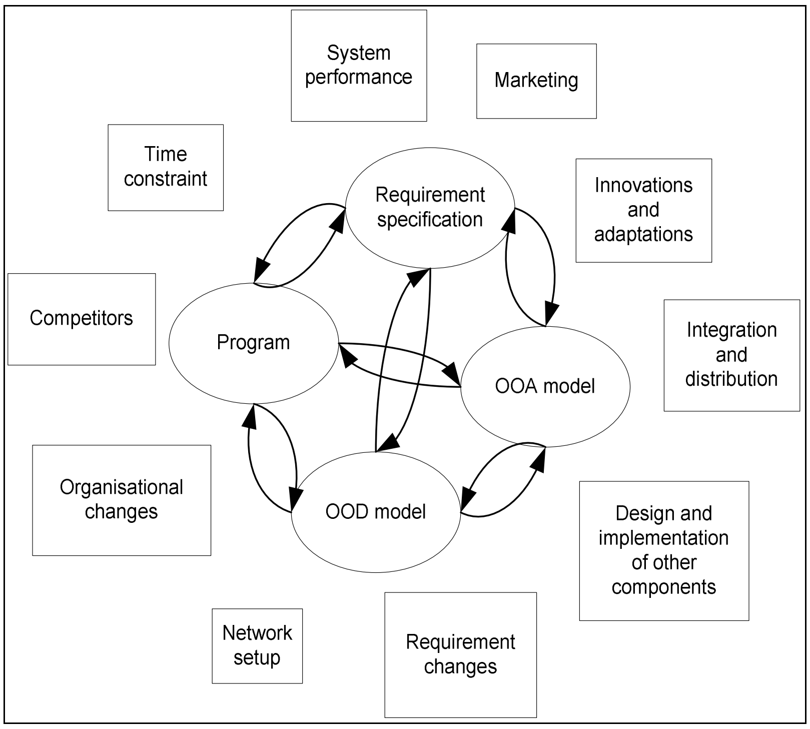


Figure 5.8: Chaotic transitions among different models in an OO software project

Another comparison can be made between an EM model and an object model of a real world entity such as a house. If we view each transformation of a definitive script to represent a house as a method, we can regard an EM model of a house as an object. This interpretation is acceptable if we already know the transformations to which a house is to be subjected in our proposed system. The distinction here is between the circumscribed knowledge required to specify an object in OO and the provisional knowledge of attributes and interactions associated with an EM model. Comprehending an object involves knowing everything we can do with it, but in itself an EM model does not circumscribe the transformations we can apply.

The difference between an EM agent and an object also reflects the different philosophies behind the two paradigms. An object contains state of its own. Communications between objects are via message passing (as implemented for instance by using synchronous function calls). Real world entities communicate with each other concurrently and freely in ways that cannot be captured by predefined message calls. EM agents can model real world entities more accurately than objects. An EM agent is more general than an object. All objects can be interpreted as EM agents but not *vice versa*.

EM versus other extensions of object-oriented paradigms

Agent-oriented software engineering can be viewed as an extension of the OO paradigm. There are two common notions of ‘agent’ in the field. The *strong* notion of agency models an agent in terms of mental notions such as beliefs, desires and intentions to be explicitly specified in both design and implementation; the *weak* notion of agency models an agent in terms of its observable properties as anything that exhibits autonomy, reactivity, pro-activity and social ability [Woo95].

If we adopt the weak notion of agency, it is difficult to identify agents because characteristics such as ‘social ability’ are difficult to define exactly. Lind [Lin00] argues for a less restrictive notion of agency. He suggests a *very weak* notion of agency whereby any entity can be an agent, and contends that “the conceptual integrity that is achieved by viewing every intentional entity in the system as an agent leads to a much clearer system design and it circumvents the problems to decide whether a particular entity is an agent or not”. This very weak notion of agency is in keeping with the EM notion of agency as discussed in chapter 2. However, the most distinctive feature of agency in EM is the mediation of agent interaction through the explicit representation of dependencies, which have so far not been well-explored in the traditional agent-oriented paradigms.

5.6 Summary

This chapter discusses the importance of maintaining conceptual integrity in a system design as the core preliminary activity prior to system conception. We point out that the main obstacle to maintaining conceptual integrity is the representation and management of the potentially incoherent experience involved in distinguishing and synthesising viewpoints at the personal and interpersonal levels. We propose that, by using ISMs, system developers can represent incoherent experiences so that they can be studied and shared through observation, interaction and experiment. The abstract discussion of these principles is illustrated with reference to the construction of the Railway model. We have also discussed and compared EM with data modelling and programming paradigms as enabling technologies to support the maintenance of conceptual integrity in a system design.

6 Beyond Systems: Ubiquitous Computing

In this chapter, we discuss how EM principles can potentially be applied to everyday practical computing as it may be in the future, where some systems can never be formalised by developers. The system concept is formed upon its use in a situation. Following Weiser [Wei91] we adopt the term ‘Ubiquitous Computing’ or in short ‘ubicomp’ to refer to an era where people will use a variety of computer-based devices to support everyday life activities. We firstly identify a variety of researches related to ubicomp. We discuss and summarise their shared visions. We argue that these visions are hindered by the lack of a conceptual framework to encapsulate the complexity and new requirements of ubicomp. In particular, little research has so far been conducted to develop a conceptual framework that explicitly supports both design and use of ubicomp devices. We argue that having a coherent conceptual framework is very significant for the conceptual integrity of ubicomp systems, and that this is fundamental to the success of ubicomp. We introduce a new conceptual framework based on EM principles and tools and illustrate this with examples. We shall discuss challenges involved in realising the framework. Finally, we shall describe some related research work and make comparisons with our proposed framework.

6.1 Visions of the future computing environment

Vannevar Bush, in his 1945 article “As We May Think”, envisaged a device that can manage and disseminate results of research [Bus45]. Baecker et al. [Bae95, p35] view Bush as the first person to see beyond the scientific use of the computer to its use as a “fundamental tool for transforming human thought and human creative activity”. Now, nearly 60 years later, the development of computer technologies has led to the realisation of Bush’s dream. A new vision of the role of computers has evolved. Computers are getting more and more pervasive. In his 2000 article “As We May Live”, Gibbs reports on research into ubicomp that applies computer technologies in our everyday life, such as a Georgia Tech’s four-bedroom house where there are more than 60 computers, 25 video cameras and 40 cabinet sensors [Gib00]. The vision has shifted from mainly concerning the way computers support us to *think* to a broader concern for the way computers can support us in *living* (cf. [Dau00, Gor97]). Devices with computing power are moving off the desktop into everyday items (cf. [Sch00a]).

Since Weiser's seminal paper [Wei91], research interests in ubicomp have grown tremendously. Amongst these, the most representative research titles include ubiquitous computing, invisible computing, disappearing computing, sentient computing and augmented reality computing. We shall briefly review these five research areas. This review will be the basis for our discussion of how EM principles can be applied to ubicomp in the rest of this chapter. All five research areas share prominent common themes, but each has its own distinctive emphasis.

Ubiquitous computing. Ubiquitous computing is also sometimes referred to as *pervasive computing* (e.g. [Ark99, Old99]). The term ubiquitous computing or ubicomp was coined with Weiser and colleagues at Xerox PARC in the late 1980s. Weiser promotes a new way of thinking about computer: "one that takes into account the natural human environment and allows the computer themselves to vanish into the background" [Wei91]. The motivating idea in ubicomp is to make computing power available through the physical environment invisibly. It has been viewed as the Third Wave of computing [Fol02]. The First Wave was many people per computer (mainframe). The Second Wave was one person per computer (personal computer). The Third Ware is characterised by many computers per person. The initial research areas identified by Weiser included new interaction devices, power consumption and wireless connectivity [Wei93]. More recent areas of interest include natural interfaces, context-aware applications, and automated capture and access [Abo00]. The goal of developing natural interfaces is to "support common forms of human expression and leverage more of our implicit actions in the world". Context-aware applications need to sense the environment and adapt the computation according to the use situation. These applications also need to provide facilities for users to capture and access live experiences.

Invisible computing. The concept of invisible computing, introduced by Norman [Nor99], is primarily concerned with how ubicomp technologies can be best integrated into everyday life. The idea of information appliances is central to invisible computing. Norman argues that general-purpose personal computers are difficult to use because they are technology-centred products that are inherently complex. The solution is to develop information appliances that are small, task-focused devices in place of big, complex, general-purpose personal computers. This is to design an information appliance to fit the task so well that the device “becomes a part of the task, feeling like a natural extension of the work, a natural extension of the person” [Nor99]. One distinctive feature of information appliances is their ability to ‘communicate’ among themselves and share relevant information. Norman suggests human-centred product development with a cross-disciplinary team of experts in marketing, engineering, and user experience. He promotes Contextual Design that includes six tasks: talk to specific customers while they work, interpret the data in a cross-functional team; consolidate data across multiple customers; invent solutions grounded in user work practice; structure the system to support this new work practice; iterate with customers through paper mock-ups; and design the implementation object model or code structure.

Disappearing computing. Disappearing computing is a European initiative on research and development of future computing. Its mission is “to see how information technology can be diffused into everyday objects and settings, and to see how this can lead to new ways of supporting and enhancing people’s lives that go above and beyond what is possible with the computer today” [Dis01]. Though the overall goal of disappearing computing is similar to that of other ubicomp research, its specific research strategy involves three sub-goals of particular interest in connection with this thesis. These are:

- creating artefacts that have the attributes of openness and connectivity;
- promoting emerging functionality through the collaboration of collections of artefacts;
- designing artefacts with the emphasis on people’s experience of them.

Sentient computing. Sentient computing is a collaborative project between the AT&T Laboratories and University of Cambridge [Sen02]. Its emphasis is on developing and exploiting technologies to give computers access to the state of their environment. The project started from the development of an ultra-sonic indoor location system. The system can provide the locations of tagged objects or people to an accuracy of about 3cm throughout a 10000 square foot building. The distinguishing feature of sentient computing is its use of sensors and resource status data to maintain a model of the real world which is shared between users and applications. One representative application enables a networked scanner to perform a selection from a list of functions presented in the form of a poster. A user can use a tagged object to point at one of the functions on the poster. This in turn triggers the scanner to perform the function. In this case, the system maintains a model of the real world that incorporates the geometric relationship between the poster and the tagged object. This model is a communication medium between the scanner application and its user. In effect, “the whole world is a user interface” [Sen01]. The goal of sentient computing is to make applications more responsive and useful by observing and reacting to the physical world [Hop99]. Research is based on three major themes: developing sensor technology, experimenting with application devices, and constructing platforms that connect sensors and devices together.

Augmented reality. Research on augmented reality aims to superimpose virtual objects upon, or compose virtual objects with, the real world. One way of augmenting reality is to overlay computer-generated graphics onto the real world. But augmented reality is not limited to sight – it might be applied to all senses. The motivation for augmented reality is to “enhance a user’s perception of and interaction with the real world. The virtual objects display information that the user cannot directly detect with his own sense” [Azu97]. A typical augmented reality system consists of three components: a head-mounted display, a tracking system, and a wearable computer [Bon02]. The head-mounted display allows us to see text and graphics generated by computers. The tracking system senses the location of a user’s head and eyes, and maintains the correct relationship between virtual objects and real world surroundings with reference to the user’s movement. The wearable computer provides portable, hands-free computational power to drive the whole system [Nap97]. Applications of augmented reality include medical visualisation, maintenance and repair, annotation, robot path planning, entertainment, and military aircraft navigation and targeting [Azu97].

Because of its potentially radical impact on everyday life, research on ubicomp has attracted many critics. A major common concern in critiques of ubicomp is that:

- ubicomp is driven by technology
- insufficient account is being taken of the human perspective on what is desirable in personal and social terms.

We are acknowledging that ubicomp is not necessarily a good thing in every respect. However, EM is offering an approach that promotes high levels of human engagement in the design and use of technology. This can make it easier for designers and users to develop ubicomp applications in a sensitive way.

6.2 Assessing the visions

Although the future development of ubicomp is difficult to predict, the approaches reviewed in the last section do reflect the same dominant emphasis in respect of four key issues: the roles to be played by *automation*, *visibility*, *connectivity* and *adaptation*. All the approaches discussed above aspire to full automation, hiding the technology from the users, indiscriminate interconnection of devices and systems that are self-adaptive. This thesis emphasises a complementary perspective: the need to keep humans in the loop; to encourage user engagement; to promote understanding and control over the interactions amongst devices; and to allow user customisation of the ubicomp environment.

6.2.1 Automation

One of the common but inadequate visions of ubicomp is having ubicomp systems that require nearly no human intervention. Negroponte advocates the use of “intelligent agents” as digital butlers that do all the work for you while you take it easy [Neg96]. Joseph describes this is as “the top of the IT agenda” [Jos02]. Tennenhouse, a vice president in the Intel Corp, advocates “getting the human out of the interactive loop” [Ten00]. This vision is only an industrial hype. Full automation is not plausible for ubicomp. The main reason is that the ubicomp environment is the environment we live in – where activities are situated and exceptions are the norm. Since we cannot prescribe the ubicomp environment, human intelligence has to be involved in solving ubicomp problems. Even the authors with visions for full automation seem to agree that there is a need for the involvement of intelligence. Joseph [Jos02] envisages that “computers will be intelligent enough to manage, configure, tune, repair, and adjust themselves to varying circumstances to handle the workload exposed to them efficiently”. Tennenhouse [Ten00] wants to automate the software creation process in ubicomp by generating software from specifications and constraints. Such visions presume that we can automate the management and specification tasks that seem to require human intelligence.

Undeniably, many people dream of sitting back and relaxing and allowing machine servants to help them to do all their work. This dream has become one of the driving motivations of ubicomp development. However, we cannot desire automation blindly for every device and aspect of ubicomp. Automation introduces problems of predictability and accountability. Edwards [Edw01] asks: “how will the occupant-users adapt to the idea that their home has suddenly reached a level of complexity at which it becomes unpredictable?" To paraphrase Langheinrich et al. [Lan02], "in order to lower the demands on human intervention in … a dynamic world … [we require] the concept of *delegation of control*, where we put automated processes in control of otherwise boring routines, yet provide *accountability mechanisms* that allow us to understand complicated control flows".

In the ubicomp environment, creating and supervising the automatic processes should be part of the users' role. We need to have *the human in the loop* and aim not to replace but to enhance and complement human abilities.

6.2.2 Visibility

Most of the visions for a ubicomp environment explicitly mention that computers should be invisible in the future (e.g. [Wei91, Nor99, Dis02]). The idea is that if we could somehow make computers vanish into the background, the complexity and frustration of using computers nowadays would disappear. In Norman's terms, to hide a technology is to hide the infrastructure of it [Nor99]. He envisages a world where information appliances with infrastructure hidden in the background largely replace conventional personal computers. This view has invited some criticism. Odlyzko [Odl99] believes that "[information appliances] will not lessen the perception of an exasperating electronic environment. The interaction of the coffee pot, the car, the smart fridge, and the networked camera will create a new layer of complexity", in which it creates new frustration. Langheinrich et al. warn that invisibility may lead to unpredictability: "… the ideal of the invisible, altogether unostentatious computer that silently hides in the background, might complicate or even impede the predictability of the system" [Lan02].

Visibility poses a dilemma. On one hand, it is desirable to hide the infrastructure of computer technology from its users, because that might just make the system easier to use and comprehend. A common view is that most users seem to have no interest in how a technology works so long as it does work. On the other hand, when things go wrong, as they often will in the case of computer technology, hidden infrastructure might hinder the possibility of fixing the problem promptly and safely.

In fact, sometimes the idea that "infrastructure should be invisible" is the fundamental cause of frustrations. For example, the latest versions of the Microsoft Windows operating system (e.g. Windows XP) hide file extensions from the user by default. There is an increasing number of people who do not know what file extensions are for, which might be a good thing because often it is the applications that mostly care about them. However, some of the file extensions can be shared by more than one application. For example, a “.txt” file can be used by Notepad, Wordpad, and MS Word. Frustration arises when a user wants to open a “.txt” file using MS Word – on double-clicking the “.txt” file, Notepad pops up every time but not MS Word! The problem then becomes: what should be visible and what should not? Unfortunately, the answer will depend on the individual user and situation.

Hjelm sees some analogy between the development of radio and development of computer technology [Hje01]. The development of radio went through three design phases: the archaic, the suppressed and the utopian. In the archaic phase, the radio was a new invention that was intrusive in a home environment and required an expert to use it. In the suppressed phase (because the product was not widely accepted), commercial applications that involved hiding the unfamiliar radio in big bulky but familiar objects such as grandfather clocks were explored. In the utopian phase, the radio was transformed into the compact, usable, and portable forms now in wide use. The development of computer technology might now be viewed as entering the suppressed phase, where people are embedding computers into every imaginable everyday object.

Streitz [Sto01] argues that causing the computer to disappear is only the first step towards achieving the final goal of “coherent experiences”. He adds “… coherent experience is the result of the combination of macro affordances (e.g. physical shape and form factor) and certain micro affordances (e.g. tactile characteristics of the artefact’s interface) in combination with the software providing appropriate interaction affordances”. We believe that this shift of emphasis to coherent experience is very important to the development of ubicomp. After all, it is *users’ engagement* with the ubicomp environment that governs its success. To appreciate the true meaning of invisibility we should ask questions about users’ engagement in addition to the more commonly asked questions about ways to hide infrastructure. Therefore, on the basis that exposing and understanding a technology is the first step towards making it conceptually invisible, it might be good for users to know and understand more about the infrastructure of the ubicomp environment. This thinking has an important implication: it leads us to place a conscious emphasis on the design of infrastructures with conceptual integrity that the user can understand easily.

6.2.3 Connectivity

High connectivity is another feature of ubicomp. Norman describes “a distinguishing feature of information appliances is the ability to share information among themselves” [Nor99]. In ubicomp, each person is surrounded by hundreds of wirelessly interconnected computers [Wei93]. Through connectivity, a collection of artefacts can act together and produce “new behaviour and new functionality” [Dis02]. Current technology is certainly capable of making this vision come true. Technologies like Bluetooth [Blu02], a short-range, low-power radio frequency technology, already promise to standardise wireless communications.

Connectivity has become part of the common language of ubicomp – so common that people often forget to justify or even think about reasons for the connections. What “new behaviour and new functionality” that connectivity supports is yet to be discovered. In fact, sometimes connecting everything to everything else is not a good thing. Connectivity can cause new complexity and frustration for ubicomp [Old99]. In [Luc99], Lucky amplifies on the potential frustrations: “My refrigerator… would refuse to open at certain hours of the day, having talked to my bathroom scales”; “My car is no longer the friend I once knew. If I exceed the speed limit, it reports me, and if I try to park illegally, it refuses to turn off or to let me open the door”.

In this context, the key issue for the user is knowing the purpose of the connections, and being able to *understand and control* them at will at any time. Edwards et al. [Edw01] point out that we need new models of connectivity for users to control, use, and debug the devices that are interacting with one another in the environment. Questions like “How can I tell how my devices are interacting? What are my devices interacting with, and how do they choose?” [Edw01] should be easy to answer in the future ubicomp environment.

6.2.4 Adaptation

In the ubicomp environment, requirements are unsettled. Users’ needs change over time, so that a ubicomp system should facilitate dynamic adaptation to various situations. Research on ubicomp usually associates adaptation with context-awareness of applications (e.g. [Abo00, Dey01, Lae01]). Abowd et al. [Abo00] point out that “ubicomp applications need to be context-aware, adapting their behaviour based on information sensed from the physical and computational environment”. The definitions of the term ‘context’ given in the literature vary but a generic definition can be found in [Dey01a]:

> *"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

Because of the aspiration to develop fully automated systems discussed earlier, the typical research focus is on capturing context by using sensor technologies. Capturing context through sensors only works when we can define the set of possible contexts needed in advance. However, in a ubicomp environment what is relevant to the interaction between a user and an application cannot be fully specified, and usually relevance changes over time. Sometimes, what is relevant is very subjective to the user. Inferences made through the preset sensors may not be accurate. As Edwards et al. [Edw01] observe "… [simple sensing] may report that I am present in a room when, instead, I have simply left my active badge on the desk".

A complementary way of trying to fulfil the adaptation requirement of ubicomp is through 'user modelling'. Traditionally, user modelling is about constructing an explicit profile of properties and preferences of the user in the system. The profile is used as the basis for adaptation and personalisation of the system. There are two approaches to user modelling: *adaptive* and *adaptable* [Fis00, Kul00]. In the adaptive approach, the system dynamically adapts itself to the current task and the current user. In the adaptable approach, the system gives substantial support to allow the user to change the functionality of the system.

Systems like GUIDE (a tourist guide system [Che01]) and PDS (a personal daily system [Byu01]) use both context-aware and user modelling approaches. Combining user modelling with context-awareness in an application improves the system's adaptation capability to some extent. However, there is no way for a system to take full account of all the preferences of the user by just maintaining an explicit representation of the properties of the user. In fact, even the user might not know his or her preferences in respect of a system. Consider one of the scenarios described in [Byu01] for PDS is "When a user passes by a theatre, the PDS can notify the user that the theatre is playing the user's favourite movie." In this case, the location together with a film preference of the user triggers the notification. The question is: who is to specify this behaviour of the system? If it is to be the system, we have the issue of properly predicting the user's state of mind; if it is to be the user, we have the issue of adequately supporting the user's need to specify the behaviour. Whichever is the answer, however, it is clear that context-awareness and user model maintenance alone are not sufficient for system adaptation. Even in this simple situation, we need mechanisms to *customise* the system to suit individual needs. The system should be able to understand the user's preference and have some reflective capability concerning the way in which these preferences are expressed within itself. The user should likewise be able to understand his or her preferences and relevant functionality of the system. Evolvability of the system comes from mutual understanding, which in turn comes from openness to interaction and customisation.

Figure 6.1 summarises the ideas of this section.

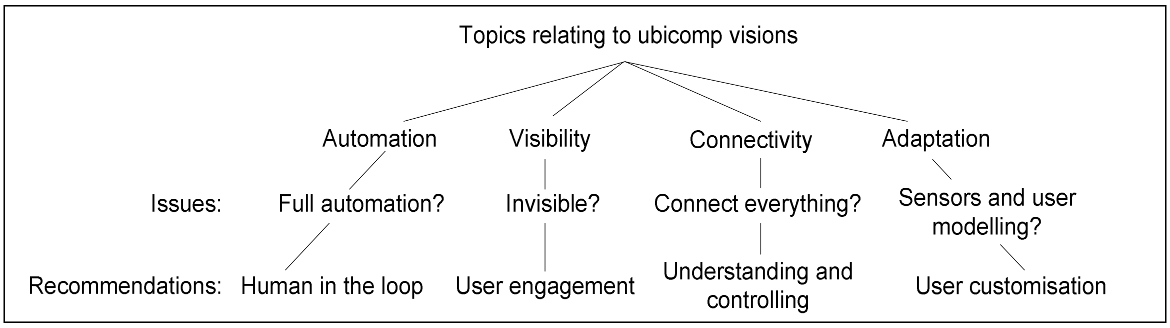


Figure 6.1: Topics, issues and recommendations relating to common ubicomp visions

6.3 A new conceptual framework (SICOD)

In this section, we shall discuss the potential application of EM principles to ubicomp and propose a new conceptual framework in which the issues mentioned in the last section can be more effectively addressed. The potential contribution of EM research to ubicomp becomes apparent when we consider the properties of a ubicomp environment. These include:

- The concurrent nature of a ubicomp environment – EM principles are based on a commonsense way of construing phenomena (chapter 2).
- The importance of context – In contrast to classical approaches to programming, EM gives prominent emphasis to modelling state and situation (chapter 2). Treating contexts as states is identified as a key issue in the ubicomp literature (e.g. [Dey01a, Rah01]).
- Unforeseeable user requirements – EM principles can be used as a heuristic way towards human problem solving (chapter 4).

- Dynamic and unpredictable integration of different devices – EM principles assist integration (cf. [Bey00a]) and help to maintain conceptual integrity (chapter 5).

The EM conceptual framework for ubicomp can be described as a framework for "controlling devices through building EM models". We call these special-purpose models Interactive Control Models (ICMs). In contrast to other conceptual frameworks (which will be described in section 6.5 below), the EM conceptual framework aims to:

- make the infrastructure of ubicomp more *visible* to the users
- provide principles to help users to maintain the conceptual integrity of their views of ubicomp systems.

We shall describe the EM conceptual framework – to be called 'soft interfaces for the control of devices (SICOD)' – in detail, and illustrate its potential use with a ubicomp example. An ICM typically consists of a set of Interactive Device Models (IDMs) and an Interactive Situation Model (ISM). The use of EM principles in the construction of IDMs has been discussed in some detail in previous papers (e.g. [Bey01b, chapter 5 in Run02]). The development and use of ISMs has been a common theme in recent EM research (cf. [Sun99a, Bey99, Bey00c, Bey01b]). In developing ISMs for a ubicomp environment, we propose novel methods for constructing ISMs from IDMs that are aimed at the end-user.

The left of Figure 6.2 shows an ICM with three IDMs (depicted by pentagons) linked to an ISM (depicted by a circle) by dependencies. The whole ubicomp environment can contain a network of ICMs (see the right of Figure 6.2). Notice that an IDM can be shared by more than one ISM. This reflects the fact that many devices are shared resources in a ubicomp environment.

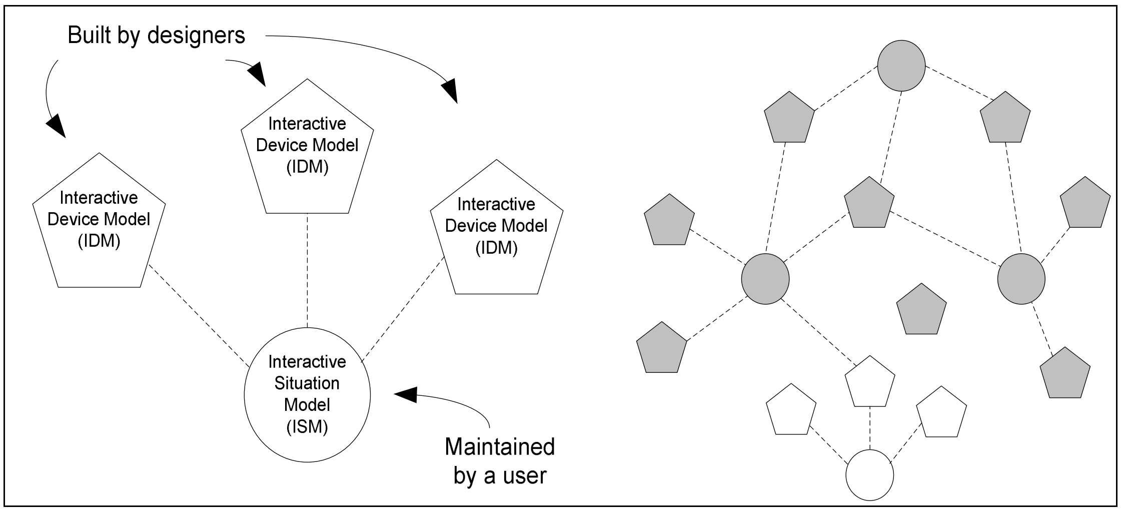

Built by designers
Interactive Device Model (IDM)
Interactive Device Model (IDM)
Interactive Device Model (IDM)
Interactive Situation Model (ISM)
Maintained by a user

Figure 6.2: An Interactive Control Model (left) and ubicomp environment with many interacting ICMs (right)

Both IDMs and ISMs are built using EM principles but the differences between them are summarised as follows:

IDM	**ISM**
Built by designers.	Built and maintained by users.
Corresponds to a particular device for generic application.	Corresponds to a particular situation for individual use.
Assists users to gain a conceptual understanding of the device.	Assists users to configure devices through creating definitions to establish dependencies between the states of IDMs and users' situated observation.

IDMs are relatively stable models which are built by the device designers. Each device has an IDM. ISMs are models created by users of devices. An ISM links a set of IDMs through dependencies.

To illustrate our conceptual framework, we consider a ubicomp scenario similar to that introduced by Huang et al. in [Hua99]. The scenario is as follows:

A user sets up a model to control the stock of a particular drink in her fridge, in this case, canned cola. Four computer-based devices are involved: a fridge, a personal Global Positioning System (GPS), a retail store information device, and a clock. The fridge maintains a count of how many cans of cola there are in it (possibly through some *kind of object tagging and detection technology); the personal GPS gives the current location of the user; the store information device gives information about the location and opening hours (possibly through a web enabled device connected to the store's homepage); the clock gives the current time and date. The user is holding a party on Friday and has to make sure that there is enough cola in the fridge. She wants the system to remind her to buy the drinks when she is near the store. If she does not go near the store before Friday, the system will remind her to buy the drinks on Friday.*

One possible realisation of this scenario within the SICOD framework is depicted in Figure 6.3a. The definitions involved are displayed in Figure 6.3b. These two Figures are complementary representations. Figure 6.3a depicts the interface for the end-user to specify the definitions in Figure 6.3b. A prototype implementation of such an interface will be described in chapter 8.

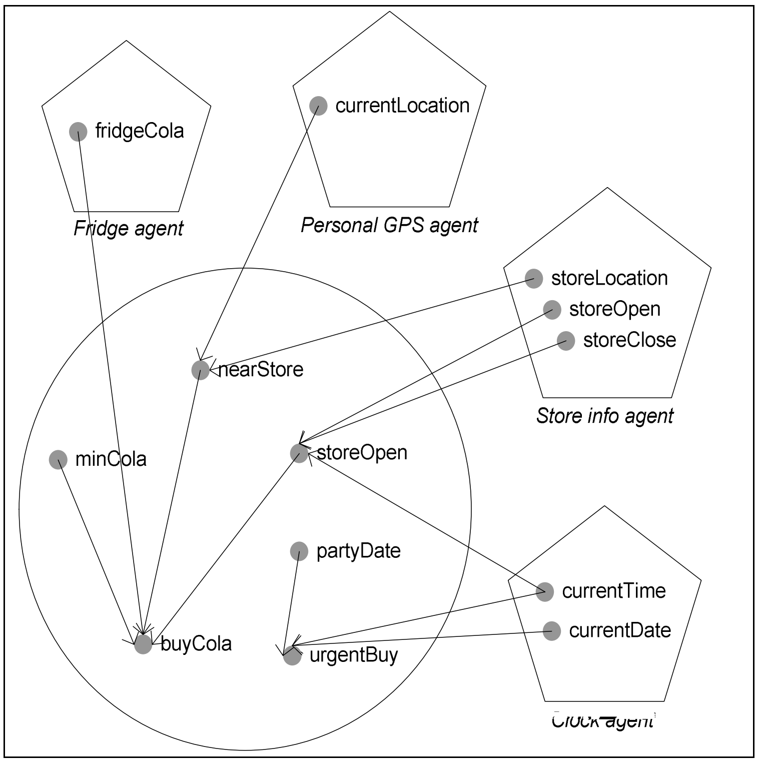


Figure 6.3a: An ICM of a particular drink stock control

```
IDMs
storeOpen is 8
storeClose is 20
storeLocation is 1234
currentTime is 10
currentDate is 20020815
currentLocation is 1234
fridgeCola is 2

ISM
partyDate is 20020820
minCola is 10
storeOpen is storeOpen < currentTime < storeClose
nearStore is storeLocation = currentLocation
buyCola is minCola > fridgeCola and storeOpen = True and nearStore = True
urgentBuy is minCola > fridgeCola and partyDate = currentDate
```

Figure 6.3b: Definitions of ICM of the drink stock control

The 13 definitions in this ICM can be divided into two groups: fixed and changeable definitions[3]. Fixed definitions are definitions provided by IDMs, for example, the user cannot change the definition of `storeOpen` (store opening time) because this definition is determined by the store. In this case, each IDM corresponds to an agent. The changeable definitions are the definitions in the ISM that express the user's special requirements of the drink stock control system. The meanings of the definitions are quite obvious when we look at the definitions in detail in the Figure 6.3b. Without going into detail about each definition, we draw attention to two important definitions: `buyCola` and `urgentBuy`. The `buyCola` definition specifies that if the stock of cola in the fridge is below the minimum amount and the user is near the store within store open hours, the system can remind her to buy the drinks. The `urgentBuy` is especially for the party schedule on Friday – so that if there is not enough cola in the fridge on the day of the party, the system will issue a warning.

Note that all the definitions contained in the IDMs in Figure 6.3a are fixed and so for reference only. These IDMs can be thought of as models of sensors linked to the real world. However, an IDM can also contain definitions for actuators that act on the real world. One example is depicted in Figure 6.3c below. Figure 6.3c extends Figure 6.3a by adding an alarm device. This device's IDM contains only one definition, `alarmOn`, where the right-hand-side of the definition can be changed by the user. The simple behaviour of the alarm device is that it generates a tone whenever the

[3] Note that in an ICM variable assignments are represented as constant definitions. For example, we write "fridgeCola is 2" instead of "fridgeCola = 2".

value of `alarmOn` is set to be `True`. We can define `alarmOn` as `buyCola` to specify that the system should remind the user to buy the drinks by generating a tone. This example shows how a device can be configured through its IDM by redefining an observable in the ISM.

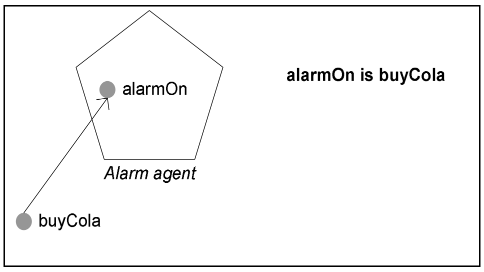


Figure 6.3c: An alarm device extension

By looking at the above application, we can identify the following advantages of the SICOD framework:

- An IDM assists the understanding of how a device works by representing the characteristic and persistent dependencies between its observables. This allows more effective communication of the designer's conceptual model of how a device works to the user.

- ICMs help users to maintain conceptual integrity of the system. For example, it is easy to find the answers to 'why' questions – cause and effect is clear and accessible through navigation of the dependency graph of an ICM (e.g. if the value of `buyCola` is `True`, the user can investigate why this is so by following the dependency links). Since conceptual integrity is a subjective matter, it is doubtful whether the designer of the system can effectively prescribe a view with conceptual integrity for the user. For this reason, we instead need a conceptual framework that helps the user to maintain conceptual integrity. Unlike a traditional window-based GUI, the SICOD framework enables users to have a global conception of system state.

- Uses of the system are highly customisable. In a typical ubicomp environment, there are no fixed boundaries for the system. The system emerges when we build an ICM to link devices together. In fact, the system is created by the user and therefore, high flexibility is guaranteed.

- The sharing of devices is mediated naturally by the framework. The values of the observables in an IDM can be used by many users concurrently. Particular definitions in an IDM may be changeable and may also be restricted to reflect the fact that they can be changed by only one user at a time. For example, this applies to the definition of `alarmOn` in Figure 6.3c.

- Contexts in the ubicomp environment are explicitly represented as states within the SICOD framework. This is in line with the practice of representing contexts as states to be found in much ubicomp literature. For examples, Dey writes that "… a collection of states can be described as a situation." [Dey01a]; Rahlff et al. defines personal context as "a snapshot of the state of the most important situational parameters: personal identification, time, location, task at hand, nearby objects, nearby people, etc." [Rah01]. Furthermore, by using definitions, we can explicitly specify the relationship between states.

- The framework supports rapid prototyping of a particular use situation. As new requirements come from the users, the best people to prototype the system are the users themselves through the building of ISMs. The designer's responsibilities are to build the functions to support the definitive notation and the sensing technology to support the automatic update of observables.

- The user can use an ISM for a system to adapt the reliable behaviours of IDMs so that they reflect the current situation. For instance, in the context of the drink stock control, a user can configure the ISM so that the notification that the supply of cola is exhausted is suspended during the night.

Apart from the advantages listed above, the SICOD framework also addresses the four important issues discussed in connection with visions for ubicomp in the last section. We shall now discuss these in turn.

6.3.1 Human in the loop

As we discussed in section 6.2.1, full automation is over-hyped in the visions of ubicomp and we need to put the human in the loop. This point can be illustrated by considering the difficulties in obtaining contextual states automatically. The states of digital devices are the easiest to obtain automatically. States of the environment, such as room temperature, illumination intensity and noise level are harder to obtain because they depend on sensor technology. What can be reliably sensed by current technology is limited to very primitive contextual state. States such as the orientation of an arbitrary physical object, the meaning of a sign and the topic of a conversation are very difficult for devices to detect automatically. It is doubtful whether all such states, which involve human perception and interpretation, can be reliably sensed by computers automatically. The ICM of the cola stock control scenario is a good example – most of the behaviour of the system is determined by the ISM, not by the IDMs.

Most activities in our everyday life are situated. Situated activities mostly contain actions that involve conscious reference to the context and the choice of course of action [Sun99a]. For this reason, ISMs in the SICOD framework play an important role in capturing context-awareness. An ISM maintained by the user of the system represents a particular use case of a system of devices that cannot be prescribed and therefore, the ISM itself cannot be automatically built by the system (cf. circumscription of use cases in UML).

The SICOD framework supports problem-solving in a ubicomp environment that is based on *intelligence captured through practical experience* – a precept that human agents tacitly use to solve problems encountered in the real world [Bey94a, Sun99a]. It supports experimentation, discovery and exploration of an environment prior to the identification of desired reliable behaviour. Automation comes later when patterns of interaction become reliable and a system emerges. However, familiar and reliable patterns cannot take account of dynamical changes in the ubicomp environment. Humans should be involved in constant revising of the ICM to adapt to new requirements and new contexts.

6.3.2 User engagement

We argued in section 6.2.2 that true invisibility comes not hiding infrastructure but from the user's engagement in the primary activities of interest. Sometimes, it is appropriate to make infrastructure visible to the user.

The SICOD framework provides a direct way of modelling the observation of the system and its environment by the user. In particular, just like cells in a spreadsheet, observables associated with an ICM are all task-oriented (the term used by Nardi in [Nar93]) – they represent states that can be perceived and observed by the user; they represent entities that are of interest to the user for the particular task or situation. The user can engage with the task at hand more easily using a spreadsheet than another conventional programming paradigm [Nar93]. An ICM inherits this advantage of the spreadsheet.

The modelling of observation provided by the SICOD framework is very useful in a ubicomp environment. Since the designer cannot prescribe all the observables in a user’s mind in advance, it is left to the user to describe the observables using an ISM. After a period of time, the management of some of these observables might follow such commonplace patterns that the designer can prescribe them in systems for other users.

In the SICOD framework, states are represented explicitly, and interactions are direct. This allows users to define what is to be observed, and allows users to engage in setting up the devices based on their subjective experience (cf. the computer as instrument discussion in [Bey01a]).

6.3.3 Understanding and controlling the connectivity

Where connectivity is concerned, we have emphasised user understanding and control of the interconnections between devices. In the SICOD framework, this is addressed by the notions of agency and dependency.

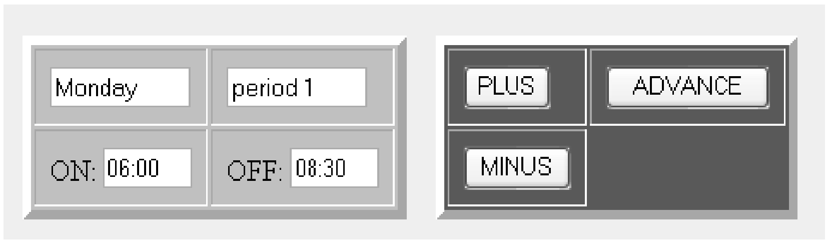
Within the framework, what to connect to what is entirely up to the user. With dependency graphs, a user is able to get a visual understanding of the interconnections. This helps the user to maintain a clear conceptual model of the communications and devices involved. The explicit representation of dependency helps to make the system traceable, so that, for instance, communication between two devices exists only if there is a dependency link. This helps to ensure that the user can tell why and how the devices are interacting each other.

The notion of agency supports a user’s natural commonsense attribution of state change. For example, with reference to the cola stock scenario, a user can refer to the store location definition but cannot change the definition; the fridge agent is responsible for counting the colas. This is a natural application of LSD analysis introduced in chapter 2.

6.3.4 User customisation

We argued in section 6.2.4 that conventional context-awareness and user modelling techniques are not sufficient to meet the system adaptation requirement for a ubicomp environment. We also need ways for the user to customise the system. Ideally, these should be flexible and easy to use. Usually there are trade-offs between flexibility and ease of use so that if a way is flexible it is usually not easy to use and *vice versa* (cf. [Odl99]). However, the SICOD framework can arguably provide ways to customise the system that are both flexible and easy to use.

To illustrate this point consider a central heating controller, called Balmoral, based on a real-life model described by Green [Gre99]. Figure 6.4 shows the control panel and a summary of instructions on how to use the controls.



Summary Instructions

These controls determine three periods of heating for each weekday. By clicking the ADVANCE button you can move from day to day and within each day you can move through the ON and OFF times for each period. The PLUS and MINUS buttons alter whichever panel is currently highlighted.

When the day of the week is highlighted, PLUS moves on to the next day, and ADVANCE brings up the settings for the day.

Figure 6.4: A layout of a central heating control panel and its instructions adapted from [Gre99]. On this control panel, there is an LCD on the left and there are buttons on the right.

By pressing buttons on this control panel, a user can set up three periods of heating for each weekday. The operation of the control panel is highly dependent on the mode switching button called ‘ADVANCE’. Mode switching buttons like this also exist in most programmable VCRs, washing machines, digital watches and desk clocks. They are also popular in conventional windows-based GUIs. Mode switching makes a system difficult to comprehend because it demands users to switch the perception of the system accordingly. With a conventional ‘press-button’ interface, however, it is usually unavoidable because of the physical constraints on the number of buttons that we can put on a control panel. For example, it would be inconvenient to have a separate pair of ‘PLUS’ and ‘MINUS’ buttons for every weekday and every heating period. We shall refer to this kind of interface as a ‘hard interface’.

In designing a hard interface, we not only have to prescribe the functionality of the system but also the possible observables and their interpretation by the user in the situations of use. This is a potential barrier to providing a system view that has conceptual integrity for the user (e.g. consider the different roles that the ‘PLUS’ and ‘MINUS’ play according to the current mode of operation). It also affects the flexibility of the resulting system. For example, the central heating interface only allows the user to enter 3 heating periods for every weekday – a rather arbitrary prescription imposed by the designers.

The SICOD framework provides a different way to configure the system. We can regard an ICM as a ‘soft interface’ to a system of ubicomp devices. Applying the SICOD framework, an ICM of central heating control would be like the one shown in the Figure 6.5a. The corresponding definitions are shown in the Figure 6.5b.

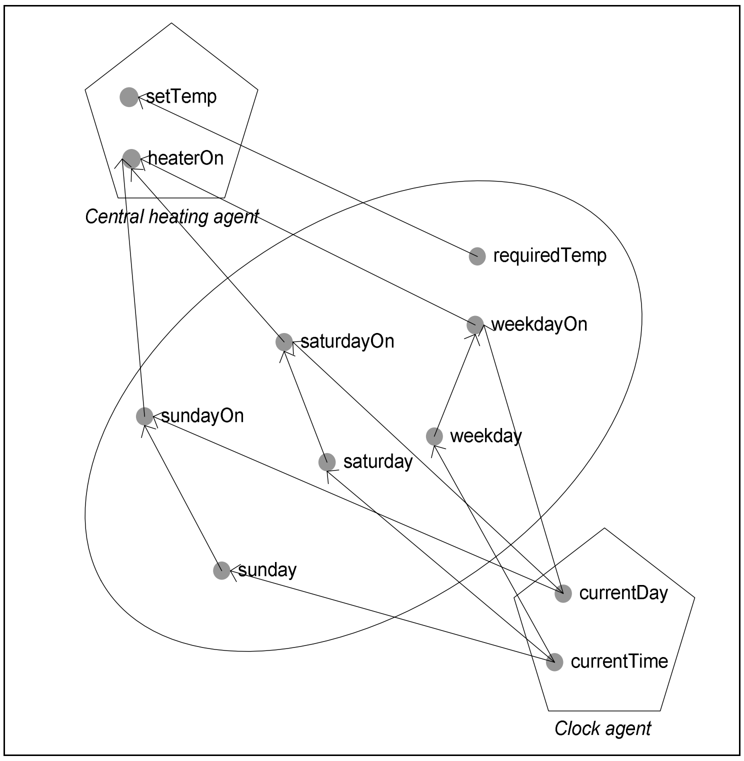


Figure 6.5a: An ICM for central heating control

```
IDM
setTemp is requiredTemp
heaterOn is sundayOn or saturdayOn or weekdayOn
currentDay is 4
currentTime is 10

ISM
requiredTemp is 20
sunday is 9 < currentTime < 22
saturday is 9 < currentTime < 12 or 16 < currentTime < 22
weekday is 7 < currentTime < 8 or 6 < currentTime < 22
sundayOn is sunday and currentDay = 7
saturdayOn is saturday and currentDay = 6
weekdayOn is weekday and 1 <= currentDay <= 5
```

Figure 6.5b: Definitions of ICM of the central heating control

The observables sunday, saturday and weekday specify the user's configuration of the heating period. The observables `sundayOn`, `saturdayOn` and `weekdayOn` evaluate to `True` only if they can match their own day and heating period with the current time and day information provided by the `Clock` agent. The central heating will be turned on only if `heaterOn` evaluates to `True`.

With the ICM, extension of the system becomes very easy. For example, suppose that the user has bought sensors to detect if there is any person in the house. The central heating ICM can make use of this information to allow more efficient use of central heating – the user can change the `heaterOn` definition to:

```
heaterOn is (sundayOn or saturdayOn or weekdayOn) and houseNotEmpty
```

A conventional central heating interface will surely have difficulty in embracing this requirement without replacing the whole control panel and developing mechanisms to link to the sensors.

The SICOD framework improves both flexibility and ease of use of the resulting system:

- the system becomes more flexible as its ICM is open to change and extension;

.

- the system becomes easier to use as the SICOD framework provides a simple interface through which to customise devices. The user needs only to learn the underlying definitive notation to be able to control a variety of different devices.

The idea of allowing the user to customise most parts of the ubicomp system is an alternative approach to the adaptation through auto-learning described in [Byu01]. Byun et al. propose several scenarios in which a ubicomp system can learn patterns of use. One of them is “If a user participates in a meeting at 10 am every fourth Monday, the PDS [their proposed system] might learn this behaviour and suggest that the user might have to go to a meeting today (the fourth Monday) at 10 am. However, a more sophisticated level of learning would enable the system to realize when such a notification is inappropriate, for example when the user is on holiday.” [Scenario D in Byu01]. Simple problems of user customisation become complex problems of artificial intelligence. Even if we do eventually have systems which are smart enough to act on behalf of users, system predictability will become an issue.

In discussing possible extension to Context Toolkit (see [Dey01a] and section 6.5), Dey describes the struggle involved in negotiating the tradeoff between supporting a complex situation and providing a simple method for describing a situation. He adds that “while designers who have domain-specific expertise can determine part of the solution [to a ubicomp problem], they will obviously not think of everything that is needed to support individual users. It is the end user who is in the best position to further specialize context-aware application to meet their individual needs.” [Dey01a]. The application of the SICOD framework is potentially a simple way to allow users to represent situations by building ISMs.

The SICOD framework transforms the role of the designer from ‘prescribing use situations’ to ‘developing reusable functionalities that allow users to specify use situations by themselves’. An analogy can be made here with the spreadsheet framework, in which designers provide a library of domain-specific functions but the actual use of these functions is for spreadsheet users with their specific tasks to determine.

With the SICOD framework, users can develop their understanding of use situations specific to them. This understanding can be animated and visualised by building ICMs. With current technologies, we can envisage users making use of Personal Data Assistants (PDAs) to build and maintain ICMs anytime and anywhere. The new tool introduced in Chapter 8, for example, can be used for this purpose. The result is a ubicomp system of devices without a fixed system boundary, capable of better adaptation to use situations and open to evolution.

6.4 Challenges for realising the SICOD framework

In this section, we identify some of the challenges that will have to be met in realizing the SICOD framework. The aim of this section is to identify the key issues for further research rather than to provide or propose immediate solutions.

Synchronisation of the external states with an ICM. There are two related questions. The first concerns keeping the virtual model up-to-date with the real-world situation. For instance, in the drink stock control example, how can the cola count information be updated in the ICM immediately after the action of buying? The second concerns the conversion of real-world analogue states into digital states. This conversion is handled by sensor technologies, but the choice of how frequently this conversion occurs (i.e. the update rate) can affect the responsiveness and integrity of the system (cf. precision, granularity and accuracy of sensor data discussed in [Hon01]). The key question is: how can a user comprehend and manage the conversion process?

Interface for maintaining an ICM. We need simple means for users to create and modify ICMs. This issue is one of the themes of this thesis. The development of WING, EME and DMT (described in later chapters) is targeted towards simplifying the model building activities. The assumption in current EM tools development is that a flat screen-like display and a keyboard is available to the user. However, we could build interfaces of other kinds when new display technologies have evolved. For example, if we combine 3D hologram technology (e.g. [Fre02]) with precision location systems such as the one at AT&T Lab [Sen02], we can create an interface by generating 3D objects and interacting with them. In this way, it might be possible to build an ICM by physically moving virtual 3D objects.

Scalability of ICMs. In some ubicomp scenarios, we are expecting hundreds of ICMs connecting thousands of devices. To implement the SICOD framework on such a big scale it would be necessary to solve problems of reliability and efficiency.

Development of suitable terminology. Introducing the SICOD framework also introduces many terms that will be unfamiliar to end-users. While they are appropriate for academic discussions, terms such as IDM, ISM, ICM and LSD are not easily interpreted by users. We need to develop more user friendly terms that still make the underlying concepts clear.

Integrating with existing technology. So far, we have concentrated on applying the SICOD framework ‘through-and-through’, expecting every device to be designed with an IDM. In the real situation, we shall have to consider an environment which includes other devices that were not designed with the SICOD framework in mind. We surely cannot throw away all current devices and replace them with new devices designed for the SICOD framework.

Safety of customisations. One good thing about traditional hard interfaces is that they prevent users from doing things that are dangerous. The open nature of ICMs might allow users to configure the system so that it exhibits dangerous behaviours. For example, imagine the consequences of connecting a car navigation system to the wrong map.

Security of private observables. Sharing observables between ICMs might sometimes be desirable. However, ICMs might also contain personal information (e.g. credit card numbers) that we might not want to share with others. Methods for attaching scopes and privileges to observables are needed so that we can specify and distinguish these sensitive observables.

Intangible interface. ICMs potentially offer a better conceptual model of the system than a traditional hard interface at the expense of sacrificing physical affordance. For example, in some contexts, the use of an ICM might be an inadequate replacement for the physical buttons on a device; it would be inappropriate to replace all the channel buttons on a TV remote controller with an ICM.

Naming conventions for the observables. How can we make sure that we are referring to the central heating system in our house and not the one next door? We shall need to standardise the naming conventions for observables. One possible solution would be to use conventions similar to URLs for the Internet.

Distributed dependency maintenance. It is relatively easy to implement a centralised model of dependency maintenance. Definitions are stored in one place. The proper order of evaluation of definitions can be determined by classical proposed a client and server model of distributed dependency maintenance [Sun99a]. The result is a distributed prototype version of TkEden called DTkEden. DTkEden is a good tool for studying the issues involved in distributed dependency maintenance.

6.5 Related work

In this section, we describe other related work in the ubicomp research community. In the past five years, the lack of guiding principles for ubicomp development has been directly addressed by many research groups around the world. As a result, many ‘toolkits’, ‘models’ and ‘APIs’ have been developed. Although researchers have used different terms and banners for their work, at some level of abstraction, each has introduced a set of guidelines on how a ubicomp environment should be implemented. We shall refer to each set of guidelines as ‘conceptual framework’ or simply a ‘framework’ for ubicomp. In this section, we briefly review five such conceptual frameworks and compare then with the SICOD framework.

6.5.1 Toolkit framework

The framework introduced in the Context Toolkit [Dey01b] is based on generalising the idea of traditional GUI toolkits. Just as GUI toolkits separate interface concerns from program development, the Context Toolkit framework tries to separate concerns between context acquisition and the use of context in an application. There are five basic software components: context widgets, interpreters, aggregators, services and discoverers. Context widgets hide the specifics of the input devices being used from an application. Their role is similar to that of device drivers. Interpreters convert low-level context data into high-level context information. Aggregators combine context information. Services execute actions based on context information. Discoverers are responsible for maintaining a registry of other software components. Interaction between components is implemented through message callbacks. The main aim of the research is to provide “concepts that make context-aware computing easier to comprehend for application designers and developers” [Dey01b]. Little consideration has been given to users of ubicomp systems. The purpose of interpreters is to provide automatic interpretation of context data. This demands that the designer prescribes the interpretation of context, which is not easy in the dynamic environment of ubicomp.

6.5.2 Layer framework

Tandler [Tan01] tries to separate the concerns of a ubicomp application into five layers. These layers share five data models: the interaction model, the physical model, the user-interface model, the tool model, and the document model. Each layer represents a level of programming abstraction – the module layer contains tailored functionality for specific applications; the generic layer contains common functionality across applications; the model layer contains definitions of the five data models; the core layer provides hybrid implementation of the underlying infrastructure for communications, event handling, device and sensor management, automatic dependency detection and update, etc. An interesting feature of this framework is the use of a declarative description to ensure that the dependency between visualisations and attributes of shared objects is automatically maintained. This framework is specific to OO programming.

6.5.3 Middleware framework

Hong and Landay [Hon01] advocate building middleware similar to the middleware of the Internet to provide communication services for ubicomp applications. In their framework, each application is responsible for implementing standardised communication data formats and protocols. Although this adds complexity, each application can be more independent. The advantages of this framework are the same as the advantages of the Internet infrastructure – it gives freedom in choosing hardware, operating system and programming language.

6.5.4 Blackboard framework

The Blackboard framework uses the blackboard metaphor [Win01]. A blackboard is a communication centre where all communication between applications takes place. Applications can post messages on the blackboard and subscribe to particular classes of message from the blackboard. A blackboard consists of two components: an event heap and a context memory. The event heap maintains short-term message storage. The context memory is a database that provides long-term message storage. The framework is data-centric rather that process-centric. Centralised communication provides opportunities for system integration [Win01].

6.5.5 Trigger-based framework

Huang et al. [Hua99] introduced a framework for ubicomp based on extending the concept of triggers originating from databases. A trigger is a constraint-action pair. The action of a trigger will be executed when the constraint is satisfied. The framework provides automatic translation of a high-level task that is input by the user (e.g. buying a drink) into triggers that are maintained by the system. The main disadvantage of heavy reliance on triggers is that it may lead to state changes that are not easy to comprehend and anticipate. The system behaviour becomes unpredictable.

6.5.6 Overall comparisons

All five conceptual frameworks adopt an engineering approach to ubicomp application – engineers design and implement a ubicomp application; users buy the application and use it according to the user manual. However, as we have discussed throughout this chapter, because our real life environment is very dynamic and user requirements are always changing from situation to situation, the problems of design and use of ubicomp applications cannot be addressed separately. Sometimes, users can also find themselves in the designer's role. It seems unlikely that an engineering approach can address the issues of automation, visibility, connectivity and adaptation discussed in section 6.2 satisfactorily.

The five frameworks discussed in this section are technology-centred. The EM SICOD framework is human-centred. Our emphasis is on using concepts that make the resulting system comprehensible not only by the developers but also by the users. Most of the five frameworks only provide good concepts for the developer to develop ubicomp applications, limiting the scope for flexibility to the design phase of the development. The SICOD framework has the potential to extend system flexibility to users, allowing them to design and customise their own ubicomp environment.

6.6 Summary

In this chapter, we have discussed many visions for ubicomp and identified key issues associated with them. We have proposed a conceptual framework (SICOD) based on EM principles. The potential advantages of the SICOD framework have been illustrated using simple examples. Some of the challenges to be met in realising the framework have also been identified. Finally, we have compared the SICOD framework with other related ubicomp research.

7 Evaluations and Prospects for EM Tools

In this chapter, we shall explore different techniques and tools that can be used to support the activities of EM. Our objectives are to evaluate possible implementations and to discuss the prospects for future development of EM tools. The chapter starts with a discussion of essential characteristics of an ideal EM tool. This is to set the context for evaluations described in later sections. Section 7.2 describes three existing technologies, Java, Excel and Forms/3, as possible EM tool implementations. Section 7.3 evaluates the principal EM tool: TkEden. Section 7.4 describes a new graphical EM tool, WING, which aims to solve some of the issues of TkEden described in section 7.3. Section 7.5 introduces yet another new EM tool, EME, which aims to explore issues that have not been addressed by WING. In section 7.5, we will highlight some prospects based on research described in previous sections.

7.1 The ideal EM tool

One crucial prerequisite for the successful application of EM is the availability of specialised computer-based tools to build artefacts as EM models. From the discussions in the previous chapters, we know that EM can address issues relating to the whole spectrum from system development to use. Chapter 5 is concerned with the activities that are involved prior to or in the preliminary stages of system development. Chapter 3 overviews the application of EM to system development. Chapter 4 gives an account of the combination of system design and use that is characteristic of EM. Chapter 6 considers the possible role for EM in the use of ubicomp systems. Despite these various different emphases, their underlying EM activities can all be captured by the DMF described in chapter 2. The DMF can be viewed as an idealised framework for EM.

It is useful here to summarise the essential characteristics of the DMF we discussed in chapter 2. They will be used as criteria for evaluating different tool implementations for *conceptual support*. The DMF has the following essential characteristics:

- **State-based observation –** In the DMF, observation of states is fundamental to experimentation and understanding of dependencies and agencies in the model. This is the basis for EM.

- **Indivisible stimulus-response patterns –** Each definition in the definition sets represents an indivisible stimulus-response pattern. There is no interruption after a stimulus until a response is given.

- **Subjective agency analysis** - Grouping of definitions and actions into agents is subjective to each agent. Therefore, each agent can have their own view of agencies in the model (cf. subject-oriented programming [Har95]).

- **Universal agency –** In the DMF, not only humans but anything can be viewed as an agent. Human and automatic agents are not necessarily distinguished in the framework. In the process of modelling, the modeller can play different roles in views of different agents in the model – a modeller can be an external observer whose interest is only to observe but not modify the model; a modeller can be an actor who sits within the model as one of the agents; a modeller can also be a director (super-agent) who directs the characteristics of other agents.

- **Concurrent agency –** Typically, a model includes lots of different agents. Each agent can act at the same time as others do. Actions can be performed in parallel.

- **Openness of privileges –** Privileges of agents towards definitions and actions of other agents are not circumscribed and can be changed easily and dynamically during the modelling process (cf. OO encapsulation of data).

- **Evolutionary construal –** Construal of observation, dependency and agency is evolutionary. The modelling process typically starts with only observations without agency. Agencies emerge from experiences obtained in the modelling process. Therefore, a model should be constructed interactively.

An ideal EM tool should implement all of these characteristics of the DMF.

However, we shall discuss in this chapter that, in practice there is still no satisfactory implementation of the DMF.

In addition to the concerns of conceptual support, we shall also evaluate interfaces of the tools. We shall investigate interface techniques that make the process of EM easier to perform. Previous tool research in the EM research group has concentrated on the conceptual support where all efforts are on bringing the DMF close to implementation. Concerns about suitable interfaces for an EM tool are relatively unexplored. There are no fixed criteria for interface evaluation. Therefore, evaluations will be driven by our experience of using the tools. We shall conduct more detail evaluation to our principal tool TkEden at interface level in section 7.3 and discuss ideas explored in development of two new tools WING and EME in section 7.4 and 7.5 respectively.

An analogy can be made with the development of the personal computer. Where conceptual support is concerned, an operating system gives a faithful support for the Von Neumann machine framework. Where the interface is concerned, graphical user interfaces make an operating system easy to use. The success of the personal computer is not possible without a well-developed graphical user interface, and the success of the graphical user interface is not possible without a well-developed operating system. Therefore, we believe that both concerns are very important. We should address both concerns in order to enhance practical applications of EM. By doing so, our main aims are:

- To explore possible options to improve applications of EM in practice (i.e. narrowing the gap between theory and practice).
- To enhance user experience for the process of EM and make EM more suitable to novice users
- To obtain insights on possible options of interfaces for improvement of current tools or development of new tools in the future.

Throughout this chapter, we shall implement a simple DMF model (called Business Deal) by using different techniques and tools. We shall use these implementations as examples used by evaluation of different techniques and tools against the essential characteristics of the DMF. The DMF model of Business Deal is shown in Figure 7.1.

```
Seller’s definitions:
asking_price is 150

Buyer1’s definitions:
budget is 100
interested is Seller:asking_price<=budget

Byer1’s latent actions:
(interested = true) -> Seller:Buyer1Offer is true
(interested = false) -> Seller:Buyer1Offer is false

Buyer2’s definitions:
budget is 1000
interested is Seller:asking_price*0.2<=budget

Buyer2’s latent actions:
(interested = true) -> Seller:Buyer2Offer is true
(interested = false) -> Seller:Buyer2Offer is false
```

Figure 7.1: The Business Deal model in the DMF (note that this is only a conceptual representation)

There are a `Seller` agent and two `Buyers` agents in the model. The `Seller` agent tries to sell his product for a specified asking price. Both `Buyers` have a budget and a criterion to determine if they are interested in making an offer (by actions) for the asking price. In this case, `Buyer1` is interested in making an offer only if the asking price is below or equal to his budget. For `Buyer2`, only if the asking price is below or equal to twenty percent of this budget. Therefore, in the whole model, there is one definition for the `Seller`, and two definitions and two actions for each of the Buyers.

There is no doubt that any of the techniques and tools discussed in this chapter can implement this scenario. However, our main evaluation aim is to investigate the *directness of translation* from the DMF model to particular implementations. By analogy, we can devise OO programs merely by using a procedural programming language (e.g. using C). However, it is far more effective if concepts of OO can be directly supported by the programming language.

7.2 Existing technologies as possible EM tool implementation

In this section, we shall explore three existing technologies as possible EM tool implementations. First, we shall experiment with model-building using a typical OO language: Java. Second, we shall experiment with a spreadsheet application: Excel. Finally, we shall experiment with a visual first-order functional language, Forms/3.

7.2.1 Model-building using Java

Java is a language specially designed for OO programming. However, in this section we shall implement the Business Deal model using Java (JDK version 1.3.1). We represent agents as *objects*, observables as *attributes* and actions as *methods*. Since there is no direct representation for definitions, we have made use of `Observable` and `Observer` classes (from the standard Java library in `java.util`) to simulate the maintenance of dependencies. The mechanism is as follows. Any class that inherits from the `Observable` class maintains a list of objects that implements the `Observer` interface. An `Observable` object can notify `Observer` objects in the list whenever changes have occurred by calling its method `notifyObservers`. This method calls `update` methods of each `Observer` objects in the list as a notification for the change.

As shown in Listing 7.1, `Buyer1` (lines 1-14) and Buyer2 (lines 15-28) agents are represented by classes implementing the `Observer` interface. The major differences between `Buyer1` and `Buyer2` are their criteria of making an offer (lines 9 and 23).

```
1.   class Buyer1 implements Observer{

2.   private double budget;
3.   private boolean interested;

4.   Buyer1(){
5.     budget=100;
6.   }

7.   public void update(Observable o,
     Object arg){
8.     Seller s=(Seller)o;
9.     interested=s.getPrice()<=budget;
10.    if(interested){
11.    s.offer("Buyer1");
12.    }
13.  }
14.  }
```

```
15.  class Buyer2 implements Observer{

16.  private double budget;
17.  private boolean interested;

18.  Buyer2(){
19.    budget=1000;
20.  }

21.  public void update(Observable o,
     Object arg){
22.    Seller s=(Seller)o;
23.  interested=s.getPrice()<=budget*.2;
24.    if(interested){
25.    s.offer("Buyer2");
26.    }
27.  }
28.  }
```

Listing 7.1: Two classes representing Buyer1 (left) and Buyer2 (right) agents.

As shown in Listing 7.2, the `Seller` (lines 29-42) agent is represented by a class inheriting from the `Observable` class. The `Deal` class is a dummy class that contains a program entry point (i.e. the `main`).

```
29.  class Seller extends Observable{

30.  private double asking_price;

31.  public void setPrice(double price){
32.    asking_price=price;
33.    setChanged();
34.    notifyObservers();
35.  }

36.  public double getPrice(){
37.    return asking_price;
38.  }

39.  public void offer(String sellerName){
40.    System.out.println(sellerName +"
     made offer.");
41.  }
42.  }
```

```
43.  class Deal{

44.  public static void main(String
     arg[]){
45.     Seller seller=new Seller();
46.     Buyer1 buyer1=new Buyer1();
47.     Buyer2 buyer2=new Buyer2();

48.     seller.addObserver(buyer1);
49.     seller.addObserver(buyer2);

50.     System.out.println("Price set to:
     10");
51.     seller.setPrice(10);
52.     System.out.println("Price set to:
     150");
53.     seller.setPrice(150);
54.     System.out.println("Price set to:
     999");
55.     seller.setPrice(999);
56.  }
57.  }
```

Listing 7.2: Seller agent (left) and the testing class (right).

A typical scenario of running this program is as follows. The main loop instantiates Seller, `Buyer1` and `Buyer2` (lines 45-47). Then, dependencies between the `Seller` and `Buyers` are made by adding `Buyers` to the list of `Observers` in the `Seller` (lines 48-49). After that, we can then test the responses of `Buyers` by setting different prices in the Seller. For example, the program calls `seller.setPrice(150);` (line 53). This sets the price in the Seller to 150 and notifies `Buyer1` and `Buyer2` (lines 31-35). The `update` methods in `Buyer1` (line 7-14) and `Buyer2` (line 21-28) are called. By their criteria, `Buyer1` does not make offer but `Buyer2` does. The output of running this programming is shown in Figure 7.2 below.

```
C:\>java Deal

Price set to: 10
Buyer1 made offer.
Buyer2 made offer.
Price set to: 150.
Buyer2 made offer.
Price set to: 999.
```

Figure 7.2: Output of running the Business Deal model in Java

Building an ‘EM model’ by using Java is difficult. The first very noticeable problem is we cannot build the model interactively – every time we change or add something to the model, we need to make a compilation and restart the test from the beginning. We cannot add or modify any definition or action during the execution of the Java model. This is not in keeping with the evolutionary characteristics of the

DMF where by modification of the model should be conducted interactively as one of the agents.

User interaction with the model has to be circumscribed before executing the model. So the user cannot be an agent within the model (cf. universal agency in the DMF). In fact, interaction between other agents in the model also has to be circumscribed.

We could represent observables by attributes which contain states of the agents. However, the dependencies between observables from different agents have to be maintained by message passing (cf. line 34 and line 9). In other words, indivisible stimulus-response patterns cannot be represented faithfully in a Java model. Besides, the focus on states is easily inhibited by paying too much attention to message passing between classes (cf. state-based observation in the DMF).

The concurrency of actions of three agents in the Java model is replaced by sequential message passing (cf. actions in lines 11 and 25). We can implement concurrency explicitly in Java but this involves setting up the mechanism manually as part of the program. What we really need is the style of concurrency that exists in the DMF (cf. concurrent agency in the DMF).

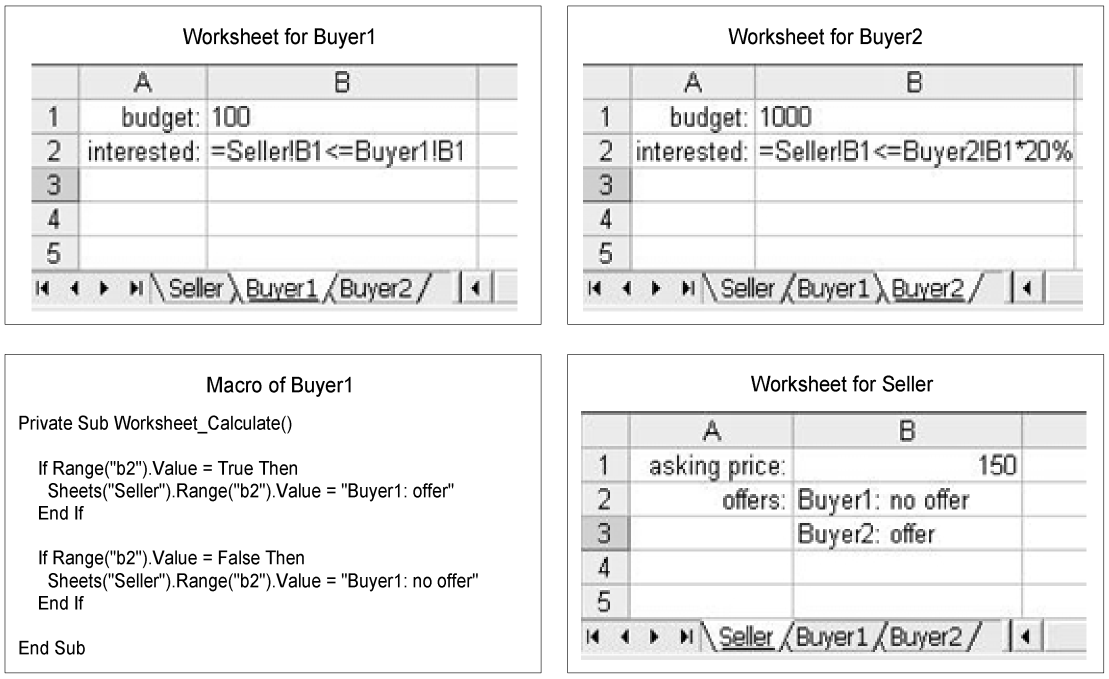
In OO philosophy, it is better to declare attributes of a class as `private` (i.e. data encapsulation). However, in the DMF every observable of an agent is accessible for other agents. This can be easily done by declaring all attributes of a class as `public`. It violates OO philosophy but facilitates the DMF philosophy (cf. openness of privileges in the DMF).

All the above observations lead us to conclude that it is difficult to use Java to build an EM model. The representation of agents as objects, observables as attributes and actions as methods seem to be direct but in fact it is not. The translation from the DMF model to the Java model is neither direct nor complete.

7.2.2 Model-building using Excel

In this section, we describe building the Business Deal model by using the spreadsheet application Excel (version 2002). In Excel, spreadsheets are called ‘worksheets’. Worksheets are grouped together as a ‘workbook’. We can represent observables as cells, dependencies as formulae, actions as macros (in Visual Basic), agents as aggregations of worksheets and macros, and the entire model as a workbook.

Figure 7.2 shows screen captures of the model. The Seller agent represented by a worksheet is at the bottom-right screen capture. The current asking price is in the cell B1. Two Buyers are two worksheets (in formulae view) at the top of the figure with their budget and criteria of making an offer. We can see in the figure how each worksheet can refer to cells in another worksheet by using the symbol ‘!’.



Worksheet for Buyer1

	A	B
1	budget:	100
2	interested:	=Seller!B1<=Buyer1!B1
3		
4		
5		

Worksheet for Buyer2

	A	B
1	budget:	1000
2	interested:	=Seller!B1<=Buyer2!B1*20%
3		
4		
5		

Macro of Buyer1

```
Private Sub Worksheet_Calculate()

   If Range("b2").Value = True Then
     Sheets("Seller").Range("b2").Value = "Buyer1: offer"
   End If

   If Range("b2").Value = False Then
     Sheets("Seller").Range("b2").Value = "Buyer1: no offer"
   End If

End Sub
```

Worksheet for Seller

	A	B
1	asking price:	150
2	offers:	Buyer1: no offer
3		Buyer2: offer
4		
5		

Figure 7.2: The Business Deal model in Excel

At the bottom-left screen capture of the figure shows an ‘event macro’ called `Worksheet_Calculate` attached to Buyer1 worksheet. Every worksheet in Excel has this macro by default. Any code specified in this macro is automatically executed whenever any cell value in the worksheet is updated. As shown in the figure, it contains actions of Buyer1 – that changes the value of Seller’s B2 cell to make an offer. A similar macro is also attached to Buyer2 which is not shown in the figure.

The translation from the DMF model to a spreadsheet model is more direct than to the Java model. This is mainly for two reasons: dependencies specified by formulae are automatically maintained (cf. indivisible stimulus-response pattern in the DMF) and we can incrementally build the model on-the-fly (cf. evolutionary in the DMF).

By using the grid layout, a spreadsheet visualises all states corresponding to cells (cf. state-based observation in the DMF). By changing from one worksheet to another, the user can play different roles as different agents in the model (cf. universal agency in the DMF).

Excel provides no support for specifying access privileges for the cells. Every worksheet is free to refer to cells in other worksheets (cf. openness of privileges in the DMF). We have also demonstrated by experiments that macros (actions) are executed sequentially in Excel. Therefore, agents cannot act concurrently (cf. concurrent agency in the DMF).

To conclude, we can use Excel as an EM tool. The translation from the DMF model to the spreadsheet model is easy. There are direct correspondences in mapping agents as worksheets, cells as observables, definitions as formulae and actions as macros. However, we have also notice two discrepancies. First, the agents cannot behave concurrently. Second, observables are normally referenced by grid co-ordinates rather than by user-given names (we can give an alias to cells but this involves extra effort). In order to use Excel as an EM tool, we also need to extend its available cell data types; it would be especial helpful to introduce graphic data types.

7.2.3 Model-building using Forms/3

Forms/3 is a first-order functional visual programming language based on spreadsheet styles of cells and formulae [Bur01]. However, there are two main differences between Forms/3 and a conventional spreadsheet. First, in Forms/3, cells are represented as rectangles which are positioned manually without a conventional grid layout. The value of a cell is visualised within the cell. The name and a *formula tag* of the cell are located at the bottom of the cell. Clicking the formula tag brings up a dialogue in which a formula can be entered (see left-hand-side of Figure 7.3). Second, in conventional spreadsheet, we can change the value of a cell by using procedural macros. In Forms/3 the value of a cell is solely defined by its formula (this is termed the ‘value rule’). Hence, it does not support procedural macros. This is consistent with one of the main aims of Forms/3: to stay within the functional paradigm of programming.

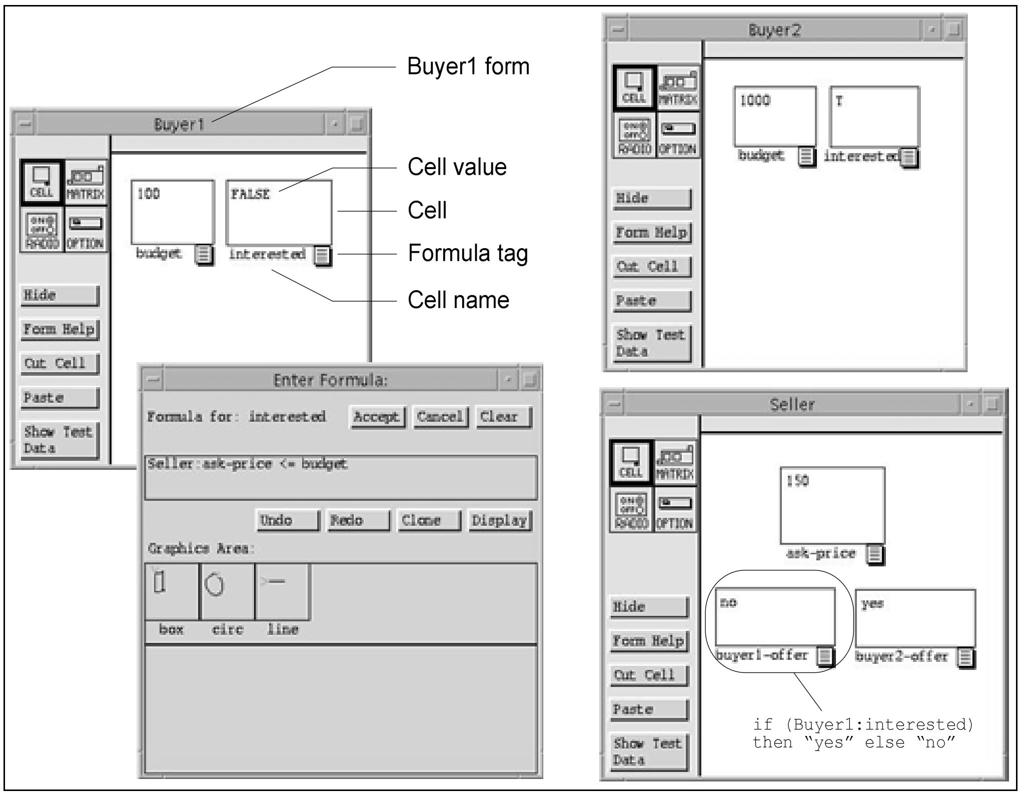


Figure 7.3: The Business Deal model in Forms/3

Figure 7.3 shows a Forms/3 implementation of the Business Deal model. We can represent agents as *forms* (similar to worksheets in Excel), observables as cells and dependencies as formulae. Because of the value rule, there is no way we can specify actions directly. Therefore, we cannot specify the 'offer-making' actions in two Buyers. The workaround was to translate actions into definitions. For example, instead of specifying the actions in Buyer1, we enter an observable in the Seller with the formula: `if (Buyer1:interested) then "yes" else "no"` (bottom-right region of Figure 7.3). However, by doing this we have changed the semantics of the model – in the DMF model `offer` is *initiated* by the Buyers but in the Forms/3 model `offer` is *perceived* by the Seller.

By using Forms/3, we can perform state-based observation, specify indivisible stimulus-response patterns and construct models evolutionarily (just like Excel). Cells can also be grouped to reflect agencies as forms. However, the inability to specify actions makes translation from the DMF model to Forms/3 model difficult and sometimes impossible without changing the semantics of the model. This limitation of using Forms/3 as an EM tool highlights the difficulty in building an EM model in using pure functional constructs.

7.3 TkEden: the principal EM tool

Our principal tool TkEden and its variants (e.g. ttyEden, Eden, DTkEden) have been used for EM for more than ten years. Over this period, the research interests of EM itself have broadened from originally being solely concerned with interactive graphics [Bey88b] to definitive programming [Yun93], and from definitive programming to system development. To justify our claims that EM is not only theoretical but also practical, the expansion in conceptual scope of EM needs to be accompanied and supported by developing practical tools. TkEden serves very well as a research prototype for trying and exploring different research ideas – several hundreds of models have been built and more than seven different definitive notations have been integrated into the tool. However, in this section, we shall take a critical view of TkEden in use, and identify its limitations as a tool for applying EM in practice.

In a TkEden script, observables are represented by definitive variables. Dependencies are explicitly specified by formulae. Actions are represented by procedures. Agents can be represented by the grouping of definitions and actions in the script, suitability annotated with comments. This is illustrated in Listing 7.3, which shows the script of the Business Deal model. The primary input mechanism is to write a script of definitions in a text input window and then introduce the script into the model by clicking the ‘Accept’ button of the ‘input window’ (cf. Figure 2.7 in chapter 2).

```
/*** seller ***/
seller_asking_price is 150;

/*** buyer1 ***/
buyer1_budget is 100;
buyer1_interested is  seller_asking_price <= buyer1_budget;

proc buyer1_action1: buyer1_interested{
    if(buyer1_interested) seller_buyer1Offer is 1;
}
proc buyer1_action2: buyer1_interested{
    if(!buyer1_interested) seller_buyer1Offer is 0;
}

/*** buyer2 ***/
buyer2_budget is 1000;
buyer2_interested is seller_asking_price <= buyer2_budget*0.2;

proc buyer2_action1: buyer2_interested{
    if(buyer2_interested) seller_buyer2Offer is 1;
}
proc buyer2_action2: buyer2_interested{
    if(!buyer2_interested) seller_buyer2Offer is 0;
}

proc monitor:seller_buyer1Offer,seller_buyer2Offer{
    writeln("buyer1:",seller_buyer1Offer," buyer2:",seller_buyer2Offer);
}
```

Listing 7.3: TkEden script of the Business Deal model

We shall first evaluate the degree of conceptual support for EM provided by TkEden by investigating the directness of the translation from a DMF model to a TkEden model. After that, we shall evaluate its interface to investigate the ease-of-use aspects.

7.3.1 Evaluation of conceptual support

Conceptually, the TkEden interpreter provides a high degree of interactiveness, so that the modeller can gradually build the model on-the-fly (cf. evolutionary construal in the DMF). Any changes are immediately reflected in the model. The modeller can use a trial-and-error approach to experiment with the model. The states of observables can be queried at any time during the modelling process, primarily by using a "`writeln();`" statement (cf. state-based observation in the DMF). The dependencies specified by using formulae are automatically maintained (cf. indivisible stimulus-response patterns in the DMF). Every observable is in a global name space, and therefore can be accessed freely from anywhere in the model (cf. openness of privileges in the DMF).

The only major discrepancy between the DMF model and the TkEden model is in the representation of agency. TkEden has no direct mechanism to represent agents. As shown in Listing 7.3, we can use comments to annotate the agents (e.g. "`/*** buyer1 ***/`" in line 3) and use naming conventions to specify the ownership of observables (e.g. `buyer1_budget` in line 4). However, these ways of representing agency can only be interpreted by human agents. We need some built-in agency representation mechanisms that are interpretable both by human and automatic agents.

In addition, concurrent actions can only be executed one by one sequentially (cf. concurrent agency in the DMF). Modelling a real-time system which has agents acting concurrently becomes very difficult. This is illustrated in the Dishwasher model described in chapter 3. In the Dishwasher model, a clock agent is introduced to simulate concurrency. However, all actions are still executed sequentially. Efforts have been made in the distributed version of TkEden (DTkEden) to allow concurrency through *actions* [Sun99a]. However, there is still no mechanism to allow *dependencies* to be maintained concurrently.

Another issue is that TkEden is a hybrid interpreter that allows both procedural variables and definitive variables to be used. The modeller has to distinguish a procedural assignment from a definition. For example, "`a = b+c;`" and "`a is b+c;`" are very different. In the former, `a` is a procedural variable. Only the result of the calculation from `b+c` is assigned to `a`. This is a one-off calculation. Value changes to `b` or `c` do not cause the value of `a` to be changed. In the latter, `a` is a definitive variable. The value of `a` will be recalculated every time the value of `b` or `c` changes. In the DMF, we can have only definitive variables, and assignment such as "`a = b + c;`" is replaced by "`a is eval(b+c);`" where `eval()` returns the 'current value' of "`b+c`". The use of procedural variables in TkEden is sometimes confusing, and it violates the characteristics of the DMF.

7.3.2 Evaluation of the interface

In this subsection, we evaluate TkEden for issues of ease of use. The result of the evaluation can give useful insights into how to build better interfaces for EM tools. With reference to the analogy with the development of the personal computer mentioned earlier, TkEden's interface is like a command prompt interface to an operating system – it provides access to all functionalities of the system but it is *not* the best access method for end-users when compared to graphical interfaces. In addition, some characteristics of the DMF can be more easily realised at the interface. For example, the process of subjective agency analysis can be facilitated by allowing the modeller to organise definitions and actions visually. This also supports the conception of the model as an artefact.

TkEden has two features that are related to ease of use. Firstly, it promotes high-level, task-specific programming style that allows the modeller to construct a model interactively by trial-and-error. This is a common feature in most end-user programming tools (cf. [Nar93]). Secondly, the Eden interpreter that is kernel of TkEden provides dynamic typing, so that definitive variables need not be declared before use. Apart from these two features, TkEden has many issues that need to be addressed in relation to ease of use. We shall discuss them one by one as follows:

- **The definitive script is not the interface –** The use of definitive script in TkEden is limited to model representation. It is *not* the interactive model itself – we cannot change the state of the model by editing the script directly. A definitive script has to be fed into TkEden by using the input window. The actual model resides in the computer memory and can only be accessed through the 'input window'.

- **Lack of default visualisation –** TkEden enables the modeller to build customised visualisations by using Donald and Scout. However, there are typically many elements of a definitive script that are only accessible to the modeller in textual form and typically only as a result of action on the modeller's part. For instance, such elements typically include the values and relationships associated with scalar variables that underlie a geometric model. By default the TkEden interpreter only presents the current values and definitions of such variables on request in response to queries of the form "`writeln(x);`" and "`?x;`". This is unlike a spreadsheet, where all the current values are displayed all the time, and dependencies between cells can be displayed graphically on demand. EM advocates the idea of modelling by building of artefacts. Supplying a default visualisation for all the elements of a model could give the modeller a more concrete feeling of building a model as an artefact.

- **Lack of mechanism to organise definitions** – There is no feature in TkEden to organise definitions and actions in accordance with user preferences. Definitions and actions can only be viewed in the predetermined ways that the interpreter allows (e.g. Eden, Scout and Donald variables can be listed in alphabetical order).

- **Difficulties in renaming observables –** TkEden provides no feature for renaming an existing observable. This involves renaming all the occurrences of the observable in formulae of all definitions in the model. This makes observables 'sticky' because if the modeller wants to rename an observable, she has to redefine all other observables that depend on it.

- **Invisible progress of actions** – In general, the modeller does not need to know the progress of actions. However, when it comes to understanding the behaviour of a model, it is useful if the modeller can inspect (for example) the order of action execution. This also makes debugging easier.

- **Difficulty in annotating definitions** – Comments can be added to definitive scripts. However, when a script is interpreted, all the comments are discarded by the interpreter. Comments help to understand the model so it would be useful to be able to attach comments to definitions even after they have been interpreted. One of the difficulties encountered in annotating scripts is that a comment may apply to a group of definitions, but there are in general many different useful ways of grouping definitions (cf. subjective agency analysis).

- **Notation syntax inconsistency –** In TkEden, different definitive notations have different syntax. For example, every statement in Eden has to end with a semicolon but in Donald every statement has to be terminated with a carriage-return. Another example is that the user does *not* have to declare variables in Eden but does have to declare them in Donald and Scout. It is difficult and confusing for the novice to learn several different syntactic conventions.

- **Complex syntax for actions –** The syntax for specifying actions is sometimes very complex. An example will be given in discussing the EME tool in subsection 7.5.2.

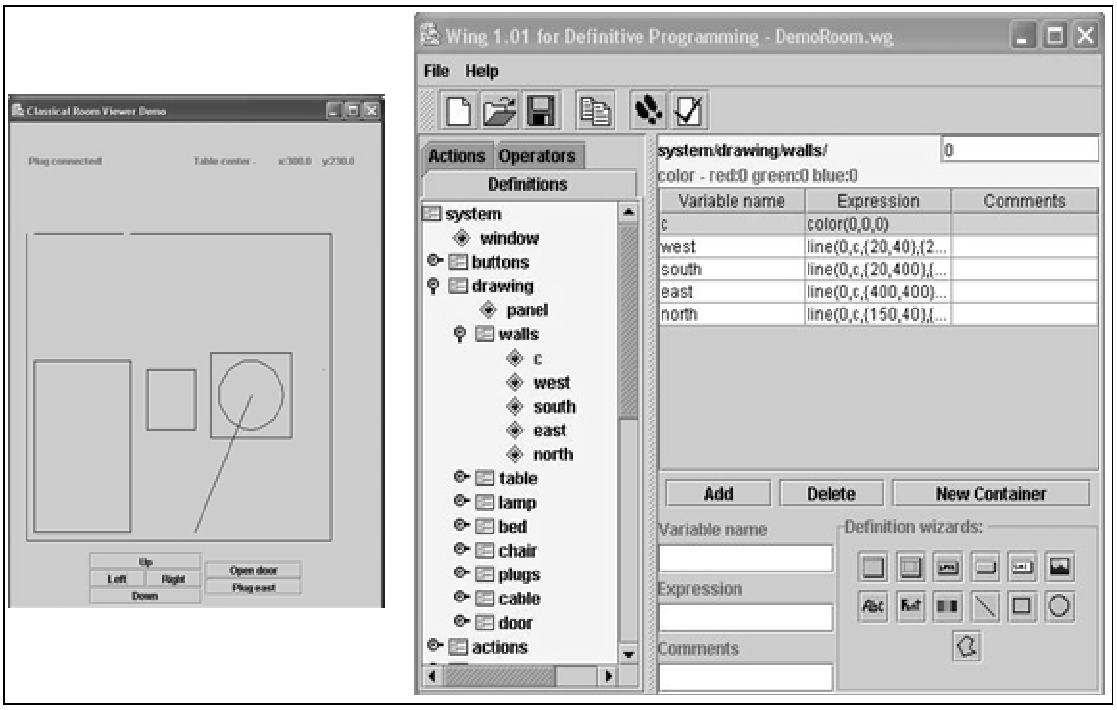
The developments of WING and EME described in the next two sections aim to address some of these issues.

7.4 WING: a graphical EM tool

WING (a WINdowing and Graphics tool) was originally conceived as the implementation of a definitive notation for windowing and graphical objects. However, during its development, the focus of attention shifted to its capabilities as a full EM tool. WING provides 15 data types in four categories, namely *basic*, *windowing*, *graphics* and *vector* data types. The modeller can also define new data types and operators. In this section, we shall only discuss features of WING that address some of the issues for TkEden identified in the last section. For more details, the reader is directed to [Won98]. Appendix F contains a technical overview of WING.

7.4.1 Organising definitions

Figure 7.4 shows the main user interface of WING loaded with a model of a room. The interface resembles a file explorer – definitions are organised within *containers* in much the same way that files are organised within directories. Therefore, conceptually, agents can be represented by containers (cf. the lack of agency representation and mechanism to organise definitions in TkEden). We call this a *directory-like* *organisation* of agents. Clicking a particular container at the left-hand-side of the interface makes all definitions in the container appear at the right-hand-side of the interface. Definitions are displayed as rows of spreadsheet-like cells and can be edited directly by double clicking the cells (cf. definitive script is not interactive in TkEden). The values of observables can be viewed by clicking the corresponding cells (cf. the lack of default visualisation in TkEden). Comments can be added to each definition and these are recorded is displayed by WING (cf. difficulty in annotating definitions in TkEden). This is illustrated in figure 7.4 below.



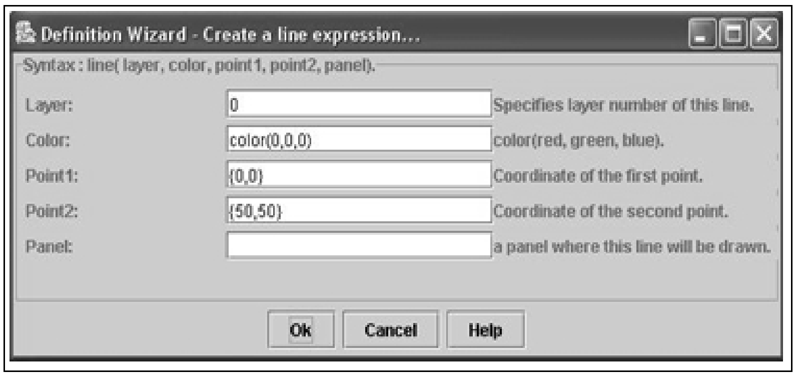
system/drawing/walls/ 0

color - red:0 green:0 blue:0

Variable name	Expression	Comments
c	color(0,0,0)	
west	line(0,c,{20,40},{2...	
south	line(0,c,{20,400},{...	
east	line(0,c,{400,400}...	
north	line(0,c,{150,40},{...	

Figure 7.4: WING (right) loaded with the Room model

In addition, WING has a method of alleviating the syntactic obstacles to entering definitions. At the bottom-left area of the interface shown in Figure 7.4, there is a set of buttons for helping the user to enter definitions. Clicking a button brings up a *definition wizard*. Figure 7.5 shows a definition wizard for specifying a line definition. A definition wizard provides a reminder of the syntax for a definition and an explanation of the significance of each parameter. By filling the form, the definition will be automatically generated. This is particularly useful for the novice modeller because she does not have to remember the syntax of all the different types of construct within different definitive notations (cf. notation syntax inconsistency in TkEden).


Definition Wizard - Create a line expression...

Syntax : line(layer, color, point1, point2, panel).

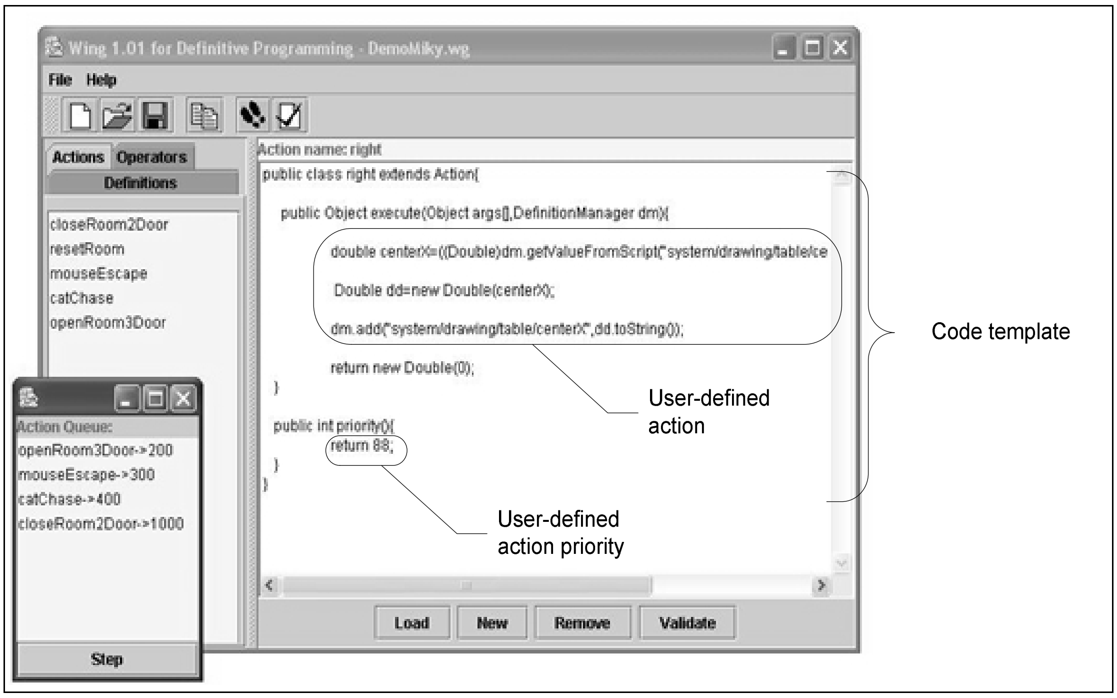
Layer:	0	Specifies layer number of this line.
Color:	color(0,0,0)	color(red, green, blue).
Point1:	{0,0}	Coordinate of the first point.
Point2:	{50,50}	Coordinate of the second point.
Panel:		a panel where this line will be drawn.

Figure 7.5: A Definition Wizard of specifying a line definition

7.4.2 Specifying actions

Actions are specified by using Java. WING provides a *code template* as the skeleton of an action. The user is only required to fill in actions to manipulate definitions in the model as shown in Figure 7.6. Each action in WING has a priority number which can be changed to control the order of action execution. The actions can be compiled and dynamically linked to the model by clicking the ‘Validate’ button. Dynamic linking of actions to the model is important because it allows user to specify actions on-the-fly without sacrificing the interactiveness of the model.

The execution of actions triggered by interaction with the model can be monitored by bringing up the action queue window as illustrated in the bottom-left window of Figure 7.6. In the figure, we can see four actions waiting in the action queue. The number at the right of each action name is the priority number. The modeller can step through the execution of actions by clicking the ‘step’ button at the bottom of the action queue window. This feature addresses the issue of invisible progress of actions in TkEden.



```
closeRoom2Door
resetRoom
mouseEscape
catChase
openRoom3Door
```

```
Action Queue:
openRoom3Door->200
mouseEscape->300
catChase->400
closeRoom2Door->1000
```

```
Action name: right
public class right extends Action{

  public Object execute(Object args[],DefinitionManager dm){

        double centerX=((Double)dm.getValueFromScript("system/drawing/table/ce

        Double dd=new Double(centerX);

        dm.add("system/drawing/table/centerX",dd.toString());

        return new Double(0);
  }

  public int priority(){
        return 88;
  }
}
```

Figure 7.6: Action specification in Java (middle) and stepping for action queue (bottom-left window)

7.4.3 Higher-order definitions

WING also explored the possibility of introducing *higher-order definitions* to an EM model by providing a 'like' operator (cf. the 'like' operator in Excel). A higher-order definition is a definition whose *formula* depends on the formula of another definition. In TkEden, formulae in definitions can only depend on the *values* of other definitions. To illustrate the concept, we can refer to the model of a simple two-layer perceptron (in the context of neural network). In this case, we are interested in the shape of the three neurons. As shown in the left-hand-side of Figure 7.7, originally they are all oval in shape. The task is to change them into a rectangular shape. By using the 'like' operator, we can specify their shapes as follows:

```
centreA is {130,90}
centreB is {130,290}
centreC is {330,190}
neuronA is oval(0, color(0,0,0), centreA, size, panel)
neuronB is like(neuronA,1, centreB, centreA)
neuronC is like(neuronA,1, centreC, centreA)
```

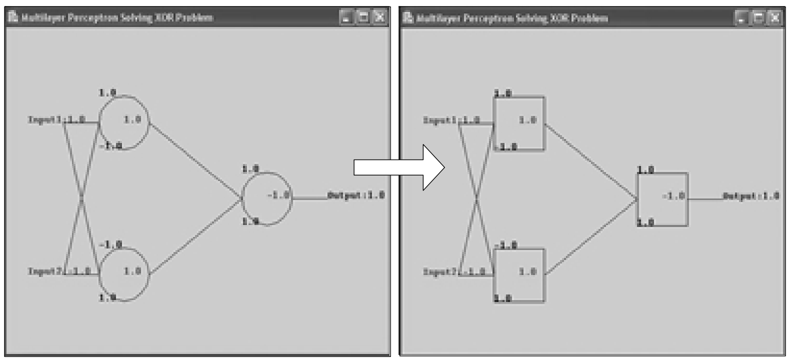


Figure 7.7: Neurons in oval shape (left) are transformed to rectangular shape (right) by higher-order definitions

The centre positions of the neurons are specified by `centreA`, `centreB` and `centreA` respectively. As we can see in the above definitions, the shape of neuron B and C is dependent on the shape of A. For example, in the definition of neuron B: "`like(neuronA,1, centreB, centreA)`" means that the formula of B is same as A *except* that the first occurrence of symbol `centreA` is replaced by `centreB` (because B is in a different position). To change all the shapes to rectangles, we only need to change formula of A from "**`oval`** `(0, color(0,0,0), centreA, size, panel)`" to "**`rect`** `(0, color(0,0,0), centreA, size, panel)`".

WING has illustrated that simple higher-order definition can increase the expressive power of definitions. However, we need to bear in mind that higher-order definitions are more difficult to understand than ordinary definitions. We need to find a balance between expressiveness and comprehensibility in the future development of higher-order definitions in EM tools.

7.4.4 Evaluation

The development of WING has provided possible solutions to most of the issues of TkEden at the interface level. In addition, it has also explored some new ideas such as definition wizards, dynamic compilation of Java code for extension, and higher-order definitions. However, the design and implementation of WING also raises a number of issues:

Firstly, there is no concurrency of agents. We can represent agents as containers. Ideally, actions with the same priority need to be executed in parallel (cf. concurrent agency in the DMF) but in WING they are executed sequentially. A similar issue arises in TkEden.

Secondly, we cannot simulate a clock. In TkEden, we can simulate concurrency by time-sharing regulated by a clock agent (cf. the use of the `todo` construct Dishwasher model). The implementation of the dependency maintainer in WING does not allow a similar clock agent to be defined. One possible solution is introducing a built-in clock agent in a separate program thread.

Finally, the use of Java language for action specification makes WING more extendable. However, it is difficult for novice modellers to specify functions and actions if they do not have experience in Java programming. We have found that it is more difficult to specify actions in WING than in TkEden. One of the aims for developing EME, to be described in the next section, is to explore the possibility of designing a simpler language for specifying actions.

7.5 EME: a tool with expressive variable referencing

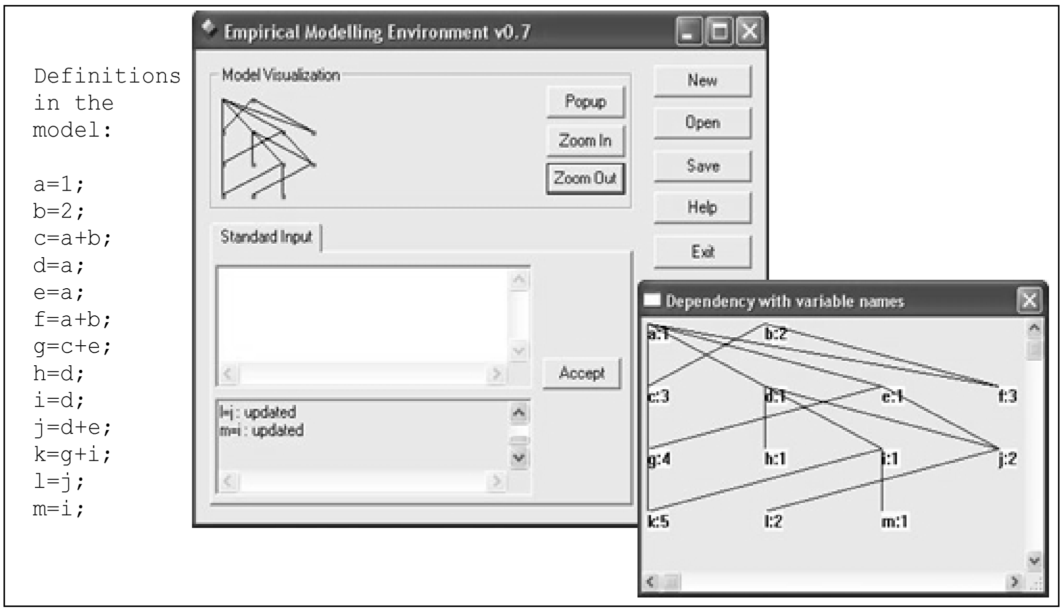
The primary aim of building the Empirical Modelling Environment (EME) was to develop a more expressive language for variable referencing. This is motivated by the need for simpler methods of specifying complex definitive script in TkEden. The variable referencing techniques also lead to alternative techniques for organising definitions. Though EME only supports simple data types, it addresses potential solutions to problems of script construction and management that are relevant to all definitive notations. Appendix G contains a technical overview for the tool.

7.5.1 Entering definitions

Like TkEden, EME accepts textual definitive script from an input window. But unlike TkEden, procedural variables and assignments are not allowed. The typical example of “a is b plus c” can be specified as:

```
a = b + c;
```

The symbol ‘`=`’ here has the same semantics as the symbol ‘`is`’ in TkEden. The semicolon at the end marks the end of a definition. Figure 7.8 shows the interface with some definitions in the model.



```
Definitions
in the
model:

a=1;
b=2;
c=a+b;
d=a;
e=a;
f=a+b;
g=c+e;
h=d;
i=d;
j=d+e;
k=g+i;
l=j;
m=i;
```

```
l=j : updated
m=i : updated
```

Figure 7.8: EME interface (left) and model visualisation (right)

As shown in the figure, the model is visualised as a hierarchical dependency graph. There are two views of the same graph. The smaller one (on the left in the figure) gives an overall visualisation of the entire model. The bigger one can be zoomed to display a particular part of the graph. The symbols in the graph are ordered in levels so that a variable is only dependent on the variables at levels on top of it. The integer value at the right of each symbol is the current value of that symbol. This feature of EME addresses the issue of lack of default visualisation of the model in TkEden. It gives the modeller a more concrete feeling of building an artefact. This feature has been further enhanced in the development of the DMT to be described in the next chapter.

7.5.2 Variable referencing

One of the unique features of EME is its variable referencing mechanism. This gives the modeller more control of how a variable reference in a definition will be evaluated. The format of a variable is shown in the following example:

```
table\lamp\bulb
```

The backslashes in this variable are context separators. In EME, variables are specified by *variable expressions*. A variable expression may contain alpha-numeric characters, back slashes, and three kinds of operators designated by angle brackets, square brackets and curly brackets.

The angle brackets < > specify that the enclosed expression will be evaluated and translated into alpha-numeric characters and then substituted into the variable expression to form a variable reference. It is a one-off translation. The square brackets `[]` specify that the enclosed expression will be evaluated and translated into alpha-numeric characters every time the definition is evaluated. Finally, the curly brackets { } specifies that we need the value of the enclosed expression only. To help understand the semantics of those brackets, here is an example. Suppose we already have the following definitions in the model:

```
i = 10;
x\10 = 99;
x\i = 88;
```

The examples in Figure 7.9 show that we can use different brackets to control exactly when a variable expression is evaluated.

User input	**Definition stored in the symbol table**	**Evaluation sequence**
`y = x\10;`	`y = x\10`	`x\10` → `99`
`y = x<i>;`	`y = x\10`	`x\10` → `99`
`y = x[i];`	`y = x[i]`	`x[i]` → `x\10` → `99`
`y = x\i;`	`y = x\i`	`x\i` → `88`
`y = {x\10};`	`y = 99`	`99`

Figure 7.9: Examples of using different variable referencing techniques

We can identify three different uses of this technique:

1) Creating a virtual list – We can create a virtual list by using this technique. For example, the following definitions specify a list of three integers:

```
a\2 = 20;
a\3 = 30;
x = a[i];
i = 1;
```

After introducing these definitions, the value of `x` will be changed according to the value of the index `i`.

2) Syntactical organisation of agents – We can specify the ownership of variables by integrating the agent name into the variable name (cf. the concept of ‘virtual agent’ in DTkEden [Sun99a]). For example, the following three definitions specify the exam marks for three students:

```
tommy\marks = 100;
annie\marks = 40;
jody\marks = 10;
```

We can refer to each student's marks by the variable expression `<currentStudent>marks` where `currentStudent` contains a name of a student.

3) Simplifying syntax of action specification – This way of referencing definitive variables is neater than the way used in TkEden. For example, here is an action taken from the TkEden timetabling model. Its purpose is to link cell contents to a function `f(x)`.

```
proc linkCells{
  auto i;
  for(i=1;i<=slotsperday*noofdays;i++){

    execute(
   "~slotsTable_" // ~slotsTable_myList[i] // "_myCaption is ~f(" // str(i) // ");");
  }
}
```

The EME implementation of the same action would be:

```
action linkCells{
  i=1;
  repeat(i<=slotsperday*noofdays){

    slotsTable<slotsTable_myList<i>><"myCaption"> = f({i});

  i={i+1};
  }
}
```

If the value of `i` is 1, the definition in the above loop is equal to:

```
   slotsTable\cell1\myCaption = f(1);
```

This is arguably more understandable than the TkEden implementation. The use of keyword `repeat` will be explained in the following subsection.

7.5.3 Procedural macro

EME provides two procedural macros to automate the variable defining process. They are a conditional if macro and a looping macro. The syntax of the conditional if is:

if (logical expression) {lines of definitions} **else** { lines of definitions}

The syntax of the looping macro is:

repeat (logical expression){lines of definitions}

The loop repeats until the logical expression evaluates to be false. To illustrate, we can define five windows by the following definitions. The result is shown in Figure 7.10.

```
i=1;
title="hello!";
repeat (i<=5) {
x1<i> ={10+i*50};
y1<i> ={10+i*20};
x2<i> ={150+i*50};
y2<i> ={400+i*20};
w<i> = window(title,x1<i>,y1<i>,x2<i>,y2<i>);
i = {i + 1};
}
```

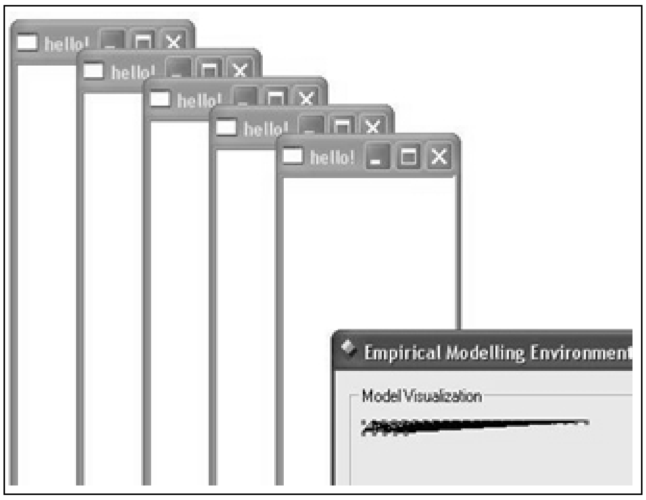


Figure 7.10: Specifying multiple windows

The window at the bottom right of Figure 7.10 illustrates how a simple iteration in EM can generate a very dense set of dependencies.

7.5.4 Complex interface widget

As illustrated in Figure 7.10 and 7.11, EME, like WING, includes features for defining windows and buttons. However, in the development of EME, we have also experimented with the introduction of complex interface widgets. For example, the following two definitions define the a simple spreadsheet grid with 8 by 10 cells depicted in Figure 7.11 (other numbers in the definitions are for specifying the size of the display area).

```
wg = window("grid test",10,10,400,400);
grid1 = grid(8,10, 10,10, 300,200, wg);
```

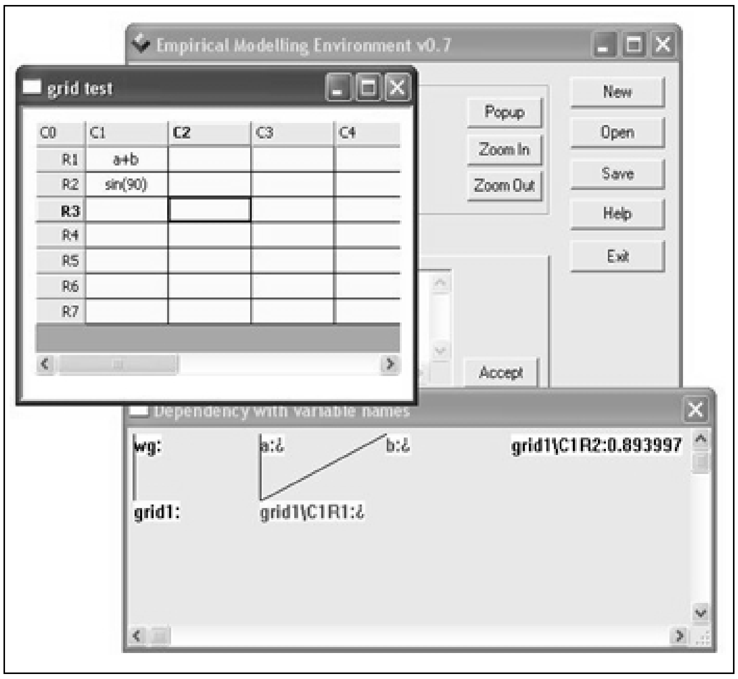


Figure 7.11: A built-in spreadsheet widget

The modeller can enter definitions into the cells of such a grid. For example, we have entered a definition `a+b` in the location of column 1 and row 1. The actual definition created is `grid1\C1R1 = a+b;`. The dependencies involved are shown in the model visualisation window at the bottom of the screen capture.

The spreadsheet grid illustrated in Figure 7.11 is actually an ActiveX control object obtained from the Web. This has demonstrated that we can link interface widgets created by using other programming language to an EM tool to facilitate the modelling of software systems with graphical user interfaces (cf. the interactor widgets in Penguims [Hud94]).

7.5.5 Evaluation

Throughout the development of EME, we have experimented with several new ideas that relate to improving interfaces to EM tools. In this subsection we give a brief evaluation of each of the experimental features we have introduced into EME:

- The *graphical visualisation* of an EM model can give an overview of all observables and dependencies within the model. This can help the modeller to comprehend the model. However, the visualisation supplied by EME is still not satisfactory. For example, the layout of the symbols at each level of the hierarchy is random. When there are a lot of dependencies between two levels, edges cross each other so that it is very difficult to see which symbol is dependent on which symbol. There is no mechanism in EME to rearrange the symbols to reduce edge crossings.

- The new technique of *variable referencing* has the potential to increase the expressiveness of the definitive script whilst at the same simplifying its syntax. However, more experiments need to be conducted to justify these claims.

- *Procedural macros* provide a simple way to automate some repetitive interactions with the model. At this stage, there are only two types of macro. We need to design more macros to accommodate a variety of user interactions. Examples are copy and paste macros and file manipulation macros.

- We have explored the possibility of introducing *complex interface widgets* into an EM tool. Further investigation is needed to justify whether this is a suitable method for extending the tool. It would be even more useful if we could design interface widgets using primitive drawing definitions (cf. the design of the Dishwasher interface in chapter 3). This gives the user much more definitive control over the functionality and appearance of an interface.

7.6 Prospects

The research described in this chapter suggests that it is difficult if not impossible to build an EM model by using OO languages or pure functional languages. Neither paradigm can fulfil the demands of supporting essential characteristics of the DMF. In particular, the concept of EM agents is neither supported by OO nor functional paradigms. In fact, the DMF has a very different philosophy from OO or functional paradigms – the OO paradigm advocates data encapsulation but the DMF advocates free access to data; the functional paradigm rejects procedural access to variables but the DMF includes procedural access (actions) to variables as essential to the representation of agents.

We have found that the degree of conceptual and interface support for EM that Excel with macros supplies is similar to that supplied by TkEden. However, TkEden is still a better choice than Excel. The main reason is that it supports a variety of different special-purpose definitive notations. Every definitive notation extends TkEden to support many new data types and different ways of specifying definitions. For example, by using Donald and Scout we can build definitive visualisations that cannot be easily built by using Excel. From a research perspective, since we can access the source code of TkEden, we can always incorporate and rebuild TkEden to reflect new ideas.

Where conceptual support for EM is concerned, the main limitation of existing EM tools is the lack of a mechanism to specify truly concurrent agency. Although pseudo-concurrency is possible by including a clock agent in TkEden, we need to incorporate real concurrency in order to study the full potential of the DMF in practice.

Where the interface for EM is concerned, we need more flexible ways to organise definitions and actions into agents. The development of WING and EME has helped us to explore two new ways to organise definitions and actions – namely directory-like organisation and syntactical organisation of agents. In the next chapter, we shall introduce another way to organise definitions that is arguably better than these two ways.

Another possible enhancement to existing EM tools is to provide a mechanism to generate an LSD specification automatically from a model. This idea was inspired by our experience of extracting an LSD specification from the Dishwasher model, as described in chapter 3. This experience suggests the following general informal rules for building each part of an LSD specification for an agent from a simple definitive script with actions:

- derivates – include all definitions in the agent as derivates.
- states – include all definitive variables on the LHS of derivates in the agent as states.
- handles – include all definitive variables on the LHS within the agent's actions as handles.
- oracles – include all definitive variables on the RHS of derivates as oracles.
- protocols – search for `if` statements in the procedures. The logical expression used by an `if` statement is the LHS (guard) of a protocol, and the sequence of definitions within the `if` statement comprise the RHS of the protocol. Any sequences of definitions that are not guarded by `if` statements in a procedure belong to a protocol with a `true` guard.

One requirement for generating LSD automatically from a script is that TkEden should ‘know’ about (i.e. have an internal representation of) agents. However, this is not the case in the current implementation of TkEden. In the future, if TkEden can be enhanced to include information about agents, further research into the automatic generation of LSD specifications from TkEden models would be viable.

7.7 Summary

In this chapter, we have discussed the evaluation of EM tools from two perspectives. With reference to conceptual support for EM, we have considered the directness of the translation from DMF models to particular implementations. At the tools interface, we have explored techniques that make the process of EM easier to perform. We have investigated three existing technologies, Java, Excel and Forms/3, as possible EM tool implementations. From this we concluded that it is difficult to use OO or pure functional languages to perform the activities of EM. We have also found that model-building in a spreadsheet with event macros captures most of the essential characteristics of the DMF. Both spreadsheets and TkEden reflect a major limitation of current tools for EM: their lack of support for the representation of agents with concurrent actions. We have also described some interface issues for TkEden. The developments of WING and EME explore new ideas that address problematic issues of TkEden. Finally, we have highlighted some major prospects for tool development in the future.

8 The Dependency Modelling Tool

In this chapter, we shall describe a new EM tool, the Dependency Modeling Tool (DMT). The motivation for developing the DMT is similar to that of developing WING and EME – they aim to enhance users' experience in the process of EM. However, the emphasis in DMT is on the ways to visualise various structures that commonly exist in EM models.

We start our discussions by identifying these structures in section 8.1. In section 8.2, we describe some tools developed by others in relation to visualising structures that are similar to the ones that exist in EM models. The development of DMT is partly inspired by some features of these tools. In section 8.3, we introduce DMT's user interface with a simple example. Two major considerations for the development of DMT are model comprehension and reuse. In section 8.4, we discuss how DMT facilitates model comprehension. In section 8.5 we shall discuss how DMT facilitates model reuse. The final section highlights various issues related further research and development DMT.

8.1 Structures in an EM model

The term 'structure' is used in this chapter to refer to some recognised pattern associated with an EM model. These patterns are directly related to the understanding of the model with respect to its context which is gained from the modelling process. For example, the definition of "`a is b+c;`" has the structure of dependency: the value of observable `a` is dependent on the values of observables `b` and `c`. Dependency is not the only type of structure that exists in an EM model. In fact, there are three common structures that can be easily distinguished from a script: *dependency structure*, *locational structure* and *contextual structure*. Dependency structure is the pattern of which observables are related to each other; locational structure refers to the physical organisation and arrangement of definitions in a script; contextual structure to grouping of definitions according to different contexts for observation and interpretation. In our experience, it is usually necessary for the modeller to keep all the structures in mind during the modelling process for purposes of model comprehension.

In a TkEden script, the dependency structure is determined by formulae of definitions. Locational structure is determined by the organisation of definitions in this list (see left-hand-side of the Figure 8.1). There is no direct support for representing contextual structure.

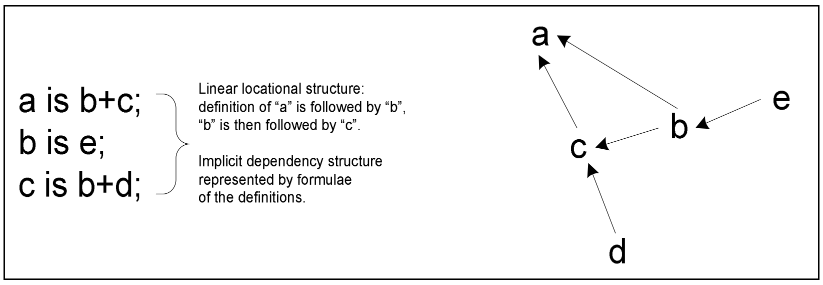

a is b+c;
b is e;
c is b+d;
Linear locational structure: definition of "a" is followed by "b", "b" is then followed by "c".
Implicit dependency structure represented by formulae of the definitions.
a
b
c
d
e

Figure 8.1: Linear locational structure and implicit dependency structure represented by a script of three definitions (left) and a dependency structure graph of the same definitions (right).

Abstractly, we can represent a dependency structure by a directed acyclic graph[4] (showing all dependencies among observables in the model). In the graph, observables are represented as nodes, and dependencies as edges. We can lay out the graph hierarchically: the nodes at the higher levels are dependent on the nodes at the lower levels of the graph[5]. Therefore, nodes at the lowest level of the graph are constant observables or 'undefined observables' (see right-hand-side of the Figure 8.1).

The importance of contextual structure seems to have been largely overlooked in our previous work, although there have been some attempts to deal with them implicitly (e.g. via openshapes in Donald [Bey86] and virtual agents in DTkEden [Sun98]). In Donald, we can define an openshape whose shape is determined by a set of other shapes. For example, an openshape `S` with two lines `L1` and `L2` is defined as:

[4] Similar to a spreadsheet, in an EM model cyclic dependencies are not explicitly represented. This is because cyclic dependencies cause an infinite loop of variable updates.

[5] This hierarchy is the basis for determining the order of variable updates. For example, a topological sort can be performed based on the hierarchy, which can minimise the number of evaluations required for variable updates. Synchronous updates are also possible while still maintaining the integrity of the model.

```
%donald
openshape S
within S {
  line L1
  line L2
}
```

Subsequently, we can refer to two lines individually by `S/L1` and `S/L2`.The contextual structure in this case is accommodated by a syntactic construct of within-clause and a reference symbol '/'. In DTkEden, we can associate definitions with a virtual agent. The following is an example of virtual agent declaration:

```
>>bird
windspeed is 20;
height is 1000;
>>
```

This defines a virtual agent `bird` with two observables. The symbol '`>>`' at the beginning and the end of the declaration specifies contextual information – in this case `windspeed` and `height` belong to the `bird` agent. The actual definitions created by the above declarations are:

```
bird_windspeed is 20;
bird_height is 1000;
```

Literally, a prefix '`bird`' has been added to both definitions with a separator '`_`'.

Representing dependency, locational and contextual structures by using textual syntax in TkEden and DTkEden has a major limitation: it is difficult for a modeller to understand these structures in isolation from other syntactic constructs. Two new tools introduced in the previous chapter had made attempts to address the limitations – WING provides direct support for organising the contextual structure by visualising using a tree explorer similar to the file explorer and locational structure by spreadsheet-like cells. EME visualises the dependency structure by drawing a dependency structure graph. But the results are still not satisfactory.

The aim of the research described in this chapter is accordingly to find better ways of representing the structures that are common to all EM models. On this account, we have developed DMT to represent the structures graphically. We believe that by representing the structures graphically in a coherent way, the experience of *building an EM model as an artefact* can be significantly enhanced. At the same time, the research enhances the prospects of making EM tools more accessible and usable for general users.

8.2 Inspiration from other tools

In this section, we discuss some existing tools that organise structures similar to the structures in EM models. Unlike TkEden script, these tools use graphical techniques to organise the structures. The development of DMT was partly inspired by our experience of using these tools.

A common contextual structure can be found in modern operating systems. That is the organisation of files by a hierarchical structure of directories. Interfaces like the file explorer provide a graphical representation of the directory structure. Most PC users nowadays use them to manage their files instead of typing in command prompts. There are some limitations on using a hierarchical structure to represent contextual structure in EM model. We shall discuss them later in this chapter. However, the idea of organising files by an explicit representation of contextual structure is invaluable to the usability of modern computers.

The importance of explicitly representing both locational and contextual structures is well attested by a popular note taking thinking skill called Mind Mapping [Buz95]. Figure 8.2 shows a Mind Map about the contents of this chapter. A Mind Map is a hierarchical graph with the highest level root located in the centre and branches radiating out in all directions. The root represents a central context of interest. The branches with keywords written on them represent concepts in the context of the keyword from a higher level branch. Relative locations between branches can also convey meanings. Empirical studies of Mind Map use indicate that identifying and managing the hierarchical structures associated with a concept helps people to organise and think about the concept more naturally and creatively [Buz95].

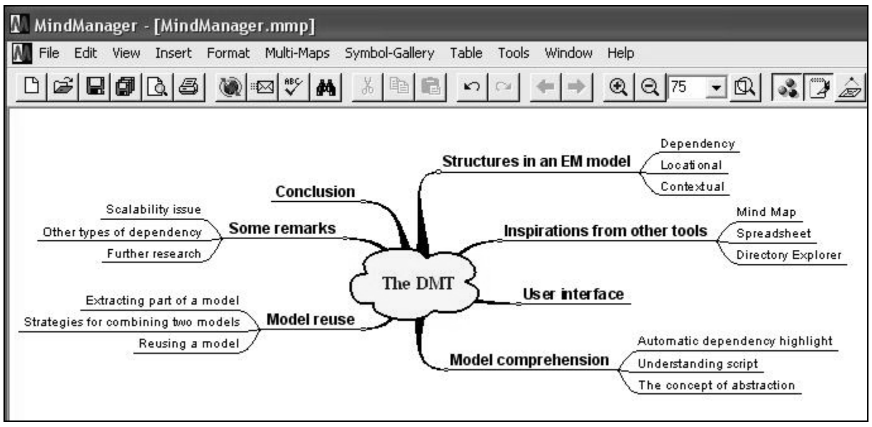
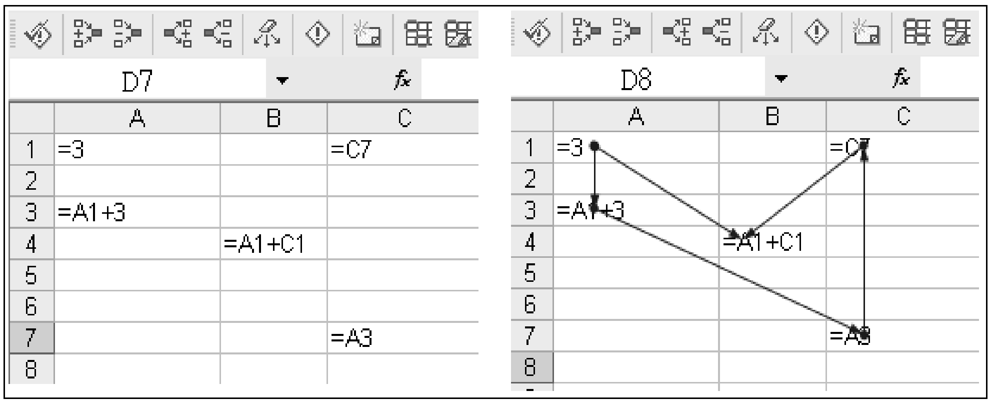

MindManager - [MindManager.mmp]
File Edit View Insert Format Multi-Maps Symbol-Gallery Table Tools Window Help
75
The DMT
Structures in an EM model
Dependency
Locational
Contextual
Inspirations from other tools
Mind Map
Spreadsheet
Directory Explorer
User interface
Model comprehension
Automatic dependency highlight
Understanding script
The concept of abstraction
Conclusion
Some remarks
Scalability issue
Other types of dependency
Further research
Model reuse
Extracting part of a model
Strategies for combining two models
Reusing a model

Figure 8.2: A Mind Map about the contents of this chapter created by using MindManager.

The Mind Map in Figure 8.2 was created by using the MindManager tool [Min00]. One important feature of the tool is that it allows the user to move the nodes freely. This can be interpreted as allowing the user to change the locational structure of the Mind Map. It helps users to organise information according to their subjective preference and, therefore, has cognitive significance.

The feature of explicitly representing dependency structure can also be found in connection with understanding a spreadsheet. The dependencies between cells in a spreadsheet are normally hidden from the user. This makes a spreadsheet difficult to understand [Gre98a]. Newer versions of spreadsheet applications contain a dependency tracing feature. For example, Excel can trace dependencies between cells by showing arrows, as shown in Figure 8.3.


D7

	A	B	C
1	=3		=C7
2			
3	=A1+3		
4		=A1+C1	
5			
6			
7			=A3
8			

D8

	A	B	C
1	=3		=C7
2			
3	=A1+3		
4		=A1+C1	
5			
6			
7			=A3
8			

Figure 8.3: A spreadsheet in Excel (left) and its dependency traces (right).

8.3 User interface

The development of DMT is motivated by the need to enhance users' experience of the process of EM. DMT provides features for users to build EM models as artefacts that are *visually* as well as *physically* more tangible than a definitive script – it uses acyclic graphs to visualise three common structures (dependency, locational and contextual structures) that exist in an EM model, and provides means to manipulate them directly by using a pointing device. In addition, a user can extract definitions created by DMT as Eden definitions, or conversely import Eden definitions from a definitive script. The current version of DMT is implemented in Java with standard Java libraries, so it is platform independent. Figure 8.4 shows the user interface of DMT with an empty model.

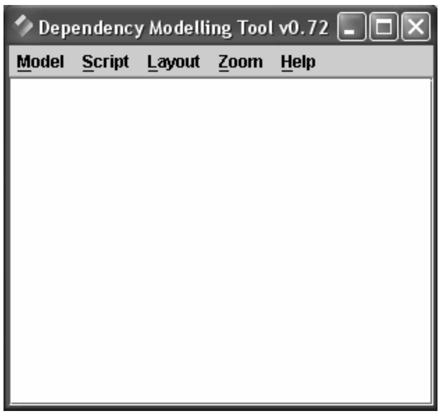

Dependency Modelling Tool v0.72
Model Script Layout Zoom Help

Figure 8.4: User interface of DMT

The interface provides a large empty white area for drawing the dependency graph structure of an EM model. The major functionalities of DMT are reflected by the primary menu options – *Model* provides save, load, print and combine models functions; *Script* provides translation functions to and from definitive scripts; *Layout* provides functions to automatically arrange graphical positions of the nodes; *Zoom* provides functions to scale the entire graph; *Help* provides online help for using the interface (see Appendix H for menu reference).

The basic means of entering a definition can be explained by creating a simple definition: `a is b+c;`. Figure 8.5 shows a sequence of steps to create a graph of this definition and the mechanism to move around the nodes of the graph. The figure illustrates the following basic features of DMT:

- A node can be created by clicking the right mouse button.
- The definition of a node can be edited by double-clicking the node with the left mouse button.
- Any undefined observables will be automatically created as new nodes.
- The details of a node can be checked by pointing at it with the mouse. The details are shown at the top-left region of the graph.
- A group of nodes can be selected by drawing a rectangular selection box around them.
- The selected group of nodes can be moved by drag and drop manipulation of the rectangular selection box (individual node can also be moved by drag and drop without a selection box).

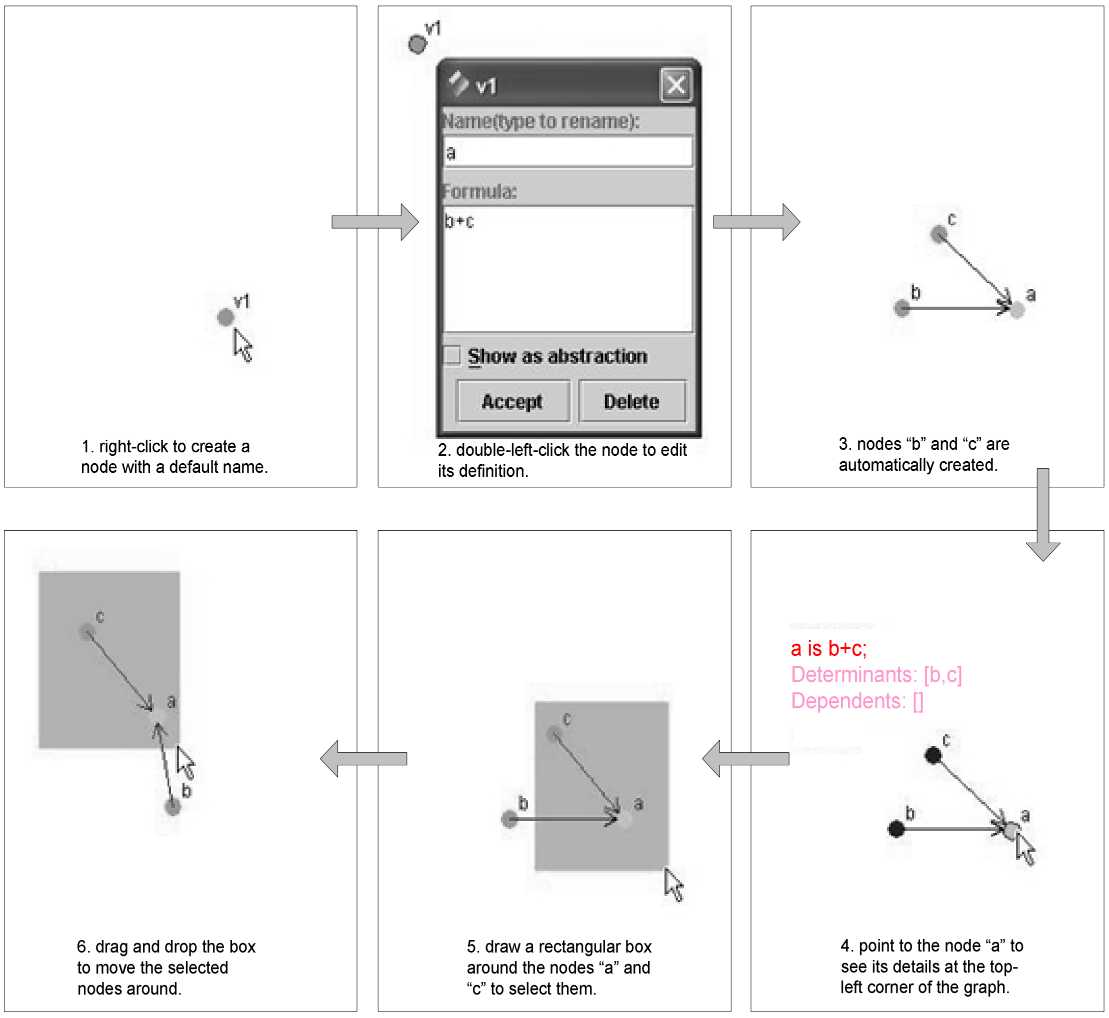

v1
1. right-click to create a node with a default name.
v1
Name(type to rename):
a
Formula:
b+c
Show as abstraction
Accept
Delete
2. double-left-click the node to edit its definition.
c
b
a
3. nodes "b" and "c" are automatically created.
a is b+c;
Determinants: [b,c]
Dependents: []
4. point to the node "a" to see its details at the top-left corner of the graph.
5. draw a rectangular box around the nodes "a" and "c" to select them.
6. drag and drop the box to move the selected nodes around.

Figure 8.5: A sequence of steps to enter a definition and to move nodes around.

DMT uses a *colour coding* for different graphical elements of the graph. Unfortunately, the figures here are printed in black and white. However, the colour coding has significance in understanding the graph. Examples of the colour coding are: nodes with definitions are coloured in grey; nodes with no definitions are coloured in green; the selection box is in light blue.

The semantics of a DMT model, when interpreted as an EM model, can be summarised as follows:

- Observables are nodes.
- Dependency structure is represented by directed-edges joining the nodes. For example, if node `a` depends on node `b`, there is a directed-edge pointing from `b` to `a`.

- Locational structure is represented by arrangement of nodes in a two-dimensional space.
- Contextual structure is represented by abstractions (which will be introduced in another section).

By providing features to graphically represent the structures in an EM and means of directly manipulate them, DMT improves the comprehensibility of the model. In addition, it provides mechanisms to allow easy reuse of existing models. We shall discuss model comprehension and reuse in more detail in the following two sections.

8.4 Model comprehension

DMT provides various features to help the user gain and maintain understanding of the developing model in the process of EM. We shall discuss these features under three headings: automatic dependency highlighting, understanding scripts and abstraction.

8.4.1 Automatic dependency highlighting

As mentioned before DMT uses colour coding to help the user understand the model. For instance, the dependencies related to a particular node are highlighted automatically. By way of illustration, the left-hand-side of Figure 8.6 shows a graph associated with the script of five definitions:

```
a is b+c;
b is 10;
d is a;
c is 10;
e is a;
```

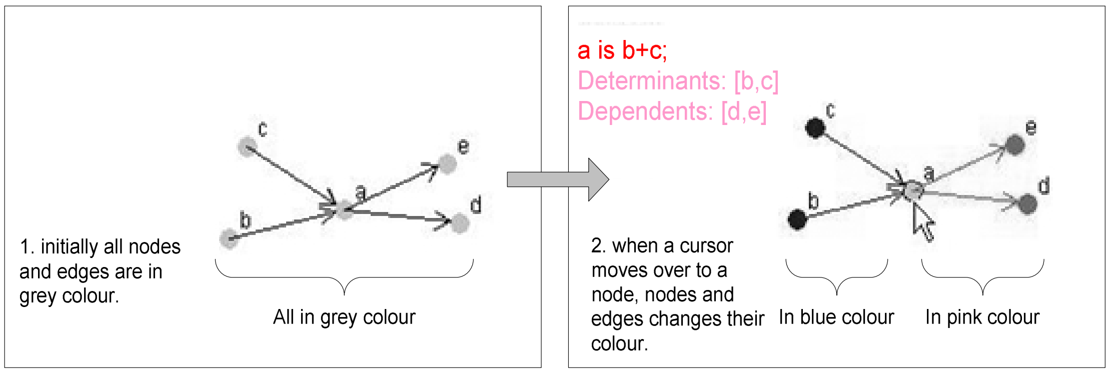

c
e
a
b
d
1. initially all nodes and edges are in grey colour.
All in grey colour
a is b+c;
Determinants: [b,c]
Dependents: [d,e]
2. when a cursor moves over to a node, nodes and edges changes their colour.
In blue colour
In pink colour

Figure 8.6: Automatic dependency highlighting

Initially all the nodes and edges are in grey colour. When the user moves the cursor over node `a`, DMT immediately highlights the nodes and edges associated with its determinants in blue and its dependents in pink (see right-hand-side of Figure 8.6). This automatic visual feedback feature is very useful especially when studying a model with a large number of nodes and edges.

8.4.2 Understanding scripts

We can use DMT as a tool for understanding existing models represented by definitive scripts. DMT can import a definitive script, interpret it and find out all the observables and dependencies represented in it. Since the only positional information explicit in the script is the linear order of the definitions, we need some methods to lay out the graph. There has been much research on algorithms for the automatic arrangement of directed graphs (e.g. [Sug81, Tol96, Pur00]). A typical criterion used for arranging the nodes is minimizing edge crossings. Ordering a directed graph hierarchically is also common. We found that such strategies are of limited use for arranging the layout of an EM model.

The geometric location of nodes in a DMT graph conveys information about a modeller's understanding of the model. A modeller's subjective perspective on the model, as reflected by the location of nodes, is difficult to capture in automatic layout algorithms. Our experience shows that one of the most effective ways to use the DMT is to allow the modeller to arrange the position of the nodes manually. For example, Listing 9.1 shows a definitive script of an ATM model. This script is imported into DMT by choosing the *Script* and *Direct Import* menu options.

```
%eden
userInput is [PINentered,required];
currtime is 71;
required is 0;
overLimitToday is accLimitPerDay <
 (accDrawnToday+required);
accLimitPerDay is 200;
transpossible is cardInMachine &&
cardValid &&  bankValid  && PINvalid
&& !overLimitToday && idValid &&
accStatus && !accOverLimit &&
moneyReady;
accDrawnToday is 0;
ATMbank is ['A', 'B', 'C'];
accStatus is 1;
environment is
[currtime,cardInMachine];
cardInMachine is 0;
cardExpiryDate is 320;
r10 is (required - 20*actual20) / 10;
card is [cardBank, cardID, cardPIN,
cardStartDate, cardExpiryDate,
cardStatus];
ATMtens is 100;
r20 is required / 20;
accOverDraftLimit is 10;
moneyOut is
[transpossible,actual5,actual10,actua
l20];
actual10 is
(ATMtens>=r10)?r10:r10-ATMtens;
cardStartDate is 1;
cardValid is (cardStatus==1) &&
(currtime >= cardStartDate) &&
(currtime <= cardExpiryDate);
cardBank is 'A';
actual20 is
(ATMtwenties>=r20)?r20:r20-ATMtwe
nties;
cardID is 123;
ATMtwenties is 100;
ATMfives is 100;
PINvalid is cardPIN == PINentered;
actual5 is
(ATMfives>=r5)?r5:r5-ATMfives;
accOverLimit is required >
(accTotal+accOverDraftLimit);
cardPIN is 999;
accTotal is 10000;
ATMcardIDlist is [123, 321];
r5 is (required - 20*actual20 -
10*actual10) / 5;
moneyReady is
(actual5*5+actual10*10+actual20*2
0)==required;
PINentered is 999;
cardStatus is 1;
ATMbalance is ATMfives *5 +
ATMtens*10 + ATMtwenties*20;
idValid is isin(cardID,
ATMcardIDlist);
accDetails is [accStatus,
accLimitPerDay, accTotal,
accDrawnToday,accOverDraftLimit];
bankValid is isin(cardBank,
ATMbank);
```

Listing 9.1: A definitive script of an ATM model

After importing the script, DMT randomly positions all the nodes representing observables in the script. The result is usually a graph with a messy arrangement of nodes where many edges cross over, and it is difficult to understand (see Figure 8.7). However, the modeller can get more understanding of the model by moving around the nodes interactively using a pointing device. Moving a node around immediately contributes to the understanding of the determinants and dependents of the observable that the node represents (because of the feature of automatic dependency highlighting). Further grouping of the nodes assists in gaining a better understanding of the model. Eventually, as shown in Figure 8.8, a well-organised layout that reflects the semantics of the model will typically emerge.

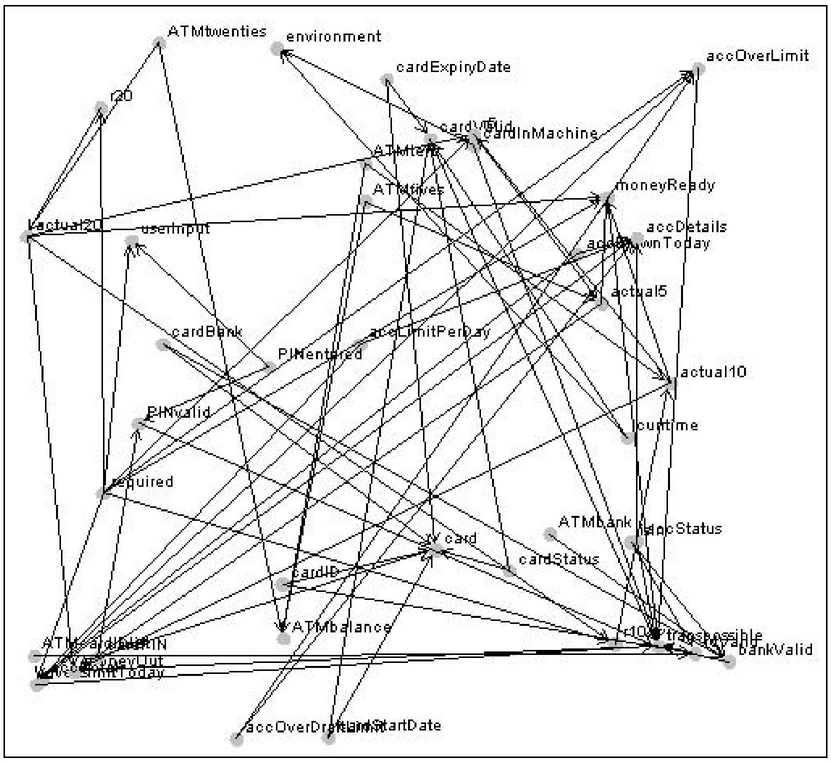


Figure 8.7: Random layout of the ATM dependency graph

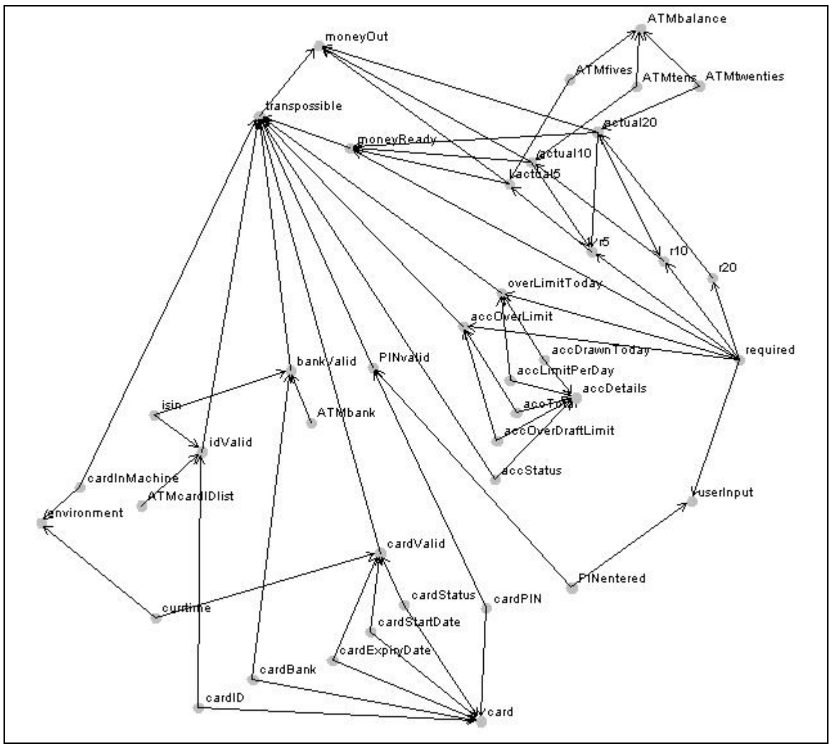


Figure 8.8: Organised representation of the ATM dependency graph

For a small model, a modeller can rapidly understand the model. However, when the model is larger, and consists of say 100 nodes and 300 dependencies, moving the nodes around becomes tiresome. The exploration of the model becomes difficult. This can be solved if we have knowledge of some observables that are more important than the others. If a modeller knows the key observables, nodes can be more easily arranged by firstly locating the nodes corresponding to the key observables, then the observables that are directly connected to the key observable and so on. In the ATM case, if we know that `transpossible` is the most important observable, the arrangement of nodes can be based on it.

For any one particular definitive script, there is a virtually infinite number of ways to layout its dependency graph. Different modellers end up with different layouts even if they all start from the same random layout. This in part reflects the fact that we all understand a particular concept differently. Building a model by arranging the nodes can contribute directly to our construal of the model. The geometric positions of the observables embody part of our understanding of the model.

Apart from understanding an existing model, geometric positioning of nodes can also help in building new models. In this case, the modeller positions an observable (node) each time he or she introduces a definition. Grouping observables and moving groups of observables in conjunction with the model-building activity can contribute visual support for model understanding as it evolves.

8.4.3 Abstraction

This subsection explains the concept of ‘abstraction’ in DMT. The contextual structure of the script can be represented in a way that is similar to the directory navigation of files in a modern operating system. Johnson et al. [Joh99] discuss different ways of representing a directory structure such as outline views, tree diagrams, Venn diagrams and tree-maps (see Figure 8.9).

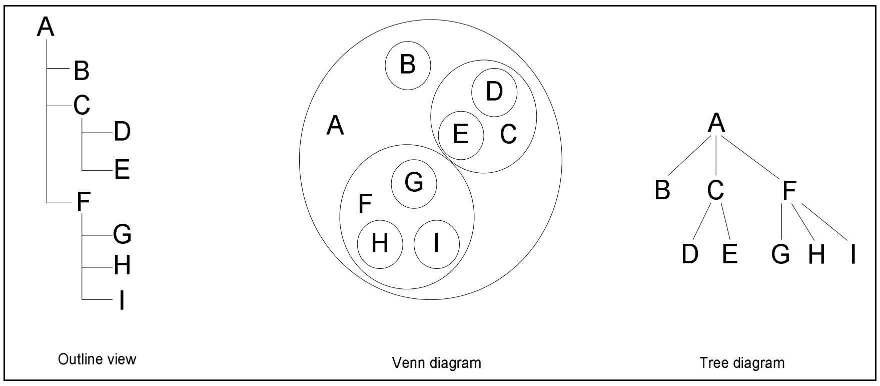
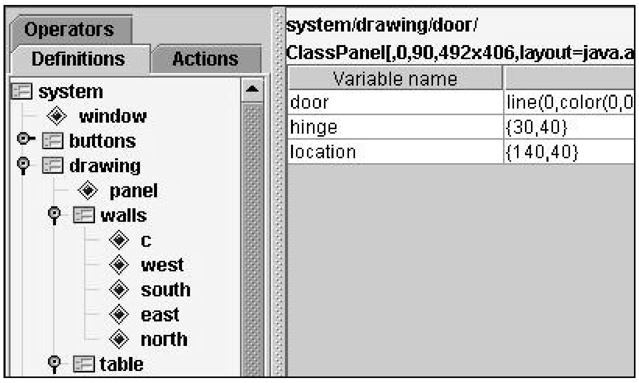


Figure 8.9: Different representations of directory structure

The design of the WING interface attempts to mimic outline view navigation (see Figure 8.10). A user can create new containers that contain sets of definitions just as directories contain sets of files. A dependency is implicitly defined in the sense that a container is dependent on the aggregation of definitions that it contains.



system/drawing/door/
ClassPanel[,0,90,492x406,layout=java.a

Variable name	
door	line(0,color(0,0
hinge	{30,40}
location	{140,40}

Figure 8.10: Outline view of containers in WING

An outline view, however, cannot represent a node with two parent nodes. A typical topological tree of an EM model has nodes with one or more parents. For example, consider the status of the variable c in following definitions:

```
a is b+c;
e is c;
b is 10;
c is 10;
```

There is no direct way of representing the dependency using an outline view. Only the other two kinds of directory representation can be used, as shown in Figure 8.

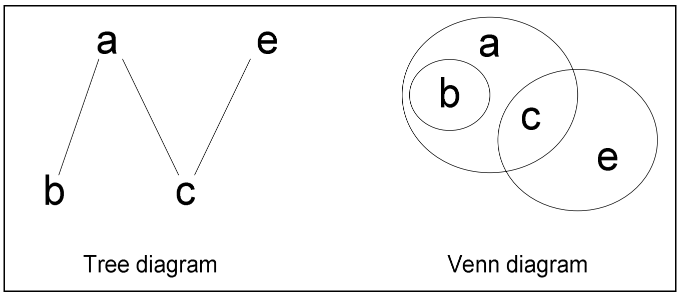
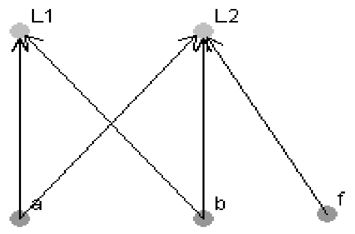


Figure 8.11: Diagram with a node with 2 parents and the Venn diagram.

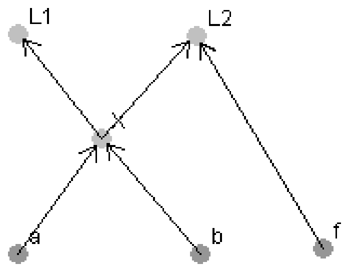
Abstraction in DMT combines the merits of the tree and Venn diagram. We can understand abstraction by firstly consider two example definitions:

```
L1 is (a+b)*4;
L2 is f+a+b;
```

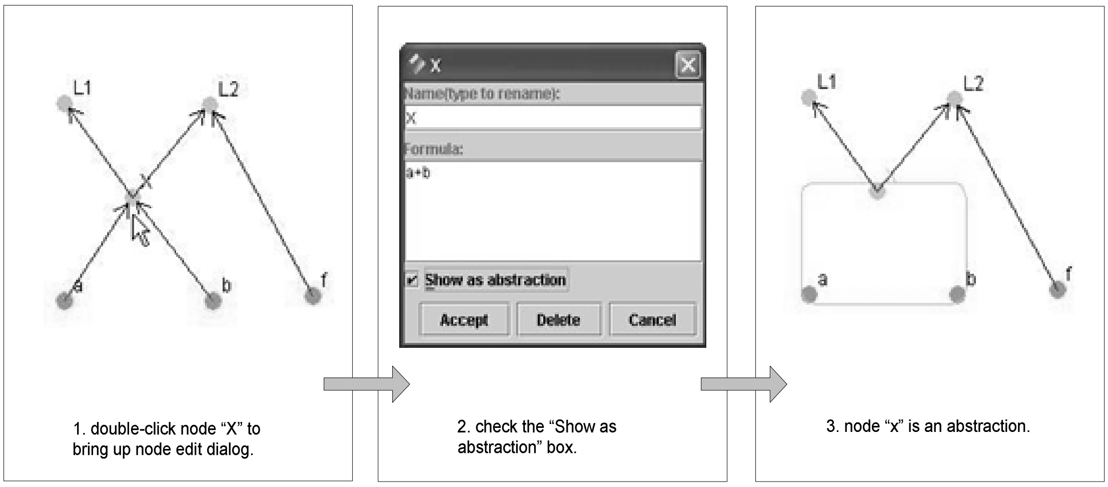


By examining the formulae in these two definitions, it is obvious that they are both dependent on observables `a` and `b` or more precisely on the expression `a+b`. This knowledge of pattern can be captured by replacing the two definitions by the following three 'equivalent' definitions:

```
L1 is X*4;
L2 is f+X;
X is a+b;
```



The third definition here is an abstraction of what we observed. Observable `X` is at a higher level of abstraction than the other observables. To represent the fact that `X` has an abstraction level different from the others, DMT allows a modeller to visual `X` differently by directly specifying it is an abstraction. If `X` is an Abstraction, the colour of it becomes orange and there is a round-cornered orange rectangle that embraces `X` and all its determinants. Figure 8.12 shows a sequence of steps to specify `X` as an abstraction. This figure also shows that the edges from `a` and `b` are hidden as a result of declaring `X` as an abstraction.


L1
L2
X
a
b
f
X
Accept
Delete
Cancel
L1
L2
a
b
f
1. double-click node “X” to bring up node edit dialog.
2. check the “Show as abstraction” box.
3. node “x” is an abstraction.

Name(type to rename):
X
Formula:
a+b
✓ Show as abstraction

Figure 8.12: A sequence of steps to set up an abstraction.

Defining an abstraction can also be viewed as a way of hiding excessive complexity. For a large model, hiding edges directed from the determinants can make the graph less messy. For example, Figure 8.13 shows an observable `Z` that depends on another 20 observables. Specifying `Z` as an abstraction hides all the edges directed towards it.

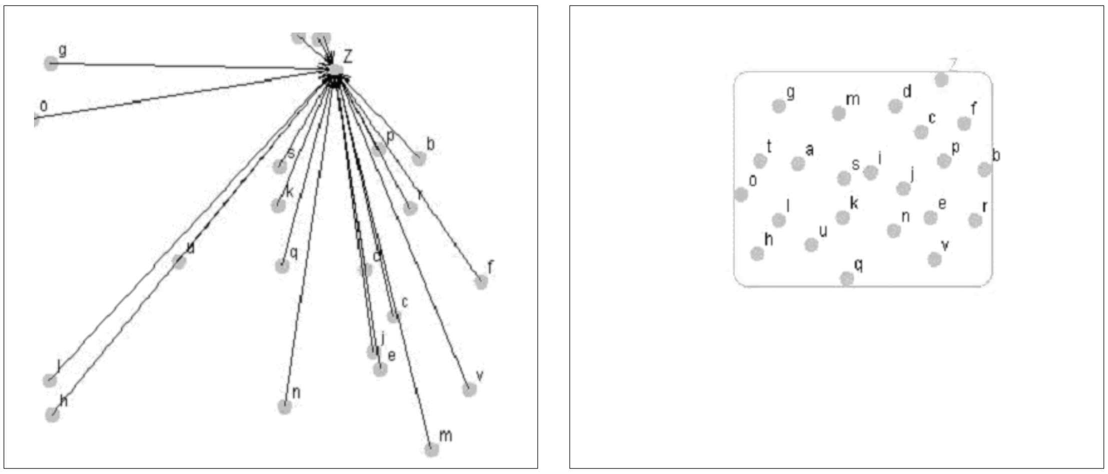


Figure 8.13: Two different representations of the same model – normal representation (left) and ‘Z’ as an abstraction (right).

Defining an abstraction is also a way to explore agency. In the ATM model, we can specify the observables `card` and `cardValid` as agents. As shown in Figure 8.14, their abstractions overlap each other. This might give a clue to the modeller that two separate sets of `card` and `cardValid` observables are needed.

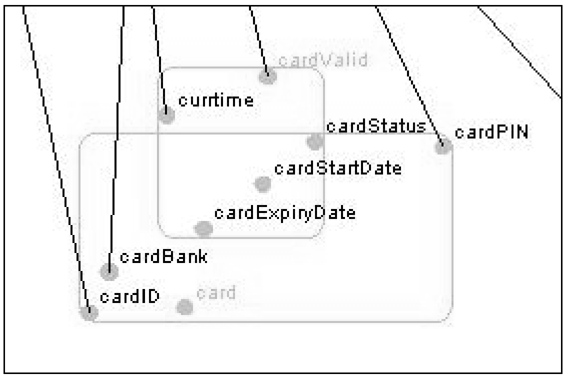


Figure 8.14: Overlapping of two abstractions.

Abstraction can be a unified way of representing agents, directories in Jam2 [Car99], structures in Modd [Geh98] and containers in WING. To summarise, abstraction can be used for:

- Hiding excessive information
- Deriving agency
- Representing agency

By experiencing this way of identifying agents, we notice that an LSD specification can be viewed as a result of the modelling process but does *not* have the generality of an arbitrary script-based EM model. This is because an LSD specification is more suitable for representing settled agents. It does not have the degree of openness that a DMT model has.

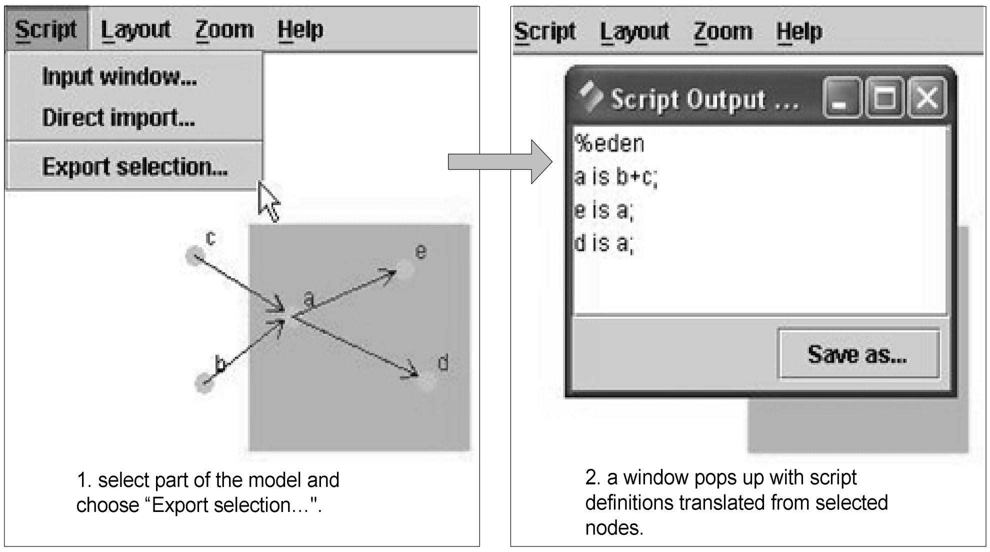
Other kinds of abstraction may also be usefully introduced into the DMT. A counterpart of Harel's hierarchical organisation of states in a statechart [Har88] is one possible candidate. As will be illustrated later in connection with modelling a draughts game, it would be useful in some contexts to be able to abstract groups of observables that exhibit a generic dependency pattern (cf. the observables relating to a single square of the draughts board, as displayed in Figure 8.21.).

8.5 Model reuse

Apart from model comprehension, the other main contribution of DMT is new ways of reusing an EM model. In subsection 8.5.1, we shall describe a mechanism to extract Eden definitions from part of a DMT model. This mechanism is very useful for selecting reusable parts of a model. Model reuse in DMT is based on well-defined strategies of combining two EM models. We shall discuss the strategies in the subsection 8.5.2 and give a simple example of model reuse in subsection 8.5.3.

8.5.1 Extracting part of a model

It is common for some part of an EM model to be generic enough to be reused in building a new model. DMT allows a modeller to extract part of a model and save it for later reuse. Figure 8.15 shows how to extract part of an existing model into a definitive script. DMT automatically appends `%eden` at the beginning of the extracted script. The modeller can save the extracted script to a file for later reuse.


Script
Layout
Zoom
Help
Input window...
Direct import...
Export selection...
c
e
a
b
d
1. select part of the model and choose "Export selection...".
Script Output ...
Save as...
2. a window pops up with script definitions translated from selected nodes.

```
%eden
a is b+c;
e is a;
d is a;
```

Figure 8.15: Extracting script definitions from a DMT model.

8.5.2 Strategies for combining two models

A definition has three ingredients: the definitive variable (or 'observable') at the left-hand-side of the definition, the formula at the right-hand-side and the current value of the variable. What does it mean to say that the definition of a variable is well-defined? Does it mean that all three ingredients of the definition are defined? – or that some ingredients are defined and some are not? We have to take a closer look at each individual ingredient of a definition before answering these questions.

A definitive variable is a metaphorical representation of some external observable. This means in effect that a variable is 'defined' as soon as it is referenced by any definition. That is to say, a particular variable should be treated as defined not only when it is given a defining formula, but when other definitions refer to it in their formulae. Only when a variable is defined in this sense can other ingredients of its definition become meaningful. If a variable has a defining formula, its value can be determined by attempting to evaluate the formula, and if the evaluation is successful, we have a defined value for the variable. On the other hand, if the evaluation fails, the value for the variable will be undefined.

In modelling a situation, the modeller may initially have only a vague idea of what defining formula is appropriate when he or she decides to introduce a variable. In that case, although the variable is defined, its formula is not yet defined and neither is its value. The following table (see Figure 8.16) illustrates all the possibilities that can arise when a variable has been first defined.

	Formula defined	Formula undefined
Value defined	A defined definition	Impossible case
Value undefined	An evaluation exhibits an error.	The dependency of the variable is still subject to investigation.

Figure 8.16: Cases when an observable is defined

DMT's strategies for combining existing models are based on the above notions of defined and undefined ingredients. Here is an example. Suppose we have two models `X` and `Y`. We want to combine them to form a model `Z`. This can be written as:

```
Z = X union Y
```

The general rule for combining two models is *to preserve as much knowledge about observables within the two models as possible, subject to avoiding conflict*. For example, if the first model has observable `v` defined and the second has not, the resulting model will have an observable `v` defined as it is in the first model. Figure 8.17 shows possible cases relating to the definition of `v` in combining `X` and `Y` to form `Z`.

model	case 1	case 2	case 3	case 4	case 5	case 6	case 7	case 8	case 9
X	@	@	@	v is 20	v is 20	v is 20	v is @	v is @	v is @
Y	@	v is 10	v is @	@	v is 10	v is @	@	v is 10	v is @
Z	@	v is 10	v is @	v is 20	conflict	v is 20	v is @	v is 10	v is @

Figure 8.17: An example of possible cases for combining models X and Y to form Z ('@' means 'undefined').

The problematic case is case 5, where definitions of `v` exist in both `X` and `Y` with different formulae. This conflict cannot in general be resolved by applying rules automatically. For example, if we are currently building a model `X` and we want to reuse model `Y` as a building block in `X`, we should choose which definition we actually want manually.

8.5.3 Reusing a model

Reusing a model can be interpreted as combining the model with the new model which we are currently building. Therefore, DMT uses the strategies discussed in the previous subsection to facilitate model reuse. It is convenient to explain and illustrate the idea by an example. The following script defines a generic triangle comprising three lines:

```
x1 is @;
y1 is @;
x2 is @;
y2 is @;
x3 is @;
y3 is @;
L1 is line(x1, y1, x2, y2);
L2 is line(x1, y1, x3, y3);
L3 is line(x2, y2, x3, y3);
```

Its DMT equivalent is shown on the left of the left-hand-side screen capture in Figure 8.18. Suppose our task is to define a generic pattern of two triangles sharing one vertex. In this case the shared vertex is `(x3, y3)`. The steps are as follows:

1. Save the generic triangle model into a file.
2. With the generic triangle still on screen, choose *Combine* to load the file. This brings up a window that contains a second generic triangle (left-hand-side of the Figure 8.18).
3. Rename the nodes in the newly loaded generic triangle to avoid name clashes with the existing ones except in the case of the vertex `(x3, y3)`.
4. Choose *Accept* to combine the two generic triangles into one figure. The result is shown in right-hand-side of Figure 8.18.

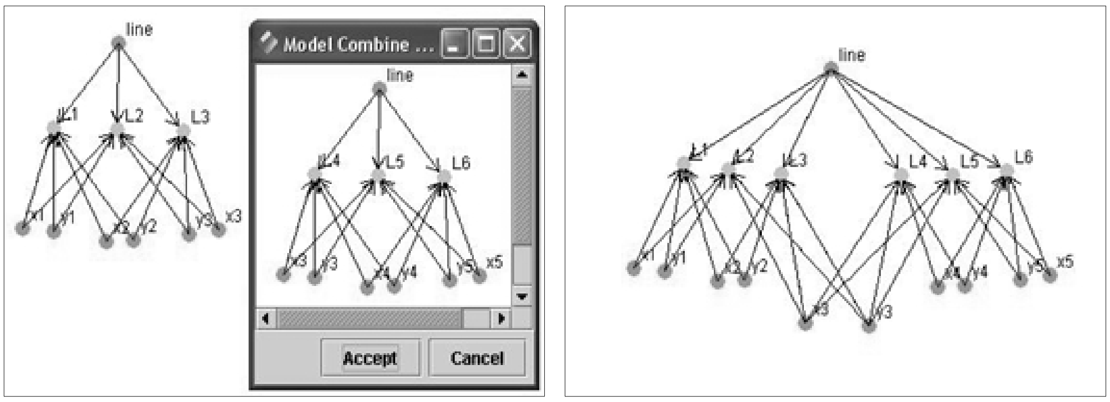


Figure 8.18: The Model Combine dialog with nodes renamed (left) and the result of combining two triangles (right).

As the example shows, when we want to reuse a model, we often need to rename the observables. The renaming of observables in a script is sometimes tedious. This is because the references to an observable may be scattered around everywhere in a long script. For example, we need to find and rename each of coordinates in the generic triangle script 3 times. With the graph representation, DMT centralises all references into one place. Therefore, in DMT, we need only carry out the renaming once for each coordinate to achieve the same result.

8.6 Some remarks

In this section, we discuss various issues related to further research and development of DMT.

8.6.1 Scalability issue

To test the scalability of DMT, we have tried to import many existing EM models in the form of definitive scripts. With a fair amount of time, we can generally rearrange the locations of nodes in each imported model from the initial random layout to a more comprehensible form. However, DMT has encountered problems when we try to visualise models with a large amount of dependencies. For example, the script for the board of an OXO game model contains 209 definitions and 814 dependencies (see Appendix I). After importing this model to DMT, we found that there is no way to rearrange the nodes to get a better layout out of the random layout (see Figure 8.19). In this case, DMT does give the user a hint about the complexity of the model.

However, it is difficult to go a step further in terms of understanding the model out of the random graph. With reference to Appendix I, we can see that the actual script is easier to understand! This illustrates a common scalability problem in visual language for modelling or programming that relates to the limited size of the screen display.

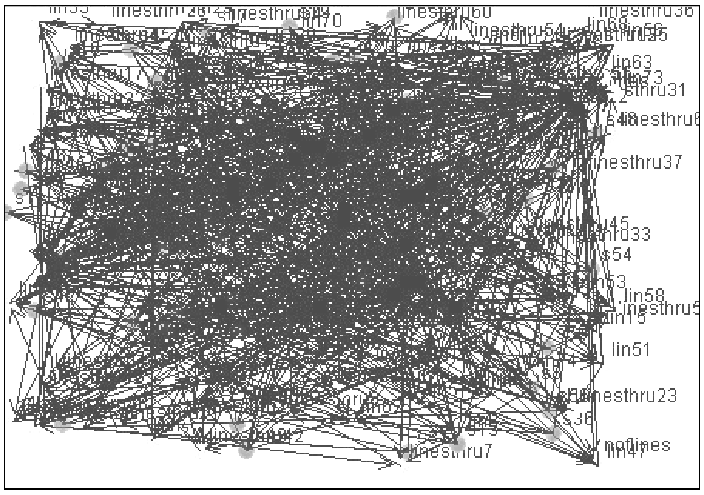


Figure 8.19: Visualisation of a definitive script with a large amount of dependencies

One possible solution is to allow user to select and visualise just parts of the whole complex model, and hide the remaining parts. A simple technique for extracting suitable subscripts for this purpose is to use a text editor to identify all definitive variables with a common pattern or feature. A more sophisticated technique, currently under development by EM research group involves storing the symbol table of a script in a relational database. All the definitions can be stored in a relational database implemented within TkEden using the Eddi definitive notation mentioned in section 2.3.1. The user can then use relational queries to select parts of the model as views. DMT can be developed to allow the user to link these views with their graphical representations. This technique has been used in studying a bug in an EM model described in the following subsection.

8.6.2 Potential for model debugging

One possible use of the DMT is to help the process of debugging EM models. By way of a practical example, we here study a bug in a draughts game model written using a TkEden script (see Figure 8.20). The draughts model contains an 8 by 8 board and some circular pieces. Each square on the board has a circle on it. The fill colour of the circle is as follows: if there is no piece occupying the square its colour should be the same as the background colour of the square, otherwise, its colour should be black or white depending on the colour of the occupying piece. When the background colour of the squares was changed by assigning a new value to the observable `bgcol` the following problem resulted: if a piece was placed on a square and then removed, the fill colour of the circle no longer matched the background colour.

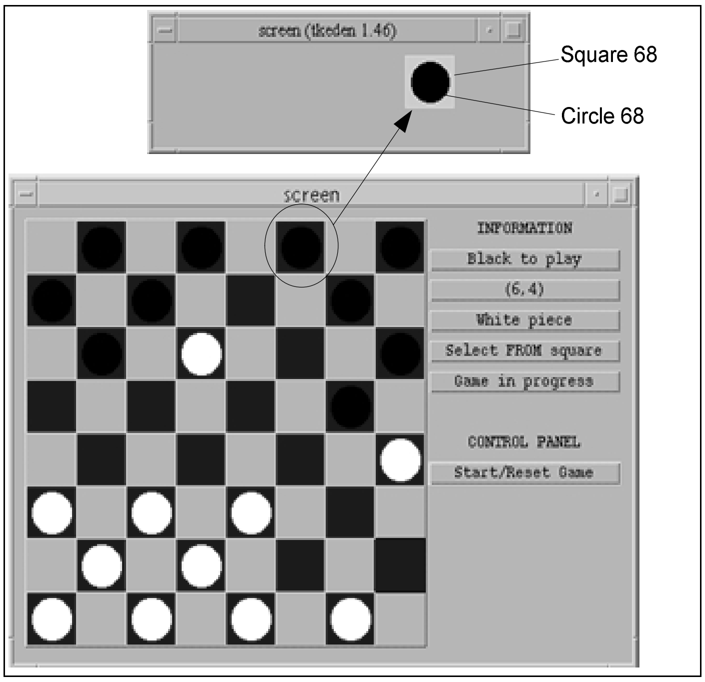


Figure 8.20: The draughts board (bottom) and the study of square 68 (top)

With the help of relation database queries such as we described in the last subsection, we are able to study the problem by selecting and extracting all the definitions relating to a particular square. In this case, we have extracted all the definitions relating to the square and circle on column 6 and row 8 of the board, as shown at the top of Figure 8.20 above. We then study the extracted definitions by using the DMT. Figure 8.21 shows the DMT of the definitions. After rearranging the nodes, we find that the DMT model divides into two sub-graphs: one for the definitions of the circle 68 and the other for the square 68 (see the top screen capture in the Figure 8.21). The fact that the fill colour of the circle (`bgcolor`) and the background colour of the square (`bgcol`) should be the same when no piece is on the square indicates that there should be a dependency between two colours. However, the DMT analysis tells us that there is no dependency between the circle and the square. The bug is removed by adding a new observable (`bckgrncol`) to represent their common colour and defining both `bgcol` and `bgcolor` to be equal to `bckgrncol`. In this way, we make a ‘link’ between the two separate dependency graphs, as shown at the bottom of Figure 8.21. By using the DMT, we found it easier to understand the complex dependencies involved in the EM model. This illustrates one way in which the DMT can lead to more effective debugging of EM models.

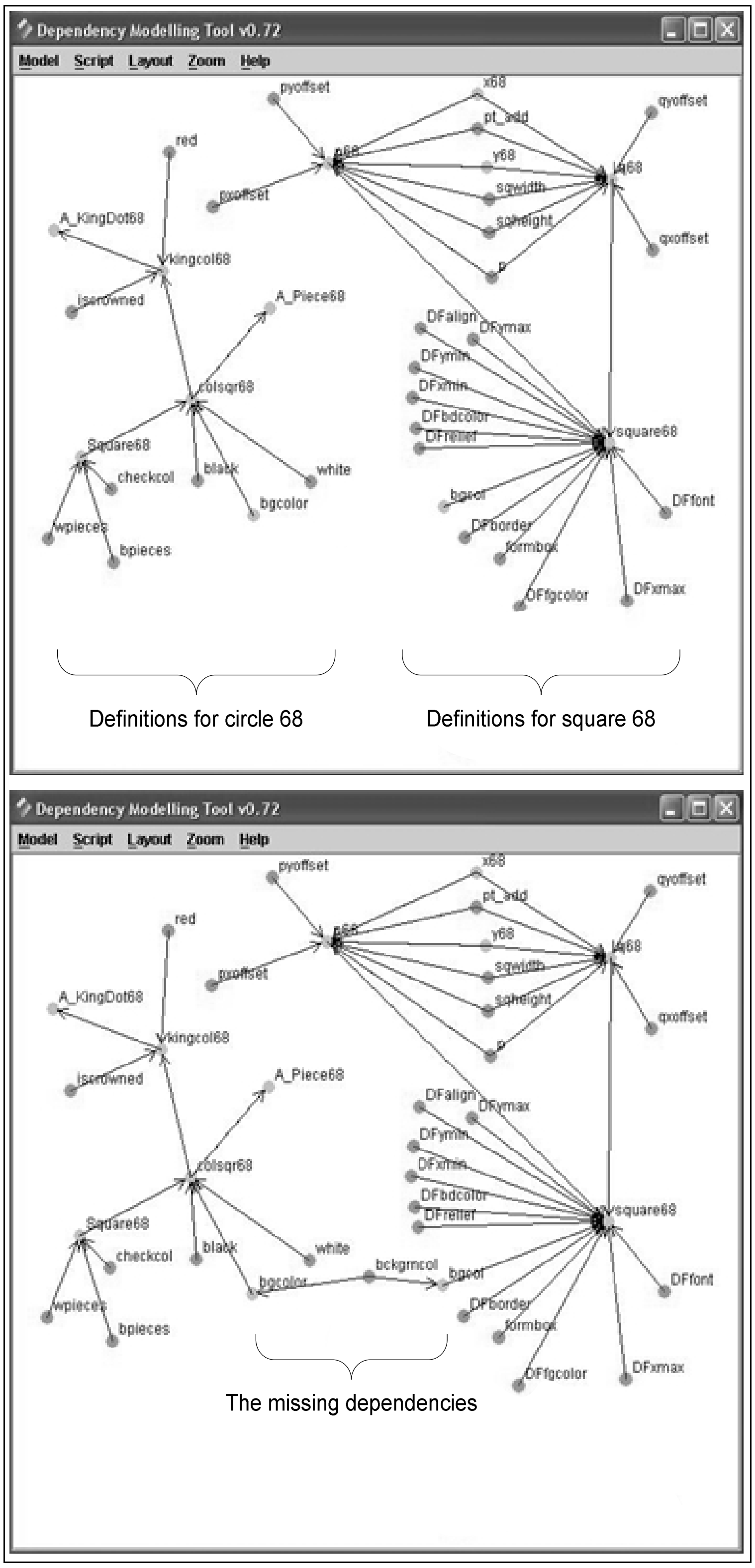


Figure 8.21: The DMT model for a single square of the draughts board (top) and the missing dependencies (bottom)

8.6.3 Other types of dependency

The development of DMT prompts us to ask a question: can we visualise all the dependencies that possibly exist in an EM model? This question leads us to identify different types of dependencies in EM models. In general, we can identify three types of dependencies: *definitive dependency, procedural dependency* and *dynamic dependency*. Definitive dependencies are specified explicitly by formula definitions. For example, the definition `a is b+c;` specifies a definitive dependency between observable `a` and its determinants `b` and `c`. As we have seen, DMT visualises this type of dependency by a directed graph. However, DMT does not visualise the other two types of dependency.

Procedural dependencies are implicitly established by actions. For example, the following action contains a procedural dependency:

```
proc add: b, c {
    a = b + c;
}
```

This action monitors changes of `b` and `c` and *assigns* the sum of them to `a`. However, by just looking at this action, we cannot be sure `a` is merely dependent on `b` and `c`. This is because there may be other actions that also change the value of `a`. Only if we are sure that there is no other action that changes the value of `a` can we replace the action with a definition: `a is b+c;`. In this case, the procedural dependency is transformed to a definitive dependency. The transformation cannot be automatically established. This is because the fact that there is no other action that can change the value of `a` cannot be generated without intelligent intervention from the modeller.

Dynamic dependencies are also implicitly established by actions. But unlike a procedural dependency, a dynamic dependency involves actions making definitions. For example, the following actions establish a dynamic dependency for `a`:

```
proc x: v1{
    a is b+c;
}

proc y: v2{
    a is y;
}
```

In this case, the definition (not value!) of `a` depends on the changes of some other external observables `v1` and `v2`. If `v1` changes, `a` will depend on `b` and `c`. On the other hand, if `v2` changes, `a` will depend on `y`.

In our experience, both procedural and dynamic dependencies are difficult to detect automatically. Detecting these dependencies may involve extensive analysis of the semantics of agents and actions in the model. On the other hand, to some extent they can also be seen as reflect the limitations of our current understanding of the scope for definitive dependency.

8.6.4 Further research

Apart from addressing the scalability issue and visualising other types of dependencies, there are many other possible interesting research topics and developments that can be conducted in relation to DMT in the future. Here we list some of them.

- Research on end-user interface – As we have mentioned in chapter 6, we can use an EM model to control a ubicomp system (as soft-interface). Techniques developed in DMT to visualise and manipulate an EM model are more user friendly than entering definitions using the input window of TkEden where small scripts are involved. Small scripts are arguably easier to build and understand by using the DMT approach than by direct use of TkEden.

- Developing other script translators – Currently, DMT allows only Eden definitions to be imported from a definitive script. Therefore, if we want to use DMT as a comprehension tool for definitive script, we also need to implement other translators for other definitive notations like Donald and Scout.

- Enhancing the dependency maintainer – The latest version of DMT has a simple dependency maintainer that can evaluate definitions with simple arithmetic and conditional operators. Further enhancements of the dependency maintainer include implementing a wider range of data types and operators such as are provided in WING..

- Integrating DMT and TkEden – DMT can be used as an alternative user-interface for inputting definitions into TkEden instead of using TkEden's input window. Integrating DMT and TkEden involves technical research on how to establish two-way communication between Java-based DMT and

TkTcl-based TkEden. Alternatively, TkEden can be used as a platform for implementing DMT.

- Development of a grid-free spreadsheet application – The main functionality of a spreadsheet application is not only calculation but also report generation. To our knowledge, all the commercial spreadsheet applications available at the time of writing this thesis are based on table layouts with grid reference. DMT provides an alternative grid-free layout. In this case, every node in the DMT graph represents a spreadsheet cell whose location can be arranged by the user freely. The user can arrange all the nodes into a report format for printing. For this purpose, the user can choose to print only current values of nodes without the drawing the nodes and edges.

8.7 Summary

In this chapter, we have discussed the research, development and use of DMT. DMT provides a means to visualise and manipulate dependency, locational and contextual structures that commonly exist in EM models. The main contributions of DMT are features to help model comprehension and reuse. In the case of model comprehension, we can trace the dependency of an observable easily by the feature of automatic dependency highlighting. We can import an existing definitive script and explore the dependency within the script interactively. In addition, we can use the concept of abstraction to represent contextual structure discovered in the model. In the case of model reuse, DMT provides interactive ways to extract and combine EM models based on well-defined strategies. We have also discussed the scalability issue of DMT and the limitations in visualising procedural and dynamic dependencies. We have also described some possible further researches and developments.

9 Conclusion

In this concluding chapter, we shall summarise and reflect upon the key ideas that underlie the contents of each chapter. We shall also review the proposals for further research discussed in previous chapters.

In the Empirical Modelling chapter (chapter 2), we have explained that the philosophical foundation of Empirical Modelling (EM) is based on a commonsense way of understanding phenomena in terms of concurrent agents, observables and dependencies. Based on this philosophy, we have proposed the Definitive Modelling Framework (DMF) as a conceptual framework for supporting the cognitive processes behind our commonsense way of understanding phenomena. The DMF is characterised by its distinctive definition-based agent representation, which supports the use of interactive computer-based artefacts (namely Interactive Situation Models or ISMs) for personal and interpersonal modelling activities. We have also described the practical tool TkEden and the LSD notation as the main means of using the DMF in practice. We can find three levels of abstraction within EM research: the philosophy, framework and practice as represented by current tools and modelling building activities. The philosophy is reinforced by the framework, and the framework is realised by the practice. This raises the question: to what extent are there conceptual gaps between the philosophy and the framework and the framework and its current practical realisation? Where the relationship between the philosophy and the framework is concerned, we believe (from our experience) that the gap is already quite narrow. There is a direct correspondence between the concepts in the philosophy and the framework. This is no surprise because the DMF is the result of unification of ideas specially designed for the philosophy as it has evolved from about 20 years of EM research. The justification of this claim is beyond the scope of this thesis. Where the practical realisation of the framework is concerned, we believe that there is still much to be done to narrow the gap. The discussions in chapter 7 and chapter 8 are especially directed towards this concern. In general, chapter 2 supplies a consolidation on EM concepts which not only serves well as an ‘EM in a nutshell’ style of introduction to EM but also serves as a foundation on which in principle any

EM research can be based.

The System Development chapter (chapter 3) aims to give a ‘bird’s-eye’ view of system development from the EM perspective. The chapter starts with a brief overview of some of the principal strands of research that relate to system development. In particular, system development has been informed by two more or less separate research strands. One approaches system development from ‘thinking about systems’; the other from ‘thinking about development activities’. We identify the EM perspective on system development with a broader and more comprehensive view to which both research strands are relevant. This view leads to the identification of three intrinsic properties of systems (complexity, predictability and unity) and three aspects of development activities (cognitive, collaborative and methodological). Where complexity is concerned, EM aims to manage complexity by modelling construals from the simplest to the deepest levels of understanding. Where predictability is concerned, EM offers an alternative approach, based on the ‘what-if’ principle characteristic of the spreadsheet, to the conventional prototyping and testing of system behaviour. It also promises to address issues of system reliability beyond the scope of mathematical analysis and prediction of expected failures. Where unity is concerned, EM presumes that there is no fixed specification for a system (i.e. no fixed system boundary). The system boundary can always be negotiated and can evolve according to the situation. Where the cognitive aspect of development activities is concerned, the EM model facilitates the evolutionary processes of knowledge creation. Where the collaborative aspect of development activities is concerned, the DMF embraces the idea of a hierarchy of human agents in which the negotiation of meaning is facilitated by sharing EM models. Where the methodological aspect of development activities is concerned, the thesis points out that both the standardisation of process (as e.g. in formal methodologies), and of representation (as e.g. in fixed architectural views), have limited application in real world system development. In EM (as is suggested by a growing body of literature), both process and representation should be allowed to evolve according to the current situation. The chapter then turns to a case study that compares EM and an OO modelling with UML. This chapter serves as an introduction to the EM perspective on system development with reference to current state-of-the-art system development research. It is arguably the most comprehensive view of system development based on EM principles so far. In particular, previous doctoral theses that relate to system development using EM only concentrate on the discussion of certain aspects of the topic that are most relevant to their specific research focus.

The Human Problem Solving chapter (chapter 4) discusses what system development using EM essentially entails – computer-supported human problem solving. We have characterised human problem solving using a computer as comprising searches in multiple problem spaces at two levels of abstraction: concrete and abstract. Traditional computer programming primarily supports problem solving either at the concrete level (e.g. spreadsheet paradigms) or the abstract level (e.g. conventional programming paradigms). We believe that EM gives more comprehensive support for problem solving, by assisting the solver in searching problem spaces at both levels of abstraction simultaneously. We argue that building an EM model (namely a Construal of the Problem Solving Situation or CPSS) is an activity well-suited for heuristic problem solving. This is illustrated with reference to a Crossnumber problem and a real-life timetabling problem. As mentioned in the System Development chapter, EM prescribes no method for system development. This chapter elaborates this idea by showing that in EM system development can be guided by situated problem solving heuristics instead of by rational predefined steps.

In the Before Systems: Conceptual Integrity chapter (chapter 5), we point out that conceptual integrity is an important concern before any system is conceived. We have discussed issues that are associated with obtaining conceptual integrity, and explained – and illustrated using the Railway model – how EM can help to address them. The key idea is that conceptual integrity is closely associated with sense-making from a sea of incoherent experiences. The principal concern in the EM perspective on system development is not modelling the functionality of a specific system but on developing a construal of the incoherent experience from which a coherent system conception may emerge. This chapter gives a comprehensive exposition of this idea.

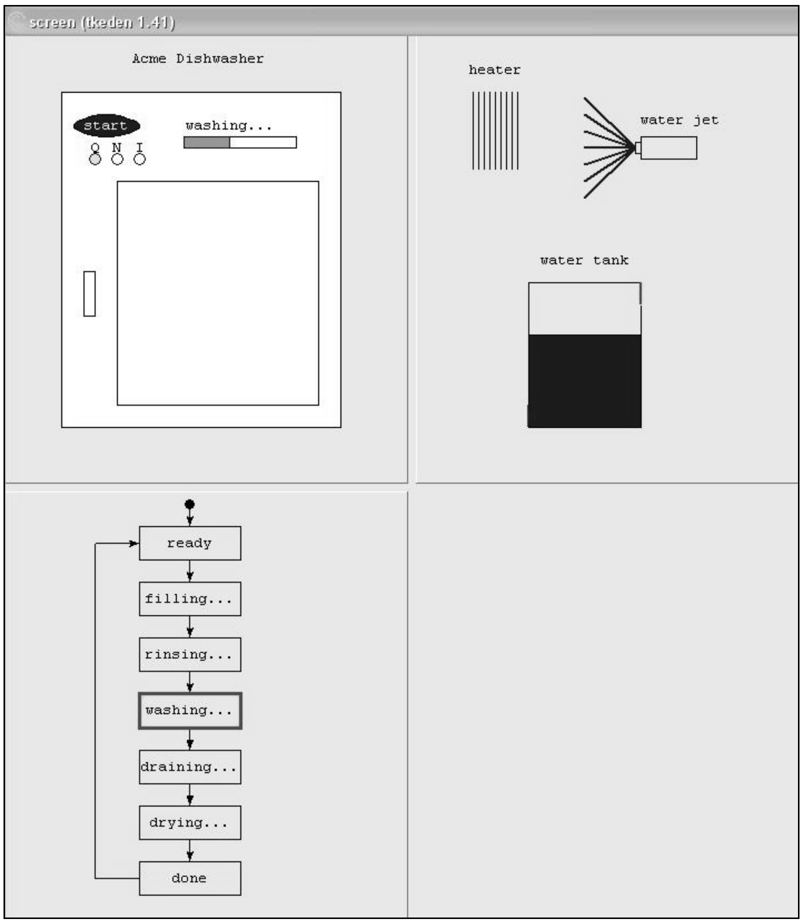
The Beyond Systems: Ubiquitous Computing chapter (chapter 6) is mainly concerned with systems that can never be formalised by developers. The trend towards ubiquitous computing (ubicomp) motivates us to consider systems that can only be formulated when they are used in particular situations. This chapter includes a critical discussion of issues common to many ubicomp research agendas (namely automation, visibility, connectivity and adaptation), and highlights the importance of human involvement in each of these (namely human in the loop, user engagement, understanding and controlling, and user customisation). We propose a new conceptual framework (SICOD) based on EM principles for the ‘development’ of ubicomp systems. We introduce the concept of a ‘soft interface’ as defined by a simple EM model – an Interactive Control Model or ICM – that facilitates the management and customisation of agents (typically ubicomp devices) whose reliable behaviour has been identified. An ICM comprises an ISM that connects a set of Interactive Device Models (IDMs). We have used simple examples to illustrate how a ubicomp system can emerge from constructing a simple ICM. This chapter has explored the potential for applying EM in the modern trend toward everyday life computing. This chapter also discusses many challenges involved in realising the full potential of EM for ubicomp (section 6.4). These challenges motivate future research on this topic.

In the Evaluations and Prospects for EM Tools chapter (chapter 7), we investigate different techniques and tools that can be used to support EM modelling activities. The main concern of this chapter is to minimise the conceptual gap between the concepts in the DMF and the tools. We explore the scope for using different existing technologies for building ISMs, and attempt to construct a simple ISM by using Java, Excel, Forms/3 and TkEden. The general conclusion is that the concepts of OO in Java and first-order functional programming in Forms/3 detract from many essential characteristics of the DMF. Where supporting EM is concerned, TkEden and Excel are the best choices. TkEden is still preferable to Excel because of its support for a variety of special-purpose definitive notations and data types. However, we have identified three major conceptual limitations of TkEden. Firstly, there is no direct mechanism to group definitions and actions into agents. The modeller can only address this limitation partially and indirectly by adding comments to TkEden scripts. Secondly, definitions and actions cannot be updated concurrently. Thirdly, the hybrid concept of procedural variable and definitive variable is confusing. There is no corresponding notion of procedural variable in the DMF. Furthermore, we have identified eight major interface limitations that obstruct the realisation of the DMF. The conceptual and interface limitations of TkEden motivate the development of WING and EME. We discuss features of WING and EME that aim to address most of the limitations. However, due to lack of time, one major limitation of EM tools has remained unexplored – definitions and actions cannot be updated concurrently. This is a significant topic for further research on EM tools.

The Dependency Modelling Tool chapter (chapter 8) describes a new visual tool that aims to enhance the experience of performing modelling activities under the DMF. The motivation behind the Dependency Modelling Tool (DMT) is that there are structures in EM models (namely the dependency, locational and contextual structures) that can be visualised to facilitate model comprehension and reuse. The main characteristic of the DMT is that it allows the modeller to build an EM model as a directed acyclic graph in which observables are nodes, the dependency structure is visualised using directed edges, the locational structure is visualised through the two-dimensional arrangement of nodes, and the contextual structure is visualised using 'abstractions'. Our experience of using the DMT suggests that visualisation and its direct manipulation mechanisms can greatly help the cognitive process of performing modelling activities, especially where model comprehension and debugging are concerned. DMT models have already been used for understanding many TkEden models created in the past. The development of the DMT has opened up many new research directions. Details are discussed in section 8.6.3. The most significant contribution of the tool is its potential for supporting the kind of end-user modelling that is essential for example for the application of EM in ubiquitous computing.

This thesis is the first comprehensive exploration of the potential of EM to support system development. It goes beyond the traditional conception and approach to system development, and promotes system development from interactive construction of computer-based artefacts. The idea of 'emergence through use' is very important in the modern context of systems where the distinction between developer and user is not so clear. With the practical tools developed during the preparation of the thesis, the gap between EM principles and implementations has been narrowed. This provides a solid foundation for future research on applying EM to meet the challenging demands of system development in different domains.

Appendix A: The Dishwasher EM model



```
/* dishwasherModel3-3.s

   The Dishwasher Model

*/


/**************************
     ROLE: A CLOCK
***************************/
%eden
stopCLK = 1;
nextClock = iClock = 0;
proc clocking : iClock, stopCLK {
     if (!stopCLK)
     {
          nextClock++;
          todo("iClock = nextClock;");
     }
}

proc stopClk{
     stopCLK = 1;
}

proc startClk{
     stopCLK = 0;
}
```

```
proc resetClk{
     iClock = nextClock = 0;
}

/*******************************
     ROLE: A MAINTENANCE ENGINEER
*******************************/

/*  unfinished */

/**********************
     ROLE: A USER
**********************/
%eden
proc startJob{
   startProcess();
}

proc changeMode{
   para char;
   if(char=='Q'){

      mode is quick;
   } else if(char=='N'){
      mode is normal;
   } else if(char=='I'){
      mode is intensive;
   }
}

proc openDoor{
    door is open;
}

proc closeDoor{

    door is closed;
}

/**********************
   ROLE: AN ENGINEER
**********************/
%eden
/* door */
open is 0;
closed is 1;
door is closed;

/* wash mode */
quick is 1;
normal is 2;
intensive is 3;

mode is normal;
rinseTime is (mode == quick)?1:((mode == normal)?2:8);
washTime is (mode == quick)?1:((mode == normal)?2:8);
dryTime is (mode == quick)?1:((mode == normal)?2:8);

/* washing progress */
ready is "ready";
go is "filling...";
filled  is "rinsing...";
rinsed is "washing...";
washed  is "draining...";
drained is "drying...";
dried   is "done";

progress is ready;

proc startProcess{
    startClk();
    startTime is time();
    progress is go;
}

proc monitorDoor: door{
    if(door==open){
         writeln("door opened.");
         if(progress==dried)
           progress is ready;
    }
    else{
         startTime is time();
         writeln("door closed.");
    }
}

/*
proc doneWash: iClock{
    if(progress==dried && door==closed){
        progress is done;
        writeln("done.");
    }
}
*/

/* tank */
tankEmpty is 0;
tankFull is 1;
tankSomewater is 2;
valveClosed is 0;
valveOpen is 1;

tankCapacity is 20000;
waterFlowPerSecond is 50;
waterLevel is 1;
waterPercent is float(waterLevel) / float(tankCapacity) * 100;
waterLevelStatus is
(waterLevel==0)?tankEmpty:((waterLevel>=tankCapacity)?tankFull:tankSomewater);
drainValve is (progress==washed && door==closed &&
waterLevelStatus!=tankEmpty)?valveOpen:valveClosed;
fillValve is (progress==go && door==closed &&
waterLevelStatus!=tankFull)?valveOpen:valveClosed;

/*
proc monitorTank: waterLevel, waterFlowPerSecond, drainValve, fillValve {

     writeln("waterlevel:", waterLevel," waterFlowPerSecond:",waterFlowPerSecond,"
drainValve:",drainValve," fillValve:",fillValve);
}
*/

proc waterSimulation: iClock{

     if(fillValve==valveOpen){
         execute("waterLevel is eval(waterLevel+waterFlowPerSecond);");
     }


     if(drainValve==valveOpen && waterLevelStatus!=tankEmpty){
         execute("waterLevel is eval(waterLevel-waterFlowPerSecond);");
         if(waterLevel<0)
             waterLevel is 0;
     }


}



proc fillTank: iClock{
    if(progress==go && door==closed && waterLevelStatus==tankFull){
        progress is filled;
        writeln("filled.");
    }
}

proc drainTank: iClock{
    if(progress==washed && door==closed && waterLevelStatus==tankEmpty){
        progress is drained;
     writeln("drained.");
    }
}
```

```
/* jet */

jetRinseWaterPerSecond is 2;
jetWashWaterPerSecond is 4;
jetSecondPerPulse is 1;

jetSpraying is 0;
jetPulsing is 1;
jetOff is 2;

jetLeft is -1;
jetRight is 1;
jetMiddle is 0;

jetStatus is (progress==filled)?jetSpraying:((progress==rinsed)?jetPulsing:jetOff);
jetDirection is (jetStatus==jetSpraying
||jetStatus==jetOff)?jetMiddle:((jetStatus==jetPulsing &&
(iClock/jetSecondPerPulse)%2==1)?jetRight:jetLeft);

/*
proc monitorJetDirect: jetDirection{
   write(jetDirection);
}
*/

proc rinseDish: iClock{
    if(progress==filled && door==closed){

       execute("waterLevel is eval(waterLevel-jetRinseWaterPerSecond);");
       if(waterLevel<0)waterLevel is 0;


       if(time()-startTime>rinseTime){
         progress is rinsed;
         startTime is time();
         writeln("rinsed.");
       }
    }
}

proc washDish: iClock{

    if(progress==rinsed &&  door==closed){

       execute("waterLevel is eval(waterLevel-jetWashWaterPerSecond);");
       if(waterLevel<0)waterLevel is 0;

       if(time()-startTime>washTime){
     progress is washed;
        startTime is time();
        writeln("washed.");
       }
    }
}

/* heater */

proc dryDish: iClock{

    if(progress==drained && time()-startTime>washTime && door==closed){
     progress is dried;
        startTime is time();
        writeln("dried.");
    }
}

/*******************************************
      DISHWASHER COMPONENTS VISUALISATIONS
*******************************************/
%donald
viewport visualisation

# tank visualisation
rectangle theTank
theTank = rectangle({100,50},{200,50+waterPercent!})
?A_theTank = "color=blue,fill=solid";
```

```
line theTankTop, theTankLeft, theTankRight, theFillValve, theDrainValve
theTankTop = [{100,180},{200,180}]
theTankLeft = [{100,70},{100,180}]
theTankRight = [{200,50},{200,160}]
?theFillValveX is (fillValve==valveOpen)?180:200;
?theDrainValveX is (drainValve==valveOpen)?80:100;
theFillValve = [{theFillValveX!,160},{200,180}]
?A_theFillValve = "color=red,linewidth=2";
theDrainValve = [{theDrainValveX!,50},{100,70}]
?A_theDrainValve = "color=red,linewidth=2";

label theTankLabel
theTankLabel = label("water tank", {150,200})

# heater visualisation
int bargap, barx,bary
line bar1,bar2,bar3,bar4,bar5,bar6,bar7,bar8,bar9
bargap = 5
barx = 50
bary = 350

bar1 = [{barx,bary },{barx,bary-70}]
bar2 = [{barx+bargap,bary },{barx+bargap,bary-70}]
bar3 = [{barx+bargap*2,bary },{barx+bargap*2,bary-70}]
bar4 = [{barx+bargap*3,bary },{barx+bargap*3,bary-70}]
bar5 = [{barx+bargap*4,bary },{barx+bargap*4,bary-70}]
bar6 = [{barx+bargap*5,bary },{barx+bargap*5,bary-70}]
bar7 = [{barx+bargap*6,bary },{barx+bargap*6,bary-70}]
bar8 = [{barx+bargap*7,bary },{barx+bargap*7,bary-70}]
bar9 = [{barx+bargap*8,bary },{barx+bargap*8,bary-70}]

?A_bar1 is (progress==drained)?"linewidth=3,color=red":"";
?A_bar2 is A_bar1;
?A_bar3 is A_bar1;
?A_bar4 is A_bar1;
?A_bar5 is A_bar1;
?A_bar6 is A_bar1;
?A_bar7 is A_bar1;
?A_bar8 is A_bar1;
?A_bar9 is A_bar1;

label theHeaterLabel
theHeaterLabel = label("heater", {barx+20,bary+20})

#jet visualisation
int jetx, jety

jetx = 200
jety = 300

rectangle theJet, theJetHead
theJet = rectangle({jetx,jety+10},{jetx+50,jety-10})
theJetHead = rectangle({jetx-5,jety+5},{jetx,jety-5})

line water0, water1, water2, water3, water4, water5, water6

water0 = [{jetx-5,jety}, {jetx-50,jety+45} ]
water1 = [{jetx-5,jety}, {jetx-50,jety+30} ]
water2 = [{jetx-5,jety}, {jetx-50,jety+15} ]
water3 = [{jetx-5,jety}, {jetx-50,jety} ]
water4 = [{jetx-5,jety}, {jetx-50,jety-15} ]
water5 = [{jetx-5,jety}, {jetx-50,jety-30} ]
water6 = [{jetx-5,jety}, {jetx-50,jety-45} ]

?A_water0 is (progress==rinsed && door==closed &&
jetDirection==jetLeft)?"linewidth=2,color=blue":"color=grey";
?A_water1 is A_water0;
?A_water5 is (progress==rinsed && door==closed &&
jetDirection==jetRight)?"linewidth=2,color=blue":"color=grey";
?A_water6 is A_water5;


?A_water2 is ((progress==filled || progress==rinsed) && door==closed &&
jetStatus==jetSpraying)?"linewidth=2,color=blue":"color=grey";
?A_water3 is A_water2;
?A_water4 is A_water2;
```

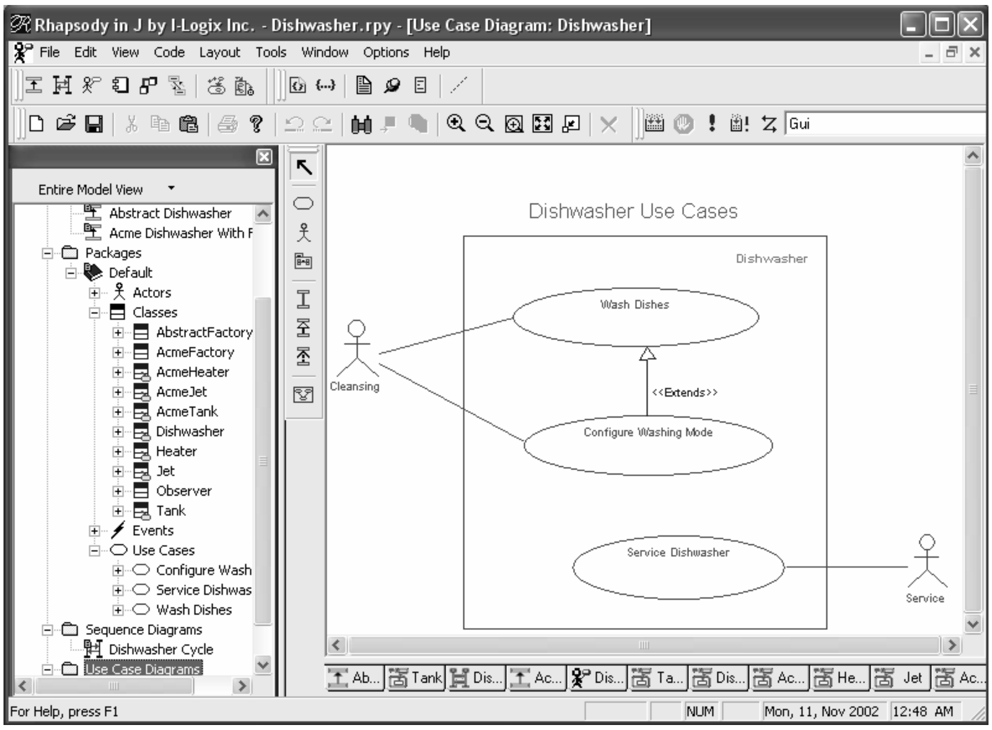
```
label theJetLabel
theJetLabel = label("water jet", {jetx+35,jety+25})


%scout
window visualisationWindow;

visualisationWindow = {
     box: [{370,0}, {730,400}],
     pict:     "visualisation",
     type:     DONALD,
     xmin:     0,
     ymin:     0,
     xmax:     360,
     ymax:     400,
     border:   1,
     sensitive: ON
};

%eden

/*******************************************
      STATECHART VISUALISATIONS
*******************************************/
%donald
viewport statechart

rectangle stateReady, stateFilling, stateRinsing, stateWashing, stateDraining,
stateDrying, stateDone
label fillingLabel,rinsingLabel,washingLabel,drainingLabel,dryingLabel, readyLabel,
doneLabel

int startx, starty, height, width, gap
point readyin,readyout,
fillin,fillout,rinsein,rinseout,washin,washout,drainin,drainout,dryin,dryout,donein,
doneout
line readytofill, filltorinse, rinsetowash, washtodrain, draintodry, drytodone

startx = 120
starty = 370
gap = 20
height = 30
width = 90


stateReady = rectangle({startx,starty},{startx+width, starty-height})
?A_stateReady is (progress==ready)?"outlinecolor=red,linewidth=3":"";
readyLabel = label(ready!, {startx + width div 2, starty - height div 2})
readyin = {startx + width div 2, starty}
readyout = {startx + width div 2, starty-height}

stateFilling = rectangle({startx,starty-(height+gap)},{startx+width,
starty-(height+gap)-height})
?A_stateFilling is (progress==go)?"outlinecolor=red,linewidth=3":"";
fillingLabel = label(go!, {startx + width div 2, starty-(height+gap) - height div 2})
fillin = {startx + width div 2, starty-(height+gap)}
fillout = {startx + width div 2, starty-(height+gap)-height}

stateRinsing = rectangle({startx,starty-(height+gap)*2},{startx+width,
starty-(height+gap)*2-height})
?A_stateRinsing is (progress==filled)?"outlinecolor=red,linewidth=3":"";
rinsingLabel = label(filled!, {startx + width div 2, starty-(height+gap)*2 - height div
2})
rinsein = {startx + width div 2, starty-(height+gap)*2}
rinseout = {startx + width div 2, starty-(height+gap)*2-height}

stateWashing = rectangle({startx,starty-(height+gap)*3},{startx+width,
starty-(height+gap)*3-height})
?A_stateWashing is (progress==rinsed)?"outlinecolor=red,linewidth=3":"";
washingLabel = label(rinsed!, {startx + width div 2, starty-(height+gap)*3 - height div
2})
washin = {startx + width div 2, starty-(height+gap)*3}
washout = {startx + width div 2, starty-(height+gap)*3-height}

stateDraining = rectangle({startx,starty-(height+gap)*4},{startx+width,
starty-(height+gap)*4-height})
```

```
?A_stateDraining is (progress==washed)?"outlinecolor=red,linewidth=3":"";
drainingLabel = label(washed!, {startx + width div 2, starty-(height+gap)*4 - height div
2})
drainin = {startx + width div 2, starty-(height+gap)*4}
drainout = {startx + width div 2, starty-(height+gap)*4-height}

stateDrying = rectangle({startx,starty-(height+gap)*5},{startx+width,
starty-(height+gap)*5-height})
?A_stateDrying is (progress==drained)?"outlinecolor=red,linewidth=3":"";
dryingLabel = label(drained!, {startx + width div 2, starty-(height+gap)*5 - height div
2})
dryin = {startx + width div 2, starty-(height+gap)*5}
dryout = {startx + width div 2, starty-(height+gap)*5-height}

stateDone = rectangle({startx,starty-(height+gap)*6},{startx+width,
starty-(height+gap)*6-height})
?A_stateDone is (progress==dried)?"outlinecolor=red,linewidth=3":"";
doneLabel = label(dried!, {startx + width div 2, starty-(height+gap)*6 - height div 2})
donein = {startx + width div 2, starty-(height+gap)*6}
doneout = {startx + width div 2, starty-(height+gap)*6-height}

readytofill = [readyout, fillin]
?A_readytofill ="arrow=last";

filltorinse = [fillout, rinsein]
?A_filltorinse ="arrow=last";

rinsetowash = [rinseout, washin]
?A_rinsetowash ="arrow=last";

washtodrain = [washout, drainin]
?A_washtodrain ="arrow=last";

draintodry = [drainout, dryin]
?A_draintodry ="arrow=last";

drytodone = [dryout, donein]
?A_drytodone ="arrow=last";

# the init dot and arrow
point stateInit
stateInit = {readyin.1,readyin.2+20}

circle initCircle
initCircle = circle(stateInit, 4)
?A_initCircle="fill=solid";

line initLine
initLine = [stateInit,readyin]
?A_initLine ="arrow=last";

#the return routine
line l1,l2,l3

l1 = [{startx, starty-(height+gap)*6 - height div 2}, {startx-40, starty-(height+gap)*6
- height div 2}]
l2 = [{startx-40, starty-(height+gap)*6 - height div 2}, {startx-40,starty-height div
2}]
l3 = [{startx, starty- height div 2}, {startx-40, starty- height div 2}]
?A_l3 ="arrow=first";

%scout
window statechartWindow;

statechartWindow = {
     box: [{0,410}, {360,810}],
     pict:     "statechart",
     type:     DONALD,
     xmin:     0,
     ymin:     0,
     xmax:     360,
     ymax:     400,
     border:   1,
     sensitive: OFF
};

%eden
```

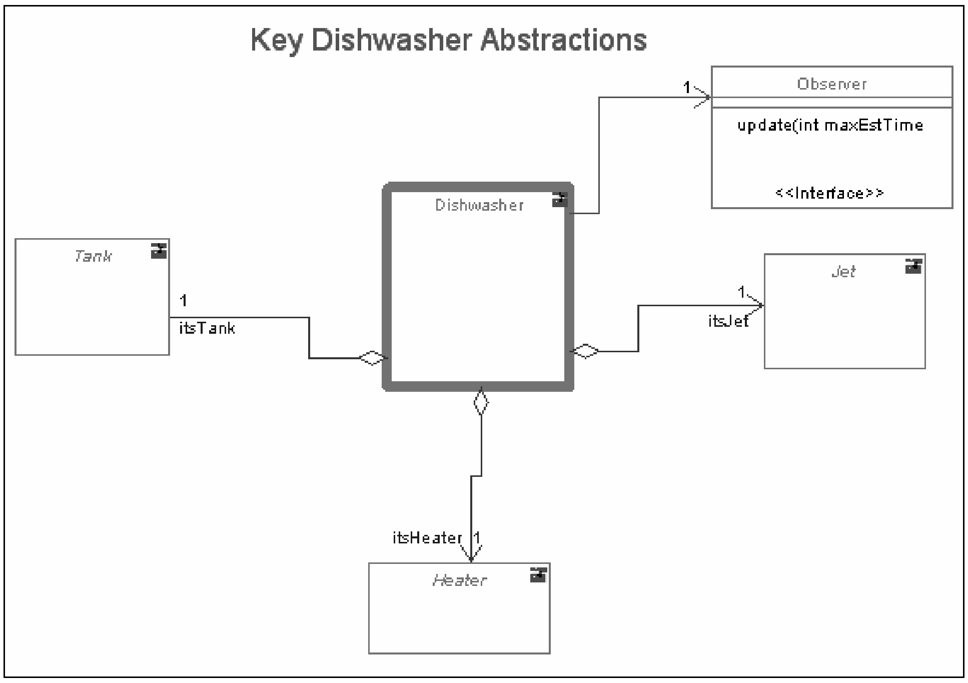
```
/******************************
  ROLE: AN INTERFACE DESIGNER
*******************************/

%donald
viewport interface

#interface title(not part of the interface)
label titleLabel
titleLabel=label("Acme Dishwasher", {173,380})

#shape of the machine
rectangle machineShape
machineShape=rectangle({50,50},{300,350})
?A_machineShape = "color=white,fill=solid";

#the start button
ellipse startButton
startButton=ellipse({90,320},{90,330},{120,320})
?A_startButton = "outlinecolor=black,color=blue,fill=solid";

label startLabel
startLabel=label("start", {90,320})
?A_startLabel = "color=white";

#the door
rectangle theInside
theInside=rectangle({100,70},{280,270})
?A_theInside = "color=grey50,fill=solid";

rectangle theDoor
?theDoorX is (door==closed)?100:270;
theDoor=rectangle({theDoorX!,70},{280,270})
?A_theDoor = "color=white,fill=solid";

rectangle theHandle
theHandle=rectangle({70,150},{80,190})


#the progress bar

%eden
func convertProgressNum{
     auto result;
     result = 0;

     if($1==filled)result=1;
     else if($1==rinsed)result=2;
     else if($1==washed)result=3;
     else if($1==drained)result=4;
     else if($1==dried)result=5;

     return result;
}
progressNum is convertProgressNum(progress);

%donald
rectangle theBarBackground
theBarBackground=rectangle({160,300},{260,310})

rectangle theBar
theBar=rectangle({160,300},{160+20*progressNum!,310})
?A_theBar = "color=green,fill=solid";

#the progress label
label progressLabel
progressLabel=label(progress!, {200,320})
?A_progressLabel = "color=black";

#mode buttons
circle quickButton
quickButton = circle({80,290},5)
?A_quickButton is (mode==quick)?"color=yellow,fill=solid":"";

label qLabel
qLabel = label("Q",{80,300})
```

```
circle normalButton
normalButton = circle({100,290},5)
?A_normalButton is (mode==normal)?"color=yellow,fill=solid":"";

label nLabel
nLabel = label("N",{100,300})

circle intensiveButton
intensiveButton = circle({120,290},5)
?A_intensiveButton is (mode==intensive)?"color=yellow,fill=solid":"";

label iLabel
iLabel = label("I",{120,300})

%eden

proc mouseMonitor:interfaceWindow_mouse{
    auto x,y;
    if(interfaceWindow_mouse[2]==4){
         x=interfaceWindow_mouse[4];
         y=interfaceWindow_mouse[5];
         if(x>=60 && x<=120 && y>=310 && y<=330)
          startJob();
         /* door handle*/
         else if(x>=70 && x<=80 && y>=150 && y<=190){
          if(door==closed) door is open;
                else door is closed;
              }
         /*quick button*/
         else if(x>=75 && x<=85 && y>=285 && y<=295){
              mode is quick;
              }
         /*normal button*/
         else if(x>=95 && x<=105 && y>=285 && y<=295){
              mode is normal;
              }
         /*intensive button*/
         else if(x>=115 && x<=125 && y>=285 && y<=295){
              mode is intensive;
              }
         else
            write("mouse x:",int(x)," y:",int(y),"\n");

    }
}

%scout
window interfaceWindow;

interfaceWindow = {
     box: [{0,0}, {360,400}],
     pict:     "interface",
     type:     DONALD,
     xmin:     0,
     ymin:     0,
     xmax:     360,
     ymax:     400,
     border:   1,
     sensitive: ON
};

%eden
/****************************
   COMBINE DONALD VIEWPORTS
****************************/
%scout

display d;
d=<interfaceWindow/visualisationWindow/statechartWindow>;
screen= d;

%eden
```

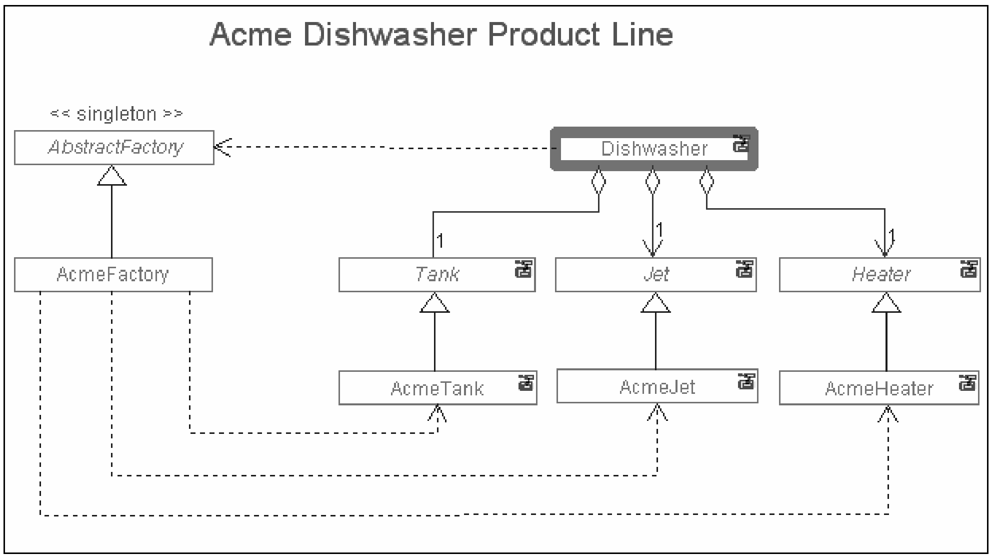
Appendix B: The Dishwasher UML model



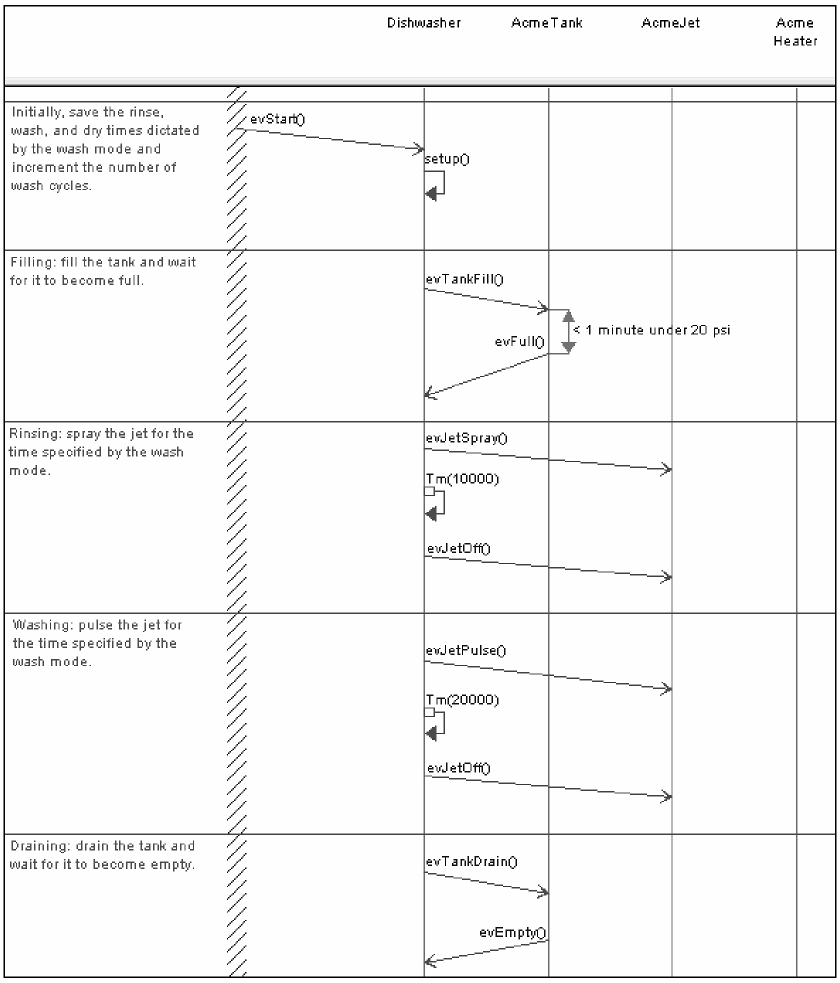
Rhapsody interface with the dishwasher Use Case



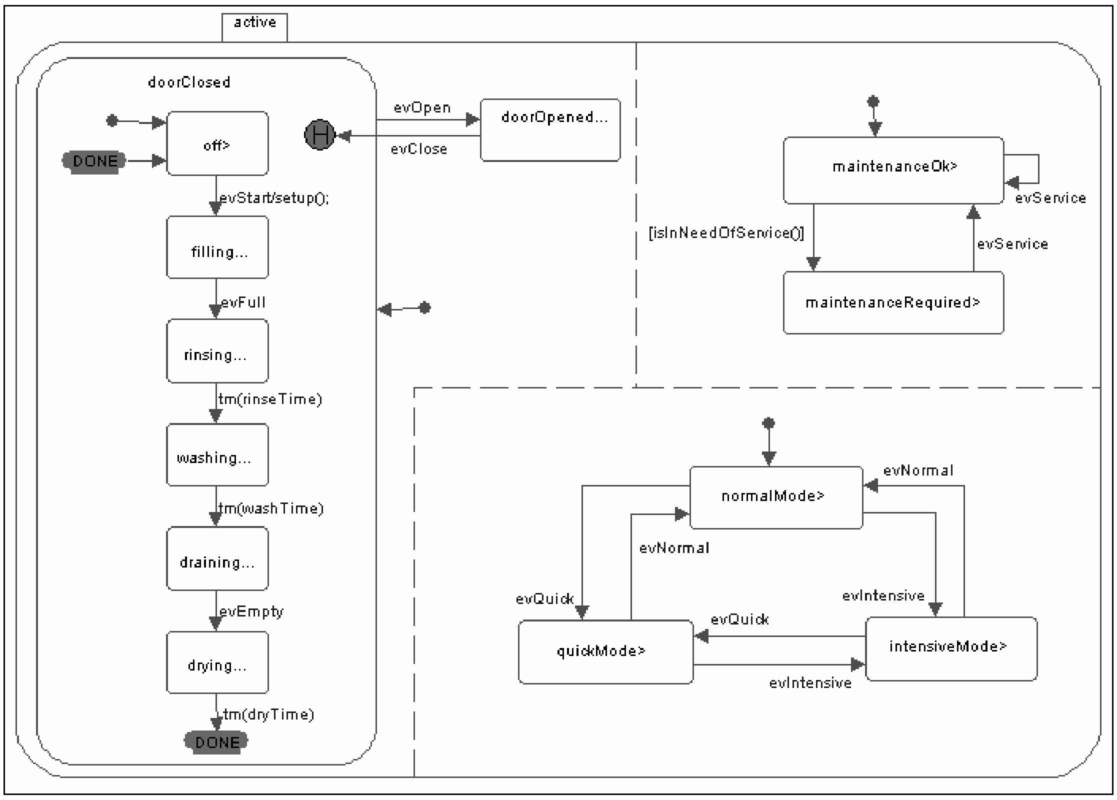
Class Diagram



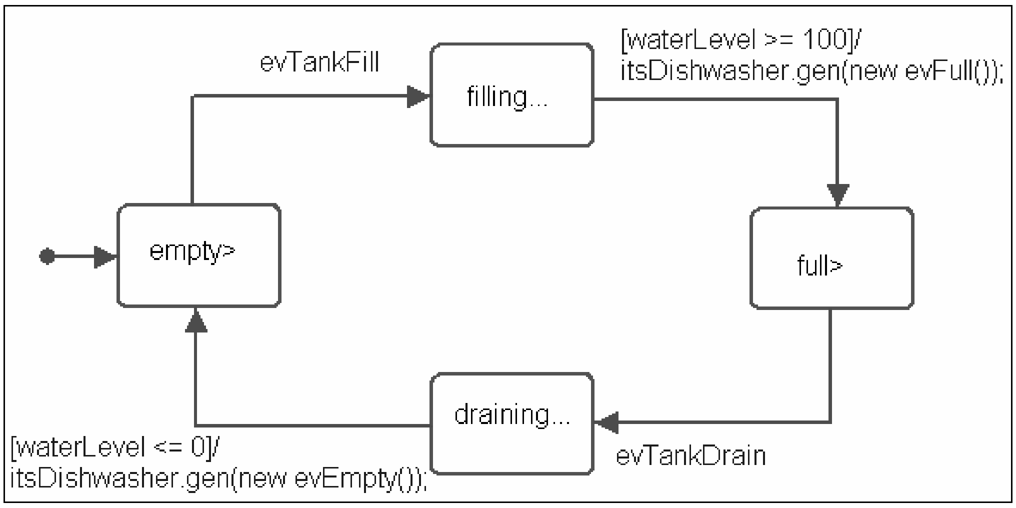
Class Diagram


Dishwasher
AcmeTank
AcmeJet
Acme Heater
Initially, save the rinse, wash, and dry times dictated by the wash mode and increment the number of wash cycles.
evStart()
setup()
Filling: fill the tank and wait for it to become full.
evTankFill()
< 1 minute under 20 psi
evFull()
Rinsing: spray the jet for the time specified by the wash mode.
evJetSpray()
Tm(10000)
evJetOff()
Washing: pulse the jet for the time specified by the wash mode.
evJetPulse()
Tm(20000)
evJetOff()
Draining: drain the tank and wait for it to become empty.
evTankDrain()
evEmpty()

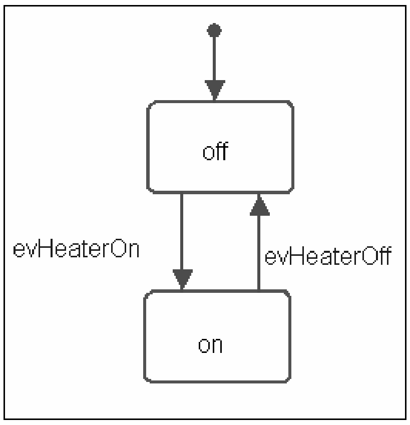
Sequence Diagram



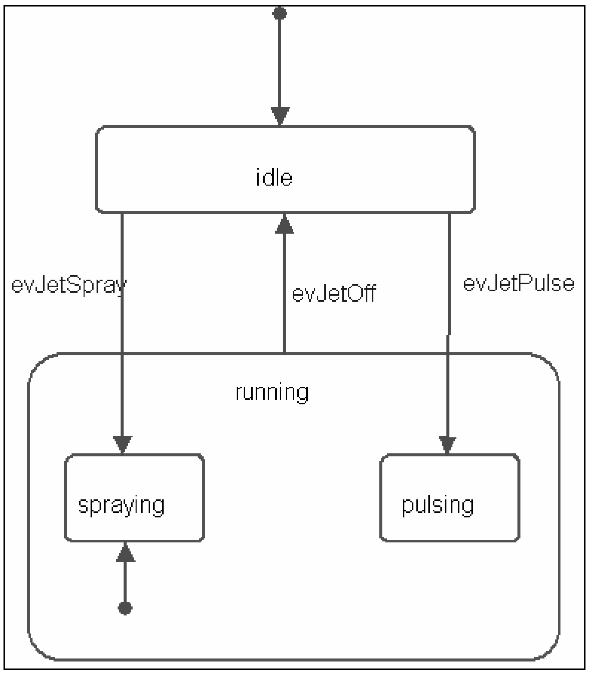
State Chart Diagram for the Dishwasher Class



State Chart Diagram for the Tank Class



State Chart Diagram for the Heater class



State Chart Diagram for the Jet class

Appendix C: An LSD account for the Dishwasher model

```
/*******************************************
    An LSD account for the dishwasher model
*******************************************/

Types:

     modeType  : enum(quick,normal,intensive)
     doorStatusType : enum(open,closed)
     progressType   : enum(go,filled,rinsed,washed,drained,dried,finished)
     tankStatusType : enum(tankEmpty,tankFull,tankSomewater)
     valveType : enum(valveClosed,valveOpen)
     jetStatusType  : enum(jetSpraying,jetPulsing,jetOff)
     jetDirectionType: enum(jetLeft,jetRight,jetMiddle)

agent Clock{
  state:
     (second)  time
     (bool)         pause
  protocol:
     pause == false -> time = time + 1
}

agent Dishwasher{
  state:
     (modeType)     mode
     (progressType) progress = go
     (second)  rinseTime
     (second)  washTime
     (second)  dryTime
     (second)  startTime = time
  oracle:
     time, door, waterLevelStatus, waterLevel, progress, mode
  derivate:
     rinseTime = (mode == quick)?1:((mode == normal)?2:8)
     washTime = (mode == quick)?1:((mode == normal)?2:8)
     dryTime = (mode == quick)?1:((mode == normal)?2:8)
     LIVE = progress != finished
  protocol:
     door == open && progress == dried -> progress = finished
     door == closed -> startTime = time
     progress==go && door==closed && waterLevelStatus==tankFull -> progress

= filled
     progress==washed && door==closed && waterLevelStatus==tankEmpty ->

progress = drained
     progress==filled && door==closed && (time-startTime) > rinseTime ->

progress = rinsed; startTime = time
     progress==rinsed && door==closed && (time-startTime) > rinseTime ->

progress = washed; startTime = time
     progress==drained && door==closed && (time-startTime) > rinseTime ->

progress = dried; startTime = time
}

agent Tank{
  state:
     (real)    tankCapacity
     (real)    waterFlowPerSecond
     (real)    waterLevel
     (real)    waterPercent
  oracle:
     progress, door, time
  derivate:
     waterPercent = waterLevel / tankCapacity * 100
     waterLevelStatus = (waterLevel==0)?tankEmpty:((waterLevel>=

tankCapacity)?tankFull:tankSomewater)
     drainValve = (progress==washed && door==closed && waterLevelStatus!=

tankEmpty)?valveOpen:valveClosed
     fillValve = (progress==go && door==closed && waterLevelStatus!=

tankFull)?valveOpen:valveClosed
  protocol:
     fillValve == valveOpen -> waterLevel = waterLevel + waterFlowPerSecond
     drainValve == valveClose && waterLevel > 0 -> waterLevel = waterLevel

- waterFlowPerSecond

}


agent Jet{
  state:
     (real)    jetRinseWaterPerSecond
     (real)    jetWashWaterPerSecond
     (real)    jetSecondPerPulse
     (jetStatusType)          jetStatus
     (jetDirectionType)  jetDirection

  oracle:
     progress, time, door
  derivate:
     jetStatus = (progress==filled)?jetSpraying:((progress==rinsed)?

jetPulsing:jetOff)
     jetDirection = (jetStatus==jetSpraying ||jetStatus==jetOff)?jetMiddle

:((jetStatus==jetPulsing && (time/jetSecondPerPulse)%2==1)?jetRight:jetLeft)
  protocol:
     progress==filled && door==closed -> waterLevel-jetRinseWaterPerSecond
     progress==rinsed && door==closed -> waterLevel-jetWashWaterPerSecond
}

agent Heater{
  state:
     (bool)    heaterOn
  oracle:
     progress
  derivate:
     heaterOn = progress == drained
}

agent Door{
  state:
     (doorStatusType)    door
  handle:
     pause
  protocol:
     door == open -> pause = true
     door == closed -> pause = false
}

agent User{
  oracle:
     Dishwasher.LIVE
  handle:
     mode, door
  protocol:
     !Dishwasher.LIVE -> Dishwasher()
     true -> mode = quick
     true -> mode = normal
     true -> mode = intensive
     true -> door = open
     true -> door = closed
}
```

Appendix D: The CPPS for a Crossnumber problem

```
/* CPPS for the Crossnumber problem */

grid =  [['x',' ',' ',' ',' '],
         [' ',' ',' ','x',' '],
         [' ','x',' ','x',' '],
         [' ','x',' ',' ',' '],
         [' ',' ',' ',' ','x']
       ];

numbers is ["1234","2125","26242","3992","4998","356","478","15","75"];

a1 is grid[1][1]; a2 is grid[1][2]; a3 is grid[1][3]; a4 is grid[1][4]; a5 is grid[1][5];
b1 is grid[2][1]; b2 is grid[2][2]; b3 is grid[2][3]; b4 is grid[2][4]; b5 is grid[2][5];
c1 is grid[3][1]; c2 is grid[3][2]; c3 is grid[3][3]; c4 is grid[3][4]; c5 is grid[3][5];
d1 is grid[4][1]; d2 is grid[4][2]; d3 is grid[4][3]; d4 is grid[4][4]; d5 is grid[4][5];
e1 is grid[5][1]; e2 is grid[5][2]; e3 is grid[5][3]; e4 is grid[5][4]; e5 is grid[5][5];

blocks  is [a1,b4,c2,c4,d2,e5];

position1 is [a2,a3,a4,a5];
position2 is [b1,b2,b3];
position3 is [d3,d4,d5];
position4 is [e1,e2,e3,e4];
position5 is [b1,c1,d1,e1];
position6 is [a2,b2];
position7 is [a3,b3,c3,d3,e3];
position8 is [d4,e4];
position9 is [a5,b5,c5,d5];

ok1 is containsString(numbers, digitsToString(position1));
ok2 is containsString(numbers, digitsToString(position2));
ok3 is containsString(numbers, digitsToString(position3));
ok4 is containsString(numbers, digitsToString(position4));
ok5 is containsString(numbers, digitsToString(position5));
ok6 is containsString(numbers, digitsToString(position6));
ok7 is containsString(numbers, digitsToString(position7));
ok8 is containsString(numbers, digitsToString(position8));
ok9 is containsString(numbers, digitsToString(position9));


set1 is findNumberWithConstraints(numbers, digitsToString(position1));
set2 is findNumberWithConstraints(numbers, digitsToString(position2));
set3 is findNumberWithConstraints(numbers, digitsToString(position3));
set4 is findNumberWithConstraints(numbers, digitsToString(position4));
set5 is findNumberWithConstraints(numbers, digitsToString(position5));
set6 is findNumberWithConstraints(numbers, digitsToString(position6));
set7 is findNumberWithConstraints(numbers, digitsToString(position7));
set8 is findNumberWithConstraints(numbers, digitsToString(position8));
set9 is findNumberWithConstraints(numbers, digitsToString(position9));


stuck is set1#==0 || set2#==0 || set3#==0 || set4#==0 || set5#==0 || set6#==0 || set7#==0
|| set8#==0 || set9#==0;
solved is ok1 && ok2 && ok3 && ok4 && ok5 && ok6 && ok7 && ok8 && ok9;

proc monitorOks: ok1, ok2, ok3, ok4, ok5, ok6, ok7,ok8,ok9 {
     writeln("oks:",ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8,ok9);
}

proc monitorProgress:solved, stuck{
     writeln("stuck:", stuck);
     writeln("solved:", solved);
}

proc monitorSets: set1,set2,set3,set4,set5,set6,set7,set8,set9{
    writeln("ok:",ok1," set1:",set1);
    writeln("ok:",ok2," set2:",set2);
    writeln("ok:",ok3," set3:",set3);
    writeln("ok:",ok4," set4:",set4);
    writeln("ok:",ok5," set5:",set5);
    writeln("ok:",ok6," set6:",set6);
    writeln("ok:",ok7," set7:",set7);
    writeln("ok:",ok8," set8:",set8);
    writeln("ok:",ok9," set9:",set9);

}

proc monitorGrid: grid{
    writeln("--------");
    writeln(grid[1]);
    writeln(grid[2]);
    writeln(grid[3]);
    writeln(grid[4]);
    writeln(grid[5]);
}

proc monitorNumbers: numbers{
    writeln("numbers:",numbers);
}

func containsString{
    para numbers, thestring;
    auto i;
    for(i=1;i<=numbers#;i++){
      if(numbers[i]==thestring)return 1;
    }
    return 0;
}

func digitsToString{
     para digits;
     auto result, i;

     result="";
     for(i=1;i<=digits#;i++){
        result = result // digits[i];
     }

     return result;
}


proc assign{
    para row, column, direction, number;

    auto i;
    autocalc=0;
    for(i=0;i<number#;i++){
       if(direction==0) grid[row][column+i] = number[i+1];
       else if(direction==1) grid[row+i][column] = number[i+1];
    }
    autocalc=1;
}

/* observations of the given set of numbers */
lengthStatistics is extractLengthStatistics(numbers);

func extractLengthStatistics{
     para numbers;
     auto result,i;

     result=[0,0,0,0,0];
     for(i=1;i<=numbers#;i++){
        result[numbers[i]#]++;
     }
     return result;
}

proc monitorLengthStatistics: lengthStatistics{
     writeln("lengthStatistics:", lengthStatistics);
}
```

```
/* strategies */

func findNumberWithConstraints{
     para numbers, constraints;
     auto result,i,j,statisfied;

     result=[];

     for(i=1;i<=numbers#;i++){
        if(numbers[i]# == constraints#){

             statisfied=1;

             for(j=1;j<=numbers[i]#;j++){
                if(constraints[j]!=' '){
                   if(numbers[i][j]!=constraints[j]){
                       statisfied=0;
                       break;
                   }
                }
             }
             if(statisfied==1)
                result = result // [numbers[i]];
        }
     }
     return result;
}
```

Appendix E: Some useful information from the literature

Table 1: Assumptions and ideals of methodical and amethodical texts

[Tru00] Table 1: Assumptions and ideals of methodical and amethodical texts.

Privileged methodical text	**Marginalized amethodical text**
1. Information systems development is a managed, controlled process idealizing logical decomposition reductionism	2. information systems development is random, opportunistic process driven by accident idealizing holism creativity
3. Information systems development is a linear, sequential process idealizing temporal causal chain	4. Information systems development processes are simultaneous, overlapping and there are gaps idealizing fragmentation parallelism disconnectedness
5. Information systems development is a replicable, universal process idealizing generalization consistency formalisms	6. Information systems development occurs in completely unique and idiographic forms idealizing choice change adhocracy
7. Information systems development is a rational, determined, and goal-driven process idealizing goal predetermination process predetermination human cooperation	8. Information systems development is negotiated, compromised and capricious idealizing conflict social constructivism human independence

Table 2: The rational problem solving paradigm and the reflect-in-action paradigms summarized

[Dor95] Figure 1: The rational problem solving paradigm and the reflect-in-action paradigms summarized.

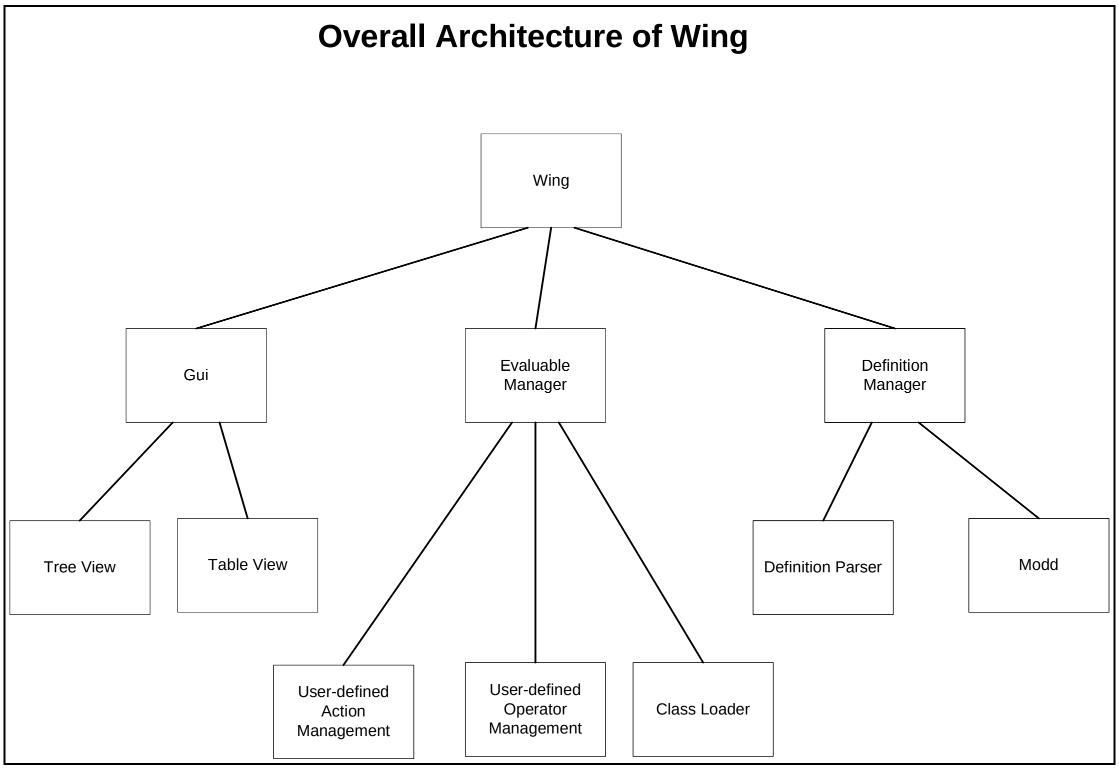
	Rational problem solving	**Reflection-in-action**
Designer	information processor (in an objective reality)	person constructing his/her reality
Designer problem	ill defined, unstructured	essentially unique
Design process	a rational search process	a reflective conversation
Design knowledge	knowledge of design procedures and ‘scientific’ laws	artistry of design: when to apply which procedure/piece of knowledge
Example/Model	optimization theory, the natural sciences	art/the social sciences

Appendix F: Technical overview of WING

We use Java (JDK1.4) as the programming language to develop WING because of its cross-platform capability and richness of graphics and user-interface library. The interpreter is developed by using JavaCC 0.7. The windowing and graphics functionality is implemented by using Java Foundation Class 1.1.

1. Overall architecture

The diagram showed below is an abstract view of the overall architecture of WING. Each rectangle box represents an abstract functionality to its lower level down to the hierarchical structure.



From the above diagram, WING can be divided into three main components namely *GUI*, *Definition Manager* and *Evaluable Manager*. The *GUI* component provides a user-friendly graphic user interface for user to interact with the system. The most important parts of are:

- *Tree View* – provides a directory tree-like view of definitions and containers, and

allows the user to select a definition or a containers through navigating the tree structure by using mouse pointer.
- *Table View* – visualises definitions in the current container as a table of cells like a spreadsheet. It allows interactive definition editing by just double-clicking the corresponding cell.

The *Evaluable Manager* provides the following functions:

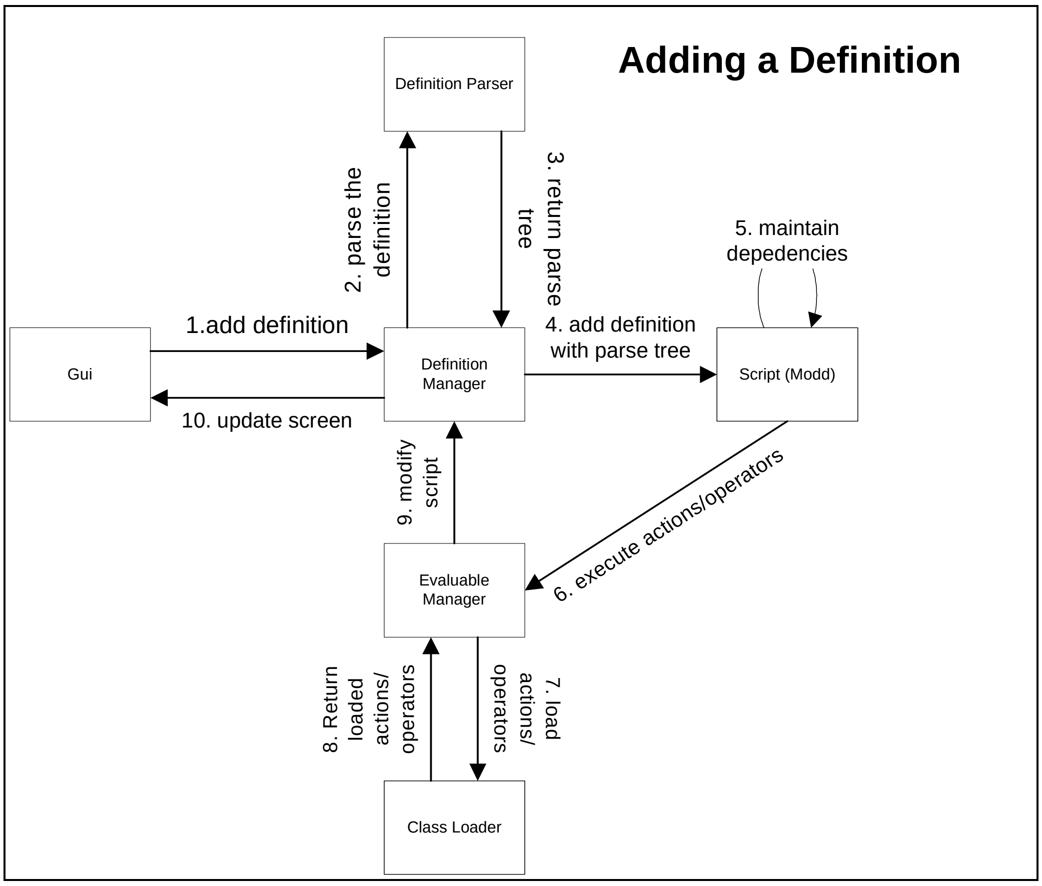
- *User-defined Action Management* – allows user to add, change, delete, compile, load and save actions.
- *User-defined Operator Management* – allows user to add, change, delete, compile, load and save user-defined operators.
- *Class Loader* – allows user to compile and bind Java actions and operators dynamically to the system.

The *Definition Manag*er provides basic functions to manipulate the definitions. It contains the following important components:

- Definition Parser – builds parse tree for each definition. A parse tree is the main communication entity between the system and Modd.
- Modd (Maintainer of Dynamic Dependency) API – maintains dependencies between definitions (developed by Gehring in the EM research group [Geh98])

2. Adding a definition

Referring to the architecture described in the last section, here is a scenario of adding a definition into the system:



As shown in the above diagram, The typical steps of adding a definition into the system are:

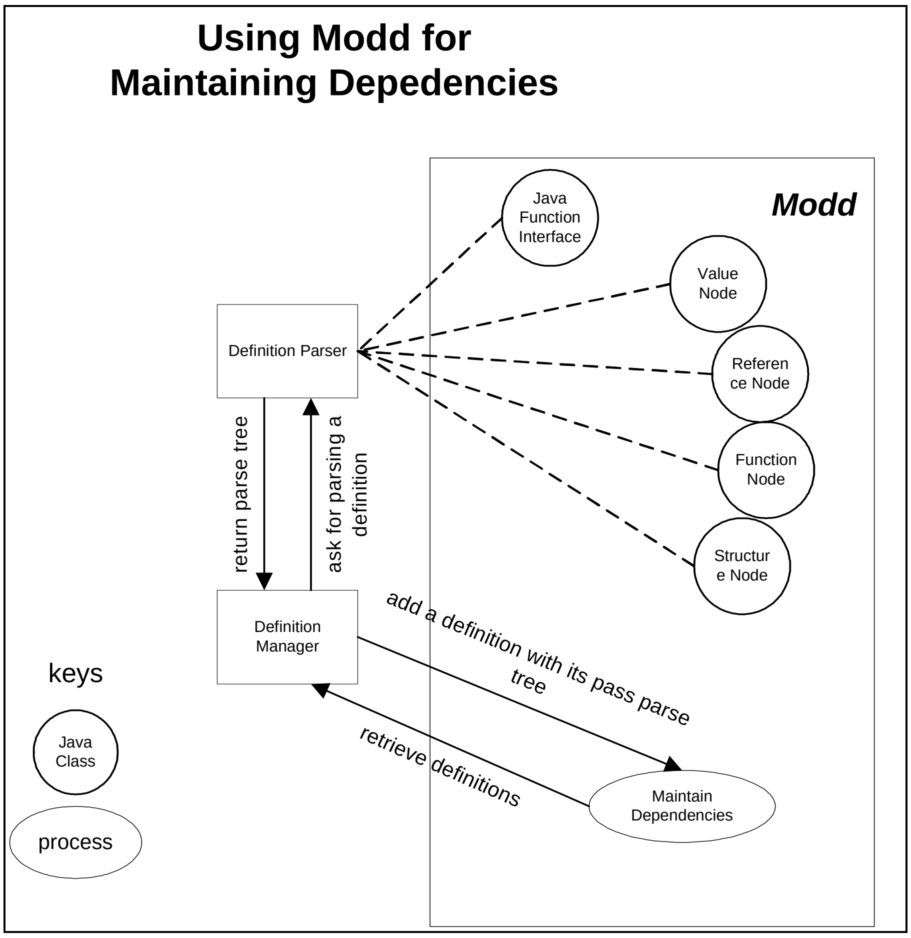
1. GUI passes the definition to the Definition Manager.
2. Definition Manager passes the definition to the Definition Parser
3. Definition Parser builds a parse tree for the definition and returns it to Definition Manager. (Assuming that there is no parse error for this definition)
4. Definition Manager adds the definition with its parse tree to the Script(the most import class in Modd)
5. Script maintains the dependencies among all definitions by evaluating a set of definitions that should be updated (the algorithm determining which definition should be updated in Modd implemented by Gehring)
6. Evaluating the definitions in Modd triggers some user-defined actions and operators that will be executed by Evaluable Manager.
7. Evaluable Manager asks ClassLoader to load the actions and operators from files.
8. ClassLoader returns the loaded actions and operators to Evaluable Manager.
9. Evaluable Manager executes the actions and operators. Actions will be executed

according to their priorities. Some of the actions may request further modification of the script. The requests are passed to Definition Manager.

10. GUI updates the screen in react to changes of the script.

3. Using Modd for maintaining dependencies

WING uses Modd to maintain the dependencies among definitions. The following diagram shows the main entities of Modd:



As shown in the above diagram, the Definition Parser uses a variety of parse tree nodes provided by Modd to build a parse tree for each definition. The tree nodes are implemented as Java classes. Functionalities of different kinds of node are:

- Value node – stores the value of a certain data type.
- Reference node – stores a reference to a definition.
- Function node – stores a function implementation. In WING, it is either an action

or an operator.

- Structure node – stores a list structure similar to EDEN. Since Structure node is still under development, WING implements its own list structure called vectors.

All parse trees that built by using the nodes above are passed to Modd from Definition Manger at run-time. Modd uses the parse tree actions stored in Function node to update the definitions, i.e. to maintain dependencies of definitions.

The parse tree actions stored in Function Node are implemented by using Java Function Interface. Therefore, the task that should be done before we could use Modd is to implement Function Interface for every operator of all data types. All Java files started with prefix “F_” in Appendix of code listing are for this purpose.

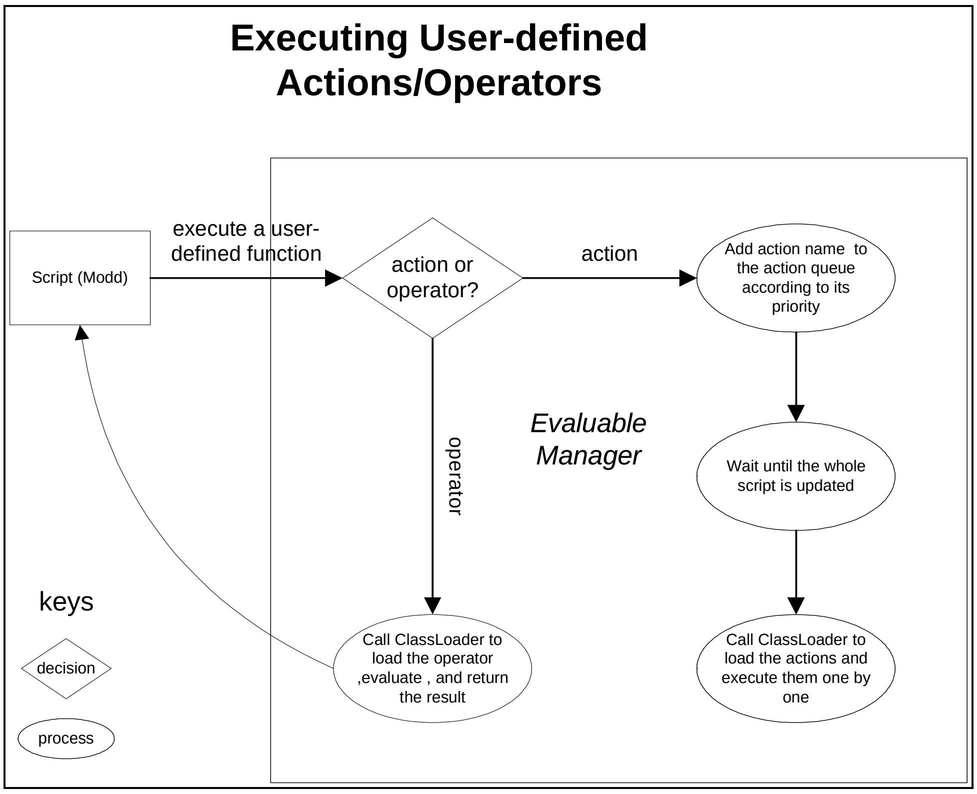
A guide for how to use Modd written by the author of Modd could be found in [Geh98]. However, we summaries the steps to use Modd related to the implementation of WING are as follows:

1. Identify data types and operators.
2. Implement data types using Java.
3. Implement operators by using Function Interface provided by Modd.
4. Implement parse actions in Definition Parser to build parse trees using:
   - Data type implemented in step 2.
   - Function Interface implemented in step 3.
   - Tree nodes provide by Modd.

At run-time, WING’s Definition Manager passes parse trees generated by Definition Parser to the Modd for maintaining the dependencies among definitions.

4. Executing user-defined actions/operators

One of the unique features of WING is its capability to allow the user to implement new actions and operators using Java. The following diagram shows procedures to execute an action or an operator:


Executing User-defined Actions/Operators
Script (Modd)
execute a user-defined function
action or operator?
action
Add action name to the action queue according to its priority
operator
Evaluable Manager
Wait until the whole script is updated
Call ClassLoader to load the operator ,evaluate , and return the result
Call ClassLoader to load the actions and execute them one by one
keys
decision
process

We summarise the essential points in the diagram as follows (note that Script is a class in Modd that maintains dependencies between definitions):

- When Script encounters a user-defined function name, it will pass the name and corresponding parameters to Evaluable Manager of WING for executing the function.
- Evaluable Manager determines whether the function is an action or an operator.
- If it is an operator, Evaluable Manager calls ClassLoader to load the operator, then execute the operator. The result is returned back to the Script.
- If it is an action, Evaluable Manager adds the action name to the actions queue according to the priority of the action. The Evaluable Manager waits until all definitions are updated. Then, it calls ClassLoader to execute actions in the queue one by one.

In the next two sections, two very useful Java programming techniques are documented. They are the methods used by WING for dynamic compiling and loading of actions/operators. They could be served as reference to implement similar functionality for other Java definitive systems in the future.

5. Dynamic compiling of Java code

To compile Java code at run-time, a class needs to have the following line in its implementation:-

```
import sun.tools.javac.Main;
```

Then, given name of the Java file, we could compile it by using following codes:-

```
boolean   state;
Main compiler=new Main(System.out, "javac " +path);
state = compiler.compile(filename);
```

where `path` is the directory name where the Java file is located. After execution of above codes, the value of state indicates whether the compilation is successful or not. If the value of state is true, the compilation is successful. If the value of state is false, there is parser error. The error messages is displayed in the standard output.

6. Dynamic loading of Java class

Techniques for dynamic loading of Java class is difficult, however McManis, in [Mcm97], provides a template to implement a Java class loader. WING modifies the template to achieve dynamic loading of actions and operators. The implementation of WINGClassLoader could be found in the file WINGClassLoader.java. The main steps for implementing a class loader are summarised as below:

1. extend the java.lang.ClassLoader.
2. implement the abstract method loadClass(). The flow of this method is as follows:
    - check if the class name is valid or not
    - check if the class has already been loaded
    - check if the class is a “system class”
    - fetch the class from class loader’s repository if possible
    - define the class for the Virtual Machine
    - resolve the class
    - return the class to the caller

For details, please refer to WINGClassLoader.java.

7. BNF of the definition notation of WING

The DefinitionParser of WING is generated by using JavaCC. The BNF of it is as follows: -

```
one_line  ::=  logical <EOL>
               |<EOL>
               |<EOF>
logical   ::=  relation ( ( <AND> | <OR> | <XOR> ) relation )*
relation  ::=  sum ( ( <SMALLER> | <SMALLER_EQUAL> | <EQUAL> |
     <GREATER>
               | <GREATER_EQUAL> | <NOT_EQUAL> ) sum )*
vector    ::=  "{" logical ( "," logical )* "}"
string    ::=  <STRING>
sum       ::=  term ( ( <PLUS> | <MINUS> ) term )*
term      ::=  exp ( ( <MULTIPLY> | <DIVIDE> ) exp )*
exp       ::=  unary ( <EXP> exp )*
unary          ::=  <MINUS> select
               |select
select    ::=  element ( <SELECT> select )*
element   ::=  <CONSTANT>
               |<TRUE>
               |<FALSE>
               |function
               |"(" logical ")"
               |vector
               |string
function  ::=  <ID> "(" ( logical ( "," logical )* )? ")"
               |<ID>
```

The functionality of DefinitionParser is only to parse the left hand side of a definition and build parse tree by using nodes provided by Modd. Other parts of notation syntax such as for variable name is done by GUI of WING.

8. Reference

[Geh98] D. K. Gehring, *Modd documentation*, Online at: http://www.dcs.warwick.ac.uk/~gehring/modd, 1998.

[Mcm97] C. McManis, The basics of Java class loaders, *Java In Depth*, Online at: http://www.javaworld.com, 1997.

Appendix G: Technical Overview of EME

EME is a MS Windows application developed using Visual C++ 6.0. It uses the Microsoft Foundation Classes (MFC). Many classes in the implementation are derived from MFC.

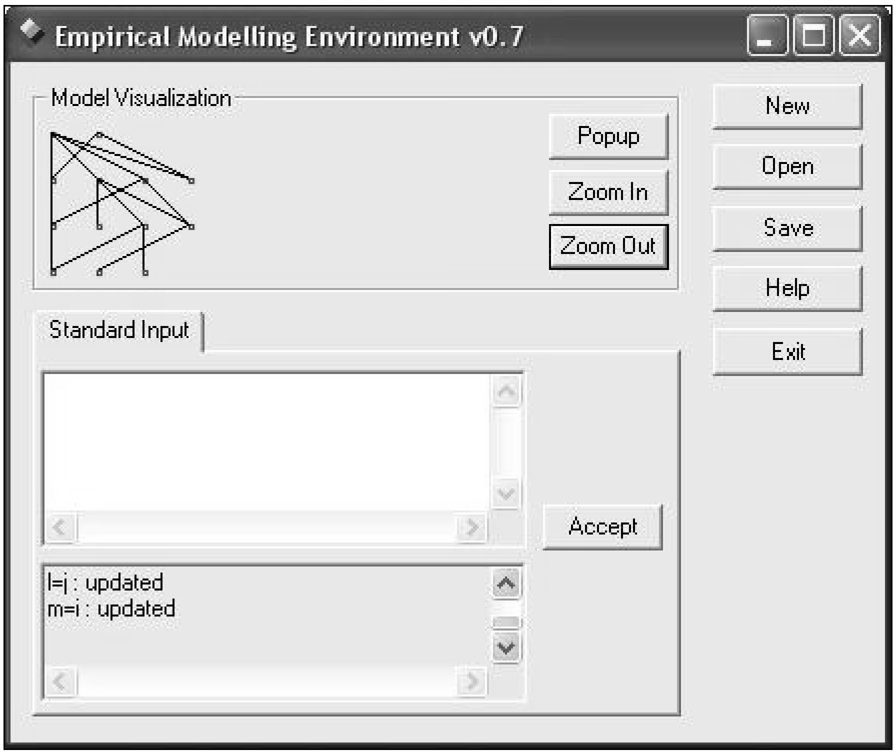
The core part of EME is an interpreter. The interpreter contains two parsers that are generated by using Parser Generator 1.11 from Bumble-bee Software Ltd. The Parser Generator generates C++ parser classes according to grammar specifications very similar to Lex and Yacc in Unix. For the visualisation of the model, EME makes use of the LEDA library. It is a library of the data types and algorithms of combinatorial computing. LEDA simplifies the process of implementing graph algorithms that are used for displaying the dependency graphs, checking cyclic dependencies and determine order of the evaluations.

1. Overall architecture

EME contains about 25 classes. They can be categorised into the following six groups:

- The interface classes
- Interpreter classes
- Symbol table classes
- Parse node classes
- Windowing classes
- Communication classes

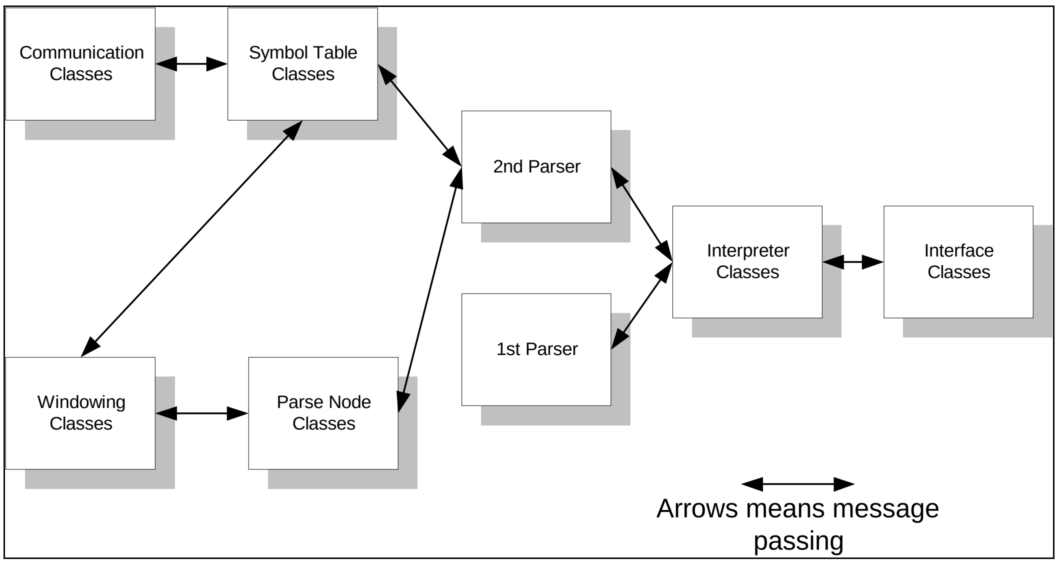
The **interface classes** implement a simple graphical user interface (as shown in the screen capture below) that a user can interact with the tool. The interface can be divided into two areas. The top area shows a model visualisation of data dependency graph with buttons of adjusting the size of the graph, and a popup button which is for opening a bigger window showing the graph in a more detail. The bottom area is for script input similar to the ‘input window’ of TkEden.



```
l=j : updated
m=i : updated
```

The **interpreter classes** implement one Lexer and two Yacc parsers. Two parsers use the same Lexer to get tokens from the input script but process the tokens in different ways that will be discussed in the following sections. The whole interpretation process of a script contains two passes, each pass uses a different parser.

The **symbol table classes** implement how definitions are stored in the memory. It also contains evaluation algorithms and graph layout algorithms. The **parse node classes** implement all the data types and their operations. The interpreter creates parse trees by using these nodes. **Windowing classes** are classes derived from MFC windowing classes that help to manage the side effects triggered by using windowing components. For example, a button should introduce a “button-clicked” definition to the symbol table when it is clicked. Finally, the **communication classes** are responsible for the peer-to-peer communication among EME application instances when we build a distributed model. The figure below shows the overall architecture of EME with arrows denoting interaction between classes.



2. The interpreter

The underlying interpreter contains one tokeniser and two parsers. All of them are generated by a C++ compiler compiler called Parser Generator. The whole interpreting process can be divided into two passes. Each parser is responsible for one pass of the process.

In the 1st pass, the syntax of inputting script is checked. All the branching positions in script are also determined and saved for later use in the 2nd pass. In the 2nd pass, the parser builds parse trees for definitions and calls symbol table to evaluate and store the definitions. It uses the branching information saved from the 1st pass to jump over a block of script if needed. For example, the logical expression of an if statement is checked. If the expression turns out to be false, we need a jump to the end of the if statement without executing anything enclosed in the if brackets. To understand this, we could imagine there is a high level program counter in the tokeniser that scans input token by token and each token is consumed and executed by the parser. The position of the program counter is at the beginning of next token. The parser tells the tokeniser when a jump over a block of scripts is needed. The position where the program counter will be jumped to is determined in the 1st pass of the interpretation process.

Here is the grammar used for both parsers:

```
lines     : lines line
| /* empty */
     ;
```

```
line : statement
     | error '\n'
     | '\n'
     ;

statement : definition
          | systemcommand
     | control
          | procedurespec
          ;
procedurespec  : ACTION var '{' lines '}'
          | FUNCTION var '{' lines '}'
               ;

definition     : var '=' rhs ';'
          ;

systemcommand  : '?' var ';'
          | UNDEFINE var ';'
     | LISTEN '(' NUMBER ')' ';'
          | CONNECT '(' expr ',' expr ',' expr ',' NUMBER ')' ';'
     | TESTSEND '(' expr ',' expr ')' ';'
          ;

control        : REPEAT '(' logicalexpr ')' '{' lines '}'
          | IF '(' logicalexpr ')' '{' lines '}'
          | IF '(' logicalexpr ')' '{' lines '}' else '{' lines '}'
          ;

var       : id
          | var '@' id
;

id        : ID
          | '<' expr '>'
     | '[' expr ']'
     | id '\\' NUMBER
| id '\\' ID
     | id '<' expr '>'
     | id ID
     | id '[' expr ']'
          ;

rhs       : expr
          ;

expr      : expr '+' expr
          | expr '-' expr
     | expr '*' expr
          | expr '/' expr
     | '(' expr ')'
     | '{' expr '}'
     | logicalexpr
          | NUMBER
     | USERTEXT
     | var
          | SIN '(' expr ')'
     | var '(' parameterlist ')'
     | var '(' ')'
          ;

parameterlist  : expr
          | parameterlist ',' expr
          ;

logicalexpr    : expr EQUAL expr
          | expr NOTEQUAL expr
     | expr SMALLEREQUAL expr
          | expr GREATEREQUAL expr
     | expr SMALLER expr
     | expr GREATER expr
     | NOT expr
          ;
```

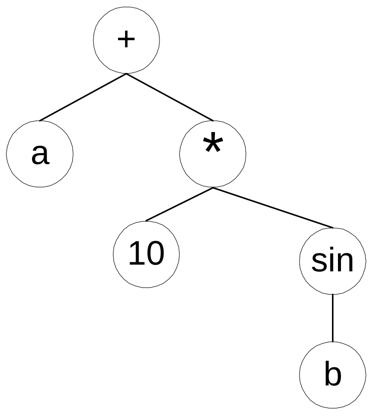
The grammar above is provisional. For example, in the current implementation, only Sin function among all the common trigonometric functions that can be implemented. The grammar here is for testing the functionalities of EME. There will be a big enhancement of the grammars in the release version of EME.

It is worth noting that, along the process of interpreting a definition, it is very important to know all the determinants that the definition is dependent on. The determinants are references to other definition variables. In a definitive system like EME, the set of determinants of a definition is dynamic. For example, in the definition x=a[i]; , the determinants of this definition is dependent on the current value of definition i. Therefore, in EME, the set of determinants of a definition will be determined not in interpreting time but every time when the definition is evaluated.

3. Building parse trees

The main purpose of the interpreter is to build parse trees that can be evaluated every time when the value of definitions need to be updated. A parse tree consists of links of parse nodes of many different types. In the current version of EME, there are about 25 types of parse nodes. They can be grouped into terminal nodes and non-terminal nodes. Typically, terminal nodes are nodes that contain primitive values of EME data types; non-terminal nodes contain operation instructions on the primitive nodes. Here gives an example of a parse tree of the definition:

x = a + 10 * sin(b);



In the above example, the terminal nodes are a,10 and b, whereas non-terminal nodes as +, * and sin function.

The advantage of building parse trees and storing them with the definition is that a script needs to be scanned once only. There is no need to interpret a definition every time when the value of it needs to be updated. The disadvantage is parse nodes are objects that are created dynamically and stored in memory. Therefore, memory usage in a large model will be an issue.

The evaluation of parse trees is implemented in the parse tree classes. The type of a parse node is represented as a unique integer number. Evaluation of a parse tree, typically consist of a sequence of recursive calls to evaluation operation implemented in parse node class. Along with the evaluation process, all the variable expressions are evaluated and variable references are obtained. The variables referenced in this process are recorded and stored as determinants of the definitions.

Implementation of most of the data types and their operations, like number and its arithmetic operations, is straightforward. However, when it comes to implementing data types with side effects, extra care should be made. Examples are the windowing types. A windowing type, such as a push button on the screen, creates two kinds of side effect – the side effect of displaying it on the screen and the side effect of event handling. To deal with these side effects, a wrapper class is created for each windowing type. The wrapper classes contain codes of putting and removing the windowing component on the screen and generating definitions when a user triggers a window event.

4. The symbol table

The **symbol table** contains a MFC implementation of hash table of **symbols**. Each of these symbols contains the following information:

- Name – variable name of the definition.
- A parse tree – parse tree of the definition created by the interpreter.
- Up-to-date flag – indicates whether the value of this symbol is up-to-date or not.
- Value – stores a parse node representing the value returned by evaluation of

its parse tree.

- String representation – a string representation of the definitions
- Dependents – array of variable names that are dependent on this definition.
- Determinants – array of variable names that this definition refers to.

Therefore, each symbol encapsulates detailed information about a definition, which is needed in the whole process of definition evaluation. The symbol table also owns:

- A dependency graph object – a LEDA graph object that stores the data dependency graph for visualisation.
- A communication sever object – responsible for network communication among instances of EME application.
- An action queue – stores a queue of actions generated in the process of evaluation. Actions in this queue will be executed one by one after all the values of definitions are up-to-date.

To help understand how the symbol table works, we list out briefly steps involved in defining a definition:

1. A definition is entered as a string to the Standard input.
2. The 1st pass of the interpreter makes sure there is no syntax error.
3. The 2nd pass of the interpreter builds parse tree while scanning the definition string.
4. The interpreter passes the variable name and its parse tree to the symbol table.
5. The symbol table evaluates the parse tree to determine all the determinants of this definition.
6. The symbol table does a topological sort on the dependency graph with the newly created determinant information. This checks cyclic dependency and also obtains a topological order of variables according to their dependencies. If there is cyclic dependency, the process stops and an error message will be generated.
7. The symbol table registers this definition as dependents of the determinants that are obtained in step 5.
8. The symbol table evaluates, in topological order, all the definitions that are registered at dependents of the definition.
9. During the evaluation process, actions are added to the action queue.

10. After all values of definitions are updated, actions in the action queue are executed one by one.

The symbol table also updates visualisation of the data dependency graph during the above process.

5. Algorithms applied

In this section, we will describe three algorithms applied in different stages of execution in EME. They are topological sort algorithm, one-way constrain satisfaction algorithm and graph layout algorithm.

a) Topological sort algorithm

We have used a modified version of topological sort algorithm from [Näh99] This algorithm is for checking cyclic dependency and determining evaluation order. The algorithm is listed in form of LEDA C++ code:

```
bool CSymboltable::TopologicalSort(GRAPH<string,int>* G, CMap<CString,LPCSTR,int,int>
&toporder)
{

     /* initialization:
     determine the indegree of all nodes and initialize a queue
     of nodes of indegree zero*/
     node_array<int> INDEG(*G);
     queue<node> ZEROINDEG;
     node v;
     forall_nodes(v,*G)
     if ( (INDEG[v] = (*G).indeg(v)) == 0 ) ZEROINDEG.append(v);

     /* removing nodes of indegree zero:
     consider the nodes of indegree zero in turn. When
     a vertex v is considered we number it and we decrease
     the indegrees of all adjacent nodes by one. Nodes whose
     indegree becomes zero are added to the rear of ZEROINDEG*/
     node w;
     edge e;
     int count = 0;

     while (!ZEROINDEG.empty())
     {
          v = ZEROINDEG.pop();
          toporder[(*G)[v]]=++count;
          forall_out_edges(e,v)
               {    w = (*G).target(e);
                    if ( --INDEG[w] == 0 ) ZEROINDEG.append(w);
               }
     }
     return (count == (*G).number_of_nodes());
}
```

This operation returns false if a cyclic dependency is detected. If it returns true, the toporder map variable containing a mapping from variable name to an integer stores the topological order of the variable.

b) Evaluation algorithm

During the evaluation process, we have applied a modified version of one-way constrain satisfaction algorithm shown below. The algorithm updates values of the definitions in topological order. More detail explanation of this algorithm can be found in [Zan01].

```
BOOL CSymboltable::UpdateAllinTopologicalOrder(CSymbol* start)
{
     CMap<CString,LPCSTR,int,int> toporder;

     p_queue<int,CSymbol*> Q; // a priority queue
     int order,n,i;
     CSymbol *s, *d;
     pq_item item;

     Q.insert(1,start);

     while(!Q.empty()){
          item=Q.find_min();
          s=Q.inf(item);

          if(!Update(s,toporder)) return FALSE;
          n=s->m_dependents.GetSize();
          for(i=0;i<n;i++){
               d=Lookup(s->m_dependents[i]);
               ASSERT(d!=NULL);
               if(d->m_uptodate==true)d->m_uptodate=false;
               toporder.Lookup(d->m_name,order);
               Q.insert(order,d);

          }

          Q.del_item(item);

     }

     return TRUE;
}
```

c) drawing of dependency graph

The difficulty of drawing a dependency graphs is how to reduce crossings of edges and arrange the position of the nodes so that the graph drawn is more comprehensible. There are many graph layout research on drawing acyclic graphs. One of the famous algorithms is called Sugiyama Algorithm [Sug81]. We list out the step on performing the algorithm as below.

- Step 1: Convert the graph in into a “proper” hierarchy. If the hierarchy has long span edges, it is converted into a proper hierarchy by adding dummy nodes and edges.
- Step 2: The number of crossings of edges in the proper hierarchy is reduced by permuting orders of nodes in each level.
- Step 3: Horizontal positions of nodes are determined according to a given set of rules.
- Step 4: The graph is drawn with the dummy nodes removed.

The above steps are very brief description of what the algorithm does. For details of this algorithm, please refer to Sugiyama’s paper [Sug81]. The progress of implementing the algorithm, at the time of writing this report, is still on the first step. At the moment, the data dependency graph drawn by EME is without edge crossing minimisations. Therefore, the data dependency graph drawn is only comprehensible for defining a small set of definitions.

6. Network communication

The implementation of network communication contains two classes. One implements a server and the other implements a client. They are all derived from CSocket in MFC. CSocket provides basic primitives for synchronise communication in TCP/IP network. EME is designed to use peer-to-peer communication model where a centralised server is not necessary. Each EME application instance is both a server and a client.

The sequences of making a connection between two EME application instances are shown below. Suppose we have two instances of EME running on two different machines connected in a computer network. We call them A and B. Just like telephone a call, one of these instances should be the caller and the other should be the receiver. However, after the connection is made, there is no difference between the caller and receiver - both can communicate freely through the connection, and if one of them closes the connection, the whole communication session will be closed.

If A calls B to make a connection, these steps of setting up the connection will start:

1. Initialise B’s server to listen a port – it can be done by the command

listen(4000);. This initialises B to listen the port number 4000.

2. A client of A calls the server at B by the connect command – for example connect("apple","banana","gem.dcs.warwick.ac.uk",4000); where the first two arguments are logical names of A and B for this connection, i.e. A recognises this connection is made to "banana" and B recognises the connection is made to "apple".
3. The server at B receives the connection request from a client from A.
4. The server at B creates a new client object that serves the client from A.
5. When the connection is made, both A and B are using a client to communicate with each other.

After the connection is made, A and B can send messages to each other. The types of these messages are ranging from simple string messages to symbol objects that store information on definitions. For example, when the determinants of a definition are obtained during the evaluation of the definitions, and one of the determinants is a remote variable that contains a @ character, the system will send an add-dependent message to the remote system. This add-dependent message registers the variable as a dependent of the remote variable so that when the remote variable's changed, this variable will be informed. This mechanism forms part of procedures in dependency maintenance of a distributed model.

8. Saving a model

In EME, we save a model into a persistent storage as states of objects from the memory. Instead of saving scripts, the system saves the whole symbol table that contains all the states and dependencies of the model. This includes saving of the names, up-to-date statuses, parse trees, string representation of the definitions in the model. Therefore, loading the model does not involve reinterpreting everything again. EME uses the Serialisation facilities in MFC library to archive this.

9. References

[Näh99] S. Näher, K. Mehlhorn, *LEDA A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, Germany, 1999.

[Sug81] K. Sugiyama, S. Tagawa, M. Toda, Methods for Visual Understanding of Hierarchical Systems Structures, In *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. Smc-11, No. 2, February 1981.

[Zan01] B. T. V. Zanden, R. Halterman, B. A. Myers, R. McDaniel, R. Miller, P. Szekely, D. A. Giuse, D. Kosbie, Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits, In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v23n6, November 2001.

Appendix H: DMT interface menu reference

Model (all menu options relate to DMT models)

- New - Closes the existing model and creates a new one.
- Load - Loads a model file.
- Save As - Saves a model file. Suggested extension of the file is ".dmt".
- Statistics - Displays some statistics of the current model.
- Print - Outputs the model to a printer. Currently, it can only print one page. You are encouraged to use zoom functions to reduce the size of a large model before printing.
- Exit - Exits the program

Script (all options refer to Eden scripts)

- Input window - Allows user to type in Eden scripts.
- Direct import - Imports an Eden script directly into the current model.
- Export selection - Exports the selected definitions to a text window.

Layout

- Hierarchical-up - Layouts the model in natural evaluation order. Leaves are at the bottom of the layout.
- Hierarchical-down - Layouts the model in natural evaluation order. Leaves are at the top of the layout.
- Random - Randomly places the definitions.

Zoom

- Enlarge 10% - Enlarges 10% of the current graph size.
- Reduce 10% - Reduces 10% of the current graph size.
- Normal 100% - Sets back the graph size to original size.
- Fit to window - Zoom the current graph to fit the window.

Help

- Help - Displays this page.
- About - Displays version information and support details.

Appendix I: Definitions for lines in an OXO model

```
allsquares is
[s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18,s19
,s20,s21,s22,s23,s24,s25,s26,s27,s28,s29,s30,s31,s32,s33,s34,s35,s3
6,s37,s38,s39,s40,s41,s42,s43,s44,s45,s46,s47,s48,s49,s50,s51,s52,s
53,s54,s55,s56,s57,s58,s59,s60,s61,s62,s63,s64];
nofsquares is allsquares#;
lin1 is [s1,s2,s3,s4];
lin2 is [s1,s5,s9,s13];
lin3 is [s1,s6,s11,s16];
lin4 is [s2,s6,s10,s14];
lin5 is [s3,s7,s11,s15];
lin6 is [s4,s7,s10,s13];
lin7 is [s4,s8,s12,s16];
lin8 is [s5,s6,s7,s8];
lin9 is [s9,s10,s11,s12];
lin10 is [s13,s14,s15,s16];
lin11 is [s1,s17,s33,s49];
lin12 is [s1,s18,s35,s52];
lin13 is [s2,s18,s34,s50];
lin14 is [s3,s19,s35,s51];
lin15 is [s4,s19,s34,s49];
lin16 is [s4,s20,s36,s52];
lin17 is [s1,s21,s41,s61];
lin18 is [s1,s22,s43,s64];
lin19 is [s2,s22,s42,s62];
lin20 is [s3,s23,s43,s63];
lin21 is [s4,s23,s42,s61];
lin22 is [s4,s24,s44,s64];
lin23 is [s5,s21,s37,s53];
lin24 is [s5,s22,s39,s56];
lin25 is [s6,s22,s38,s54];
lin26 is [s7,s23,s39,s55];
lin27 is [s8,s23,s38,s53];
lin28 is [s8,s24,s40,s56];
lin29 is [s9,s25,s41,s57];
lin30 is [s9,s26,s43,s60];
lin31 is [s10,s26,s42,s58];
lin32 is [s11,s27,s43,s59];
lin33 is [s12,s27,s42,s57];
lin34 is [s12,s28,s44,s60];
lin35 is [s13,s25,s37,s49];
lin36 is [s13,s26,s39,s52];
lin37 is [s14,s26,s38,s50];
lin38 is [s15,s27,s39,s51];
lin39 is [s16,s27,s38,s49];
lin40 is [s16,s28,s40,s52];
lin41 is [s13,s29,s45,s61];
lin42 is [s13,s30,s47,s64];
lin43 is [s14,s30,s46,s62];
lin44 is [s15,s31,s47,s63];
lin45 is [s16,s31,s46,s61];
lin46 is [s16,s32,s48,s64];
lin47 is [s17,s18,s19,s20];
lin48 is [s17,s21,s25,s29];
lin49 is [s17,s22,s27,s32];
lin50 is [s18,s22,s26,s30];
lin51 is [s19,s23,s27,s31];
lin52 is [s20,s23,s26,s29];
lin53 is [s20,s24,s28,s32];
lin54 is [s21,s22,s23,s24];
lin55 is [s25,s26,s27,s28];
lin56 is [s29,s30,s31,s32];
lin57 is [s33,s34,s35,s36];
lin58 is [s33,s37,s41,s45];
```

```
lin59 is [s33,s38,s43,s48];
lin60 is [s34,s38,s42,s46];
lin61 is [s35,s39,s43,s47];
lin62 is [s36,s39,s42,s45];
lin63 is [s36,s40,s44,s48];
lin64 is [s37,s38,s39,s40];
lin65 is [s41,s42,s43,s44];
lin66 is [s45,s46,s47,s48];
lin67 is [s49,s50,s51,s52];
lin68 is [s49,s53,s57,s61];
lin69 is [s49,s54,s59,s64];
lin70 is [s50,s54,s58,s62];
lin71 is [s51,s55,s59,s63];
lin72 is [s52,s55,s58,s61];
lin73 is [s52,s56,s60,s64];
lin74 is [s53,s54,s55,s56];
lin75 is [s57,s58,s59,s60];
lin76 is [s61,s62,s63,s64];
alllines is
[lin1,lin2,lin3,lin4,lin5,lin6,lin7,lin8,lin9,lin10,lin11,lin12,lin
13,lin14,lin15,lin16,lin17,lin18,lin19,lin20,lin21,lin22,lin23,lin2
4,lin25,lin26,lin27,lin28,lin29,lin30,lin31,lin32,lin33,lin34,lin35
,lin36,lin37,lin38,lin39,lin40,lin41,lin42,lin43,lin44,lin45,lin46,
lin47,lin48,lin49,lin50,lin51,lin52,lin53,lin54,lin55,lin56,lin57,l
in58,lin59,lin60,lin61,lin62,lin63,lin64,lin65,lin66,lin67,lin68,li
n69,lin70,lin71,lin72,lin73,lin74,lin75,lin76];
noflines is alllines#;
linesthru1 is [lin1,lin2,lin3,lin11,lin12,lin17,lin18];
linesthru2 is [lin1,lin4,lin13,lin19];
linesthru3 is [lin1,lin5,lin14,lin20];
linesthru4 is [lin1,lin6,lin7,lin15,lin16,lin21,lin22];
linesthru5 is [lin2,lin8,lin23,lin24];
linesthru6 is [lin3,lin4,lin8,lin25];
linesthru7 is [lin5,lin6,lin8,lin26];
linesthru8 is [lin7,lin8,lin27,lin28];
linesthru9 is [lin2,lin9,lin29,lin30];
linesthru10 is [lin4,lin6,lin9,lin31];
linesthru11 is [lin3,lin5,lin9,lin32];
linesthru12 is [lin7,lin9,lin33,lin34];
linesthru13 is [lin2,lin6,lin10,lin35,lin36,lin41,lin42];
linesthru14 is [lin4,lin10,lin37,lin43];
linesthru15 is [lin5,lin10,lin38,lin44];
linesthru16 is [lin3,lin7,lin10,lin39,lin40,lin45,lin46];
linesthru17 is [lin11,lin47,lin48,lin49];
linesthru18 is [lin12,lin13,lin47,lin50];
linesthru19 is [lin14,lin15,lin47,lin51];
linesthru20 is [lin16,lin47,lin52,lin53];
linesthru21 is [lin17,lin23,lin48,lin54];
linesthru22 is [lin18,lin19,lin24,lin25,lin49,lin50,lin54];
linesthru23 is [lin20,lin21,lin26,lin27,lin51,lin52,lin54];
linesthru24 is [lin22,lin28,lin53,lin54];
linesthru25 is [lin29,lin35,lin48,lin55];
linesthru26 is [lin30,lin31,lin36,lin37,lin50,lin52,lin55];
linesthru27 is [lin32,lin33,lin38,lin39,lin49,lin51,lin55];
linesthru28 is [lin34,lin40,lin53,lin55];
linesthru29 is [lin41,lin48,lin52,lin56];
linesthru30 is [lin42,lin43,lin50,lin56];
linesthru31 is [lin44,lin45,lin51,lin56];
linesthru32 is [lin46,lin49,lin53,lin56];
linesthru33 is [lin11,lin57,lin58,lin59];
linesthru34 is [lin13,lin15,lin57,lin60];
linesthru35 is [lin12,lin14,lin57,lin61];
linesthru36 is [lin16,lin57,lin62,lin63];
linesthru37 is [lin23,lin35,lin58,lin64];
linesthru38 is [lin25,lin27,lin37,lin39,lin59,lin60,lin64];
linesthru39 is [lin24,lin26,lin36,lin38,lin61,lin62,lin64];
linesthru40 is [lin28,lin40,lin63,lin64];
linesthru41 is [lin17,lin29,lin58,lin65];
linesthru42 is [lin19,lin21,lin31,lin33,lin60,lin62,lin65];
linesthru43 is [lin18,lin20,lin30,lin32,lin59,lin61,lin65];
linesthru44 is [lin22,lin34,lin63,lin65];
linesthru45 is [lin41,lin58,lin62,lin66];
linesthru46 is [lin43,lin45,lin60,lin66];
linesthru47 is [lin42,lin44,lin61,lin66];
linesthru48 is [lin46,lin59,lin63,lin66];
linesthru49 is [lin11,lin15,lin35,lin39,lin67,lin68,lin69];
linesthru50 is [lin13,lin37,lin67,lin70];
```

```
linesthru51 is [lin14,lin38,lin67,lin71];
linesthru52 is [lin12,lin16,lin36,lin40,lin67,lin72,lin73];
linesthru53 is [lin23,lin27,lin68,lin74];
linesthru54 is [lin25,lin69,lin70,lin74];
linesthru55 is [lin26,lin71,lin72,lin74];
linesthru56 is [lin24,lin28,lin73,lin74];
linesthru57 is [lin29,lin33,lin68,lin75];
linesthru58 is [lin31,lin70,lin72,lin75];
linesthru59 is [lin32,lin69,lin71,lin75];
linesthru60 is [lin30,lin34,lin73,lin75];
linesthru61 is [lin17,lin21,lin41,lin45,lin68,lin72,lin76];
linesthru62 is [lin19,lin43,lin70,lin76];
linesthru63 is [lin20,lin44,lin71,lin76];
linesthru64 is [lin18,lin22,lin42,lin46,lin69,lin73,lin76];
linesthru is
[linesthru1,linesthru2,linesthru3,linesthru4,linesthru5,linesthru6,
linesthru7,linesthru8,linesthru9,linesthru10,linesthru11,linesthru1
2,linesthru13,linesthru14,linesthru15,linesthru16,linesthru17,lines
thru18,linesthru19,linesthru20,linesthru21,linesthru22,linesthru23,
linesthru24,linesthru25,linesthru26,linesthru27,linesthru28,linesth
ru29,linesthru30,linesthru31,linesthru32,linesthru33,linesthru34,li
nesthru35,linesthru36,linesthru37,linesthru38,linesthru39,linesthru
40,linesthru41,linesthru42,linesthru43,linesthru44,linesthru45,line
sthru46,linesthru47,linesthru48,linesthru49,linesthru50,linesthru51
,linesthru52,linesthru53,linesthru54,linesthru55,linesthru56,linest
hru57,linesthru58,linesthru59,linesthru60,linesthru61,linesthru62,l
inesthru63,linesthru64];
```

Appendix J: An LSD account for a train arrival and departure protocols

This is an LSD account for a train arrival and departure protocols developed by Y. P. Yung in the EM research group. The involved agents are a stationmaster, a guard, a driver, a train, passengers and doors.

```
agent sm() {

oracle    (time) Limit, Time,      // knowledge of time to elapse before departure due
     (bool) guard_raised_flag,     // knowledge of whether the guard has raised his flag
     (bool) driver_ready,          // knowledge of whether the driver is ready
     (bool) around[d],             // knowledge of whether there's anybody around doorway
     (bool) door_open[d];          // the open/close status of door d (for d = 1 ..
number_of_doors)

state     (time) tarrive = |time|, // the S-M registers time of arrival
     (bool) can_move,              // the signal observed by driver for starting engine
     (bool) whistle = false,       // the whistle is not blowing
     (bool) whistled = false,      // the whistle has not blown
     (bool) sm_flag = false,       // S-M lowers flag
     (bool) sm_raised_flag = false;// S-M has not raised flag

handle    (bool) can_move,
     (bool) whistle,
     (bool) whistled,
     (bool) sm_flag,
     (bool) sm_raised_flag;
     (bool) door_open[d];          // the open/close status of door d (for d = 1 ..
number_of_doors)

derivate
               number_of_doors
     (bool) ready =     /\     (!door_open[d]);    // are all doors shut?
                 d = 1
     (bool) timeout = (Time - tarrive) > Limit;   // departure due

protocol
     door_open[d] ^ !around[d] -> door_open[d] = false; (d = 1 .. number_of_doors)
     ready ^ timeout ^ !whistled -> whistle = true; whistled = true; guard(); whistle =
false;
     ready ^ whistled ^ !sm_raised_flag -> sm_flag = true; sm_raised_flag = true;
     sm_flag ^ guard_raised_flag -> sm_flag = false;
     ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ !can_move -> can_move = true;

}



agent guard() {

oracle    (bool) whistled, sm_raised_flag, brake;

state     (bool) guard_raised_flag = false,
     (bool) guard_flag = false,
     (bool) brake;

handle    (bool) guard_raised_flag, guard_flag;

derivate  LIVE = engaging || whistled;

protocol
     engaging -> brake = true; running = false;
     sm_raised_flag ^ brake -> brake = false; guard_flag = true; guard_raised_flag = true;
     guard_flag ^ !sm_flag -> guard_flag = false;

}




agent driver() {

oracle    (bool) can_move, engaged, whistled;
     (position) at, from;

handle    (position) to, from,
     (bool) running,
     (bool) driver_ready = false;

state     (bool) driver_ready = false,
     (position) from;

protocol
     whistled ^ !driver_ready -> driver_ready = true;
     engaged ^ from <> at -> from = at; to = next_station_after(at);
     can_move ^ engaged -> driver_ready = false; running = true;

}




agent train() {

state     (bool) running = true,
     (bool) brake = false,
     (bool) door_open[d] = false, (d = 1 .. number_of_doors)
     (position) from = station1,
     (position) to = station2,
     (position) at = some_position,
     (bool) engaging, engaged, leaving, alert;

handle    (bool) alert;

derivate
     (bool) engaging = running ^ to == at,
     (bool) leaving = running ^ from == at,
     (bool) engaged = !running;

protocol
     engaging ^ !alert -> alert = true; guard(); sm();
     leaving ^ alert -> alert = false; delete guard(), sm();

}




agent passenger((int) p, (int) d, (position) _from, (position) _to) {
// passenger p intending to travel from station _from to station _to
// and he will access through door d of the train
oracle    (position) at,
     (bool) door_open[d];

state     (bool) pos[p] = OUT_DOOR, alighting[p], boarding[p], join_queue[p,d];

handle    (position) from[p] = _from;
     (position) to[p] = _to;
     (int) door[p] = d;
     (bool) pos[p],
     (bool) door_open[d];

derivate
     alighting[p] = at == to[p] ^ pos[p] != OUT_DOOR && engaged;
     boarding[p] = at == from[p] ^ pos[p] != IN_DOOR && engaged;
     join_queue[p,d] = (alighting[p] ^ door_open[d] ^ pos[p] == IN_DOOR) ||
          (boarding[p] ^ door_open[d] ^ pos[p] == OUT_DOOR);
     LIVE = !(at == to[p] ^ pos[p] == ON_PLATFORM);
```

```
protocol
     at == to[p] ^ pos[p] == AT_SEAT -> pos[p] = IN_DOOR;
     alighting[p] ^ !door_open[d] -> door_open[d] = true;
     alighting[p] ^ pos[p] == AT_DOOR ^ door_open[d] ^ !queuing[d]
          -> pos[p] == OUT_DOOR; door_open[d] = false; pos[p] = ON_PLATFORM;
     alighting[p] ^ pos[p] == AT_DOOR ^ door_open[d] ^ queuing[d]
          -> pos[p] == OUT_DOOR; pos[p] = ON_PLATFORM;
     boarding[p] ^ !door_open[d] -> door_open[d] = true;
     boarding[p] ^ pos[p] == AT_DOOR ^ door_open[d] ^ !queuing[d]
          -> pos[p] = IN_DOOR; door_open[d] = false; pos[p] = AT_SEAT;
     boarding[p] ^ pos[p] == AT_DOOR ^ door_open[d] ^ queuing[d]
          -> pos[p] = IN_DOOR; pos[p] = AT_SEAT;

}



agent door((int) d) {

oracle    (int) pos[p], door[p]; (p = 1 .. number_of_passengers)

state     (bool) queuing[d], occupied[d], around[d];

derivate
     queuing[d] = there exists p such that join_queue[p,d] == true;
     occupied[d] = there exists p such that (pos[p] == AT_DOOR ^ door[p] == d)
     around[d] = there exists p such that (door[p] == d ^
          (pos[p] == IN_DOOR || pos[p] == AT_DOOR || pos[p] == OUT_DOOR))

protocol  queuing[d] ^ !occupied[d] ^ join_queue[p,d] == true
          -> pos[p] = AT_DOOR; (p = 1 .. number_of_passengers)
}
```

References

[Abo99] Gregory D. Abowd, Software Engineering Issues for Ubiquitous Computing, In proceedings of ICSE’99, April 1999.

[Abo00] Gregory D. Abowd, Elizabeth D. Mynatt, Charting Past, Present and Future Research in Ubiquitous Computing, *ACM Transactions on Computer-Human Interaction*, Special issue on HCI in the new Millenium, 7(1):29-58, March 2000.

[Adz94a] V.D.Adzhiev, W.M.Beynon, A.J.Cartwright, Y.P.Yung, An Agent-oriented Framework for Concurrent Engineering, In *Proc. IEE Colloquium: Issues of Cooperative working in Concurrent Engineering*, Digest 1994/177, 9/1-9/4, October 1994.

[Adz94b] V.D.Adzhiev, W.M.Beynon, A.J.Cartwright and Y.P.Yung, A Computational Model for Multiagent Interaction in Concurrent Engineering, In *Proceedings of CEEDA'94*, Bournemouth Univ., 227-232, 1994.

[Adz94c] V.D.Adzhiev, W.M.Beynon, A.J.Cartwright and Y.P.Yung, A New Computer-Based Tool for Conceptual Design, In *Proceedings of Workshop of Computer Tools for Conceptual Design*, p171-188, University of Lancaster, 1994.

[Adz99] V. Adzhiev, M. Beynon, A. Rykhlinski, *Empirical Modelling of Multi-agent Systems with Inherent Concurrency*, Technical Report, University of Aizu, Japan, 1999.

[Ara95] Agustin A. Araya, Questioning ubiquitous computing, In *Proceedings of the 1995 ACM 23rd annual conference on Computer Science*, ACM Press, 1995.

[Ark99] W. S. Ark, T. Selker (Eds.), *IBM Systems Journal*, Vol38 No. 4, Special issue on Pervasive Computing, IBM Corporation, 1999. Online at: http://www.research.ibm.com/journal/sj/384/tocpdf.html.

[Azu97] Ronald T. Azuma, a Survey of Augmented Reality, In *Presence: Teleoperators and Virtual Environments 6*, p355-385, August 1997.

[Bae95] R. M. Baecker, J. Grudin, W. A. S. Buxton, S. Greenberg(Eds), *Human-Computer Interaction: Towards the Year 2000*, Morgan Jaufmann, 1995.

[Ban92] Liam J. Bannon, Customization and Tailoring of Software Systems Thinking about the Context of Tinkering and Tailoring, *Oksnøen Symposium*, 23-28 May, 1992.

[Bas92] R. Baskerville, J. Travis, D. P. Truex, Systems without Method: the Impact of New Technologies on Information System Development Projects, In *Transactions on the Impact of Computer Supported Technologies in Information System Development*, p241-260, Amsterdam: Elsevier Science, 1992.

[Ber99] N. Bianchi-Berthouze, L. Berthouze, T. Kato, Understanding subjectivity: An interactionist view, In *Proceeding International Conference on User Modeling '99*, p3-13, Canada, 1999.

[Bey85] W. M. Beynon, Definitive Notations for Interaction, In *Proceedings of HCI'85*, p23-34, Cambridge University Press, 1985.

[Bey86a] W. M. Beynon, The LSD Notation for Communicating Systems, CS-RR-087, Department of Computer Science, University of Warwick, 1986.

[Bey86b] W. M. Beynon, D. Angier, T. Bissell, S. Hunt, *DoNaLD: A line-drawing system based on definitive principles*, CS-RR-086, Department of Computer Science, University of Warwick, 1986.

[Bey88a] W. M. Beynon, M. T. Norris, Functional Specification and Description Language, *SDL CCITT Standard Z100*, 1988.

[Bey88b] W. M. Beynon, Definitive principles for interactive graphics, *NATO ASI Series F*, v40, Springer-Verlag, p1083-1097, 1988.

[Bey88c] W. M. Beynon, M. T. Norris, M. D. Slade, Definitions for Modelling and Simulating Concurrent Systems, In *Proceedings of IASTED conference ASM'88*, Acta Press, 1988.

[Bey90a] W. M. Beynon, M. T. Norris, R. A. Orr, M. D. Slade, Definitive Specification of Concurrent Systems, In *Proceedings of UKIT'90*, p52-57, 1990.

[Bey90b] W. M Beynon, M. Slade, S. Yung, *Protocol Specification in Concurrent Systems Software Development*, RR163, Department of Computer Science, University of Warwick, UK, 1990.

[Bey92a] W. M. Beynon, I. Bridge, Y. P. Yung, Agent-Oriented Modelling for a Vehicle Cruise Controller, In *Proceedings of ESDA*, p159-165, ASME PD-v47n4, 1992.

[Bey92b] W. M. Beynon, S. B. Russ, The Interpretation of States: a New Foundation for Computation?, In *Proceedings of PPIG'92,* Loughborough, January 1992.

[Bey94a] W. M. Beynon, Agent-oriented modelling and the explanation of behaviour, In *Proceedings of International Workshop on Shape Modelling Parallelism,* Interactivity and Applications, p54-63, University of Aizu, Japan, 1994.

[Bey94b] W. M. Beynon, A. Cartwright, Y. P. Yung, *Databases from an Agent-oriented Perspective,* CS-RR-278, Department of Computer Science, University of Warwick, UK, 1994.

[Bey97] W. M. Beynon, Empirical Modelling for Educational Technology, In *Proceedings of Cognitive Technology '97*, University of Aizu, Japan, IEEE, p54-68, 1997.

[Bey98] W. M. Beynon, R. I. Cartwright, J. Rungrattanaubol, P-H. Sun, *Interactive Situation Models for Systems Development,* Research Report CS-RR-086, Department of Computer Science, University of Warwick, UK, 1998.

[Bey99] W. M. Beynon, P-H. Sun, Computer-mediated communication: a Distributed Empirical Modelling perspective, In *Proceedings of CT'99*, San Francisco, August 1999.

[Bey00a] W. M. Beynon, S. Maad, Integrated Environments for Virtual Collaboration: an Empirical Modelling Perspective, In *Proceedings of the 5th World Conference On Integrated Design and Process Technology,* 2000.

[Bey00b] W.M.Beynon, A. Ward, S. Maad, A. Wong, S. Rasmequan, and S. Russ, The Temposcope: a Computer Instrument for the Idealist Timetabler, In *Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling, Constance*, Germany, August 16-18, 2000.

[Bey00c] W. M. Beynon, S. Rasmequan, S. Russ, The Use of Interactive Situation Models for the Development of Business Solutions, In *Proceedings workshop on Perspective in Business Informatics Research (BIR-2000),* Rostock, Germany, March 31-April 1, 2000.

[Bey01a] W. M. Beynon, Y. C. Chen, H. W. Hseu, S. Maad, S. Rasmequan, C. Roe, J. Rungrattanaubol, S. Russ, A. Ward, A. Wong, The Computer as Instrument, In *Proceedings of Cognitive Technology 2001: Instruments of Mind*, p476-489, University of Warwick, August 2001.

[Bey01b] W. M. Beynon, C. Roe, A. Ward, A. Wong, Interactive Situation Models for cognitive aspects of user-artefact interaction, In *Proceedings of Cognitive Technology 2001: Instruments of Mind*, University of Warwick, August 2001.

[Bey02a] W. M. Beynon, Concurrent Systems Modelling: Agentification, Artefacts, Animation, In *MSc Lecture Notes T1: Introduction to Empirical Modelling*, Department of Computer Science, University of Warwick, UK, 2002.

[Bey02b] W. M. Beynon, A Perspective on Concurrent Systems, In *MSc Lecture Notes M2: Introduction to Empirical Modelling*, Department of Computer Science, University of Warwick, UK, 2002.

[Blu02] *The Official Bluetooth Website*, Online at http://www.bluetooth.com, June, 2002.

[Boe76] B. W. Boehm, Software Engineering, In *IEEE Transactions on Computers*, v C-25n12, p1226-1241, 1976.

[Boe81] B. W. Boehm, *Software Engineering Economic*s, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Bon02] Kevin Bonsor, *How Augment Reality Will Work*, Howstuffworks, online at http://www.howstuffworks.com/augmented-reality.htm, May 2002.

[Bro95] F. P. Brooks, *The Mythical Man-Month – Essays on software engineering*, 4th Anniversary Edition, Addison-Wesley, 1995.

[Bro95] F. P. Brooks, *The Mythical Man-Month – Essays on software engineering*, 4th Anniversary Edition, Addison-Wesley, 1995.

[Bur01] M. Burnett, J. Atwood, R. W. Djang, H. Gottfried, J. Reichwein, S. Yang, Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, In *Journal of Functional Programming*, v11n2, p155-206, March 2001.

[Bus45] Vannevar Bush, *As We May Think*, Atlantic Monthly, July 1945.

[Buz95] Buzan T., Buzan B., *The Mind Map Book*, BBC Consumer Publishing, 1995.

[Byu01] Hee Eon Byun, Keith Cheverst, Exploiting User Models and Context-Awareness to Support Personal Daily Activities, In *Proceedings of Workshop on User Modelling for Context-aware Applications*, User Modelling 2001 Conference, July 2001.

[Cap96] F. Capra, *The Web of Life: A New Synthesis of Mind and Matter*, London: HarperCollins Publishers, 1996.

[Car95] J. Carter, *The Relational Database*, International Thomson Computer Press, 1995.

[Car99] Richard Cartwright, *Geometric Aspects of Empirical Modelling: Issues in Design and Implementation*, PhD Thesis, Department of Computer Science, University of Warwick, UK, 1999.

[Che93] P. Checkland, *System Thinking, System Practice*, Chichester, England: John Wiley & Sons Ltd, 1993.

[Che01] Keith Cheverst, Nigel Davies, Keith Mitchell, Christos Efstratiou, Using Context as a Crystal Ball: Rewards and Pitfalls, In *Personal and Ubiquitous Computing*, v5, Springer-Verlag London, 2001.

[Che02] Yih-Chang Ch'en, *Empirical Modelling for Participative Business Process Reengineering*, PhD Thesis, Department of Computer Science, University of Warwick, UK, 2002.

[Cod70] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, *Communications of the ACM*, p377-387, June 1970.

[Cor94] D. Corne, R. Ross, L. Fang, Evolutionary Timetabling: Practice, Prospects and Work in Progress, In *Proceedings of the UK Planning and Scheduling SIG Workshop*, Strathclyde, 1994.

[Cro96] M. Crowe, R. Beeby, J. Gammack, *Constructing systems and information: a process view*, McGraw-Hill, 1996.

[Cyp93] A. Cypher (Ed.), *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge MA, 1993.

[Dan99] Daniel Salber, Anind K. Dey, Gregory D. Abowd, The Context Toolkit: Aiding the Development of Context-Enabled Applications, In *The Proceedings of CHI'99*, Pittsburgh, PA, May 15-20, 1999, ACM Press.

[Dau00] K. Dautenhahn, C. L. Nehaniv, Living with Socially Intelligent Agents: A Cognitive Technology View, in K. Dautenhahn (ed.), *Human*

Cognition and Social Agent Technology, John Benjamins Publishing Co., 2000.

[Daw86] R. Dawkins, *The Blind Watchmaker*, Longman, London 1986.

[Dey01a] Anind K. Dey, Understanding and Using Context, In *Personal and Ubiquitous Computing 2001*, Volume 5 Issue 1, Springer-Verlag London, 2001.

[Dey01b] Anind K. Dey, Gregory D. Abowd, A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, In *Human-Computer Interaction*, 16, 2001.

[Dis02] *The Disappearing Computer*, Online at http://www.disappearing-computer.net, July, 2002.

[Dor95] Kees Dorst, Judith Dijkhuis, Comparing paradigms for describing design activity, In *Design Studies*, v16, p261-274, 1995.

[Edm95] E. Edmonds, G. Fischer, S. J. Mountford, F. Nake, D. Riecken, R. Spence, Creativity Interacting with Computers, In *CHI'95 Proceedings*, 1995.

[Edw01] W. Keith Edwards, At Home with Ubiquitous Computing: Seven Challenges, In G. D. Abowd, B. Brumitt, S. A. N. Shafer (Eds): *Ubicomp 2001*, LNCS 2201, p256-272, 2001.

[Elr01] T. Elrad(moderator), M. Aksit, G. K. Lierberherr, Ossher, Panelists, Discussing Aspects of AOP, *Communications of The ACM*, v44n10, October, 2001.

[EMWeb] *The Homepage of Empirical Modelling*, Online at http://www.dcs.warwick.ac.uk/modelling.

[Eva01] Michael Evans, W. M. Beynon, Carlos N. Fischer, Empirical Modelling for the Logistic of Rework in the Manufacturing Process, In *Proceedings of 16th Brazilian Congress of Mechanical Engineering,* p226-234, November 2001.

[Eva92] Malcolm Eva, *SSADM Version 4: A User's Guide*, London: McGraw-Hill, 1992.

[Fer01] Kiran Fernandes, Vinesh Raja, John Keast, W.M.Beynon, Pui Shan Chan and Mike Joy, Business and IT Perspectives on AMORE: a Methodology for Object-Orientation in Re-engineering Enterprises, In

Systems Engineering for Business Process Change: New Directions, Springer-Verlag, December 2001.

[Fey64] R. Feynman, R. Leighton, M. Sands, *The Feynman Lectures on Physics: Volume II,* Addison Wesley, 1964.

[Fis00] Gerhard Fischer, User Modeling in Human-Computer Interaction, in *10th Anniversary Issue of "User Modeling and User-Adapted Interaction(UMUAI),* 2000.

[Fis01] Carlos N Fischer, W.M.Beynon, Empirical Modelling of Products, In *Proceedings of International Conference on Simulation and Multimedia in Engineering Education*, Phoenix, Arizona, January 2001, p20-26, The Society for Modelling and Simulation International, 2001.

[Fit94] B. Fitzgerald, The Systems Development Dilemma: Whether to Adopt Fomalised Systems Development Methodologies or Not?, In *Proceedings of Second European Conference on Information Systems*, p691-706, 1994.

[Fol02] *Free On-Line Dictionary of Computing*, Imperial College Department of Computing, Online at http://foldoc.doc.ic.ac.uk/foldoc/, January 2002.

[Fre02] David H. Freedman, *Are Holograms Finally for Real?*, Business 2.0, July 2002.

[Gal94] Stratis Gallopoulos, Elias Houstis, John Rice, Computer as Thinker/Doer: Problem-Solving Environments for Computational Science, In *IEEE Computational Science and Engineering*, 1994.

[Gar79] M. R. Garey, D. S. Johnson, *Computers and Intractability – A guide to NP-completeness*, W. H. Freeman and Company, 1979.

[Gar99] Gardner, *The OXO model*, Reference Code: oxoGardner1999, Model Online at EM electronic data repository: http://empublic.dcs.warwick.ac.uk/projects, 1999.

[Geh96] Dominic Gehring, Spreadsheets and programming, *PPIG Postgraduate Student Workshop*, 1996.

[Geh98] Dominic Gehring, *The Modd API*, Online at: http://www.dcs.warick.ac.uk/~gehring/modd, 1998.

[Gib00] W. Wayt Gibbs, As We May Live, *Scientific American*, November 200.

[Gog94] J. A. Goguen, Requirements Engineering as the Reconciliation of Technical and Social Issues, In *Requirements Engineering: Social and Technical Issues*, edited with Marina Jirotka, Academic Press, pages 165-199, 1994

[Gog96] J. A. Goguen, Formality and Informality in Requirements Engineering, In *Proceedings of Fourth International Conference on Requirements Engineering*, IEEE Computer Society, pages 102-108, April 1996.

[Goo90] D. Gooding, *Experiment and the Making of Meaning: Human Agency in Scientific Observation and Experiment*, Kluwer Academic Publishers, 1990.

[Goo01] D. Gooding, Experiment as an Instrument of Innovation: Experience and Embodied Thought, In *Proceedings of Cognitive Technology 2001: Instruments of Mind*, p130-140, 2001.

[Gor97] B. Gorayska, J. P. Marsh, J. L. Mey, Putting the horse before the cart: Formulating and exploring methods for studying Cognitive Technology. In J. P. Marsh, C. L. Nehaniv, B. Gorayska (eds.), *Proc. Second International Conference on Cognitive Technology (CT'97)*, IEEE Computer Society Press, 1997.

[Gre98a] T. Green, A. Blackwell, *Cognitive Dimensions of Information Artefacts: a tutorial*, http://www.ndirect.co.uk/~thomas.green/workstuff/Papers/, September 1998. (Originally prepared for the BCS HCI Conference of 1998.)

[Gre98b] Darryl Green, Ann DiCaterino, *A Survey of System Development Process Models*, Center for Technology in Government, University of Albany, USA, 1998.

[Gre99] Thomas Green, *The BALMORAL Central Heating Controls*, Online at http://www.ndirect.co.uk/~thomas.green/workStuff/devices/controllers/HeatingB.html, December 1999.

[Gri98] Rebecca E. Grinter, Recomposition: Putting It All Back Together Again, In *Proceedings of CSCW 98*, ACM, Seattle Washington, USA, 1998.

[Hao85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewoord Cliffs, 1985.

[Har88] David Harel, On Visual Formalisms, In *Communication of ACM*, volume 31, number 5, May 1988.

[Har95] William Harrison, Harold Ossher, Subject-Oriented Programming (A Critique of Pure Objects), *OOPSLA'93*, p411-428, 1993.

[Hew95] Thomas T. Hewett, Cognitive Factors in Design: Basic Phenomena in Human Memory and Problem Solving, In *CHI'95 Proceedings*, 1995.

[Hje01] Sara Ilstedt Hjelm, Designing the invisible computer – from radio-clock to screenfridge, *Digital Visions and User Reality*, Copenhagen, 31 Oct – 3 Nov, 2001.

[Hoc90] J.-M. Hoc, A. Nguyen-Xuan, Language Semantics, Mental Models and Analogy, In *Computers and People Series Psychology of Programming*, p139-156, 1990.

[Hol99] L. E. Holmquist, J. Redström, P. Ljungstrand, Token-Based Access to Digital Information, In Gellersen, H. W. (Ed.), *Handheld and Ubiquitous Computing*, Lecture Notes in Computer Science No. 1707, pp. 234 - 245. Springer-Verlag, 1999.

[Hon01] Jason I. Hong, James A. Landay, An Infrastructure Approach to Context-Aware Computing, In *Human-Computer Interaction*, 16, 2001.

[Hop99] Andy Hopper, *Sentient Computing*, The Royal Society Clifford Paterson Lecture, University of Cambridge and AT&T Lab Cambridge, 1999.

[Hua99] Andrew C. Huang, Benjamin C. Ling, Shankar Ponnekanti, Armando Fox, Pervasive Computing: What Is It Good For?, In *Proceedings of Workshop on Mobile Data Management (MobiDE) in conjunction with ACM MobiCom '99*, Seattle, WA, September 1999.

[Hud94] Scott E. Hudson, User Interface Specification Using an Enhanced Spreadsheet Model, In *ACM Transactions on Graphics*, v13n3, p209-239, July 1994.

[Ilo02] Software: Rhapsody Version 4, http://www.ilogix.com/products/rhapsody/, I-Logix Inc., 2002.

[Jac02] B. Jacobs, *Object Oriented Programming Oversold!*, online at http://www.geocities.com/SiliconValley/Lab/6888/oopbad.htm, May, 2002.

[Jac92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, *Object-oriented Software Engineering: A User Case Driven Approach*, Addison-Wesley, 1992.

[Jam96] W. James, *Essays in Radical Empiricism*, Bison Books, 1996.

[Jen00] N.R. Jennings, On agent-based software engineering, *Artificial Intelligence*, 117(2000) 277-296.

[Joh99] Brian Johnson, Ben Shneiderman, Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structure, In S. K. Card, J. D. Mackinlay, B. Shneiderman (Eds.), *Readings in Information Visualization: Using Vision to Think*, 1999.

[Jor68] N. Jordan, Some Thinking about Systems, Chapter 5 in *Themes in Speculative Psychology*, London: Tavistock Publishing, 1968.

[Jos02] Mini K. Joseph, Now, it's autonomic computing, *Times News Network*, July 2002.

[Kai99] H. Kaindl, Difficulties in the Transition from OO analysis to Design, in *IEEE software*, September/October 1999, pp.94-102.

[Kei92] Reinhard Keil-Slawik, Customizing Artifacts an Ecological Perspective, *Oksnøen Symposium*, 23-28 May, 1992.

[Kim97a] J. Kim, F. J. Lerch, Why is Programming (Sometimes) So Difficult?: Programming as Scientific Discovery in Multiple Problem Spaces, In *Information Systems Research*, 8 (1), 25-50, 1997.

[Kim97b] J. Kim, J. Hahn, F. J. Lerch, How is the Designer Different from the User? Focusing on a Software Development Methodology, In *Proceedings of the 7th Empirical Studies of Programmers*, p69-90, 1997.

[Kim99] Jinwoo Kim, Jungpil Hahn, Hyoungmee Hahn, *How Do We Understand a System with (so) Many Diagrams? Cognitive Integration Processes in Diagrammatic Reasoning*, Department of Business Administration, Yonsei University, Korea, 1999.

[Kla88] D. Klahr, K. Dunbar, Dual space search during scientific reasoning, In *Cognitive Science*, 12, p1-55, 1988.

[Kul00] Bill Kules, *User Modelling for Adaptive and Adaptable Software Systems*, Department of Computer Science, University of Maryland, April 2000.

[Lae01] Kristof Van Laerhoven, Kofi Aidoo, Teaching Context to Applications, In *Personal and Ubiquitous Computing 2001*, Volume 5 Issue 1, Springer-Verlag London, 2001.

[Lan02] M. Langheinrich, V. Coroama, J. Bohn, M. Rohs, *As we may live – Real-world implications of ubiquitous computing*, Submitted for publication, Online at , www.inf.ethz.ch/vs/publ/papers/implications-uc2002.pdf, April 2002.

[Lid94] S. W. Liddle, D. W. Embley, S. N. Woodfield, A Seamless Model for Object-oriented Systems Development, In *Proceedings of International Symposium of OO Methodologies and Systems*, p123-141, 1994.

[Lin00] J. Lind, *Issues in Agent-Oriented Software Engineering*, The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000.

[Log95] Brian Logan, Janet McDonnell, *Design Dialogues: One*, Design Studies, May 1995.

[Loo98] M. Loomes and S. Jones, Requirements Engineering: A Perspective through Theory-Building, in *Proc. Third International IEEE Conference on Requirements Engineering*, IEEE Computer Society Press, 1998.

[Loo01] M. Loomes and C. L. Nehaniv, Fact and Artifact: Reification and Drift in the History and Growth of Interactive Software Systems, in *Cognitive Technology: Instruments of Mind 4th International Conference*, CT 2001, Warwick, UK, Springer LNAI vol 2117, pp 25-39. August 6-9, 2001.

[Luc99] Robert W. Lucky, Connections, In Reflections column of *IEEE Sprectrum*, March 1999.

[Maa02] Soha Maad, *An Empirical Modelling Approach to Software System Development in Finance: Applications and Prospects*, PhD Thesis, Department of Computer Science, University of Warwick, UK, March 2002.

[Mar98a] R. T. Marshak, Open Text Livelink Intranet: Expanding Document Management to Support Collaborative Team Projects, Patricia Seybold Group's Workgroup Computing Report Vol.21, No.7, 1998.

[Mar98b] Michael E. Martinez, What Is Problem Solving?, In *Phi Delta Kappan*, p605-609, April 1998.

[Mcc79] John McCarthy, Ascribing mental qualities to machines, In *Philosophy Aspects in Artificial Intelligence*, Harvester Press, 1979.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[Min00] Software: *MindManager*, Mindjet LLC, 2000.

[Min88] M. Minsky, *The Society of Mind*, Picador London, 1988.

[Mye98] Brad A. Myers, *Natural Programming Project Overview and Proposal*, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, January 1998.

[Nap97] Lisa Napoli, Wearable Computers: The User Interface Is You, *CyberTimes*, October 1997.

[Nar93] B. A. Nardi, *A small matter of programming – perspectives on end user computing*, MIT Press, Cambridge, Massachusetts, London, England, 1993.

[Nau01] Peter Naur, *Antiphilosophical Dictionary – Thinking Speech Science/Scholarship*, Naur.com Publishing, 2001.

[Neg96] Nicholas Negroponte, *Being Digital*, Coronet London, 1996.

[New72] A. Newell, H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Nor99] Donald A. Norman, The invisible computer: why good products can fail, the personal computer is so complex, and information appliances are the solution, MIT Press, 1999.

[Oco00] Patrick D. T. O'Connor, Commentary: Reliability – Past, Present, and Future, In *IEEE Transactions on Reliability*, v49n4, December 2000.

[Odl99] Andrew Odlyzko, The Visible Problems of the Invisible Computer: A Skeptical Look at Information Appliances, First Monday, 1999.

[Oed02] *Oxford English Dictionary Online*, Online at http://dictionary.oed.com, August 2002.

[Ope02] TkEden open-source distribution, Online at: http://www.sourceforge.net/projects/eden/, 2002.

[Opt00] Optime, ASAP, Online at http://www.asap.cs.nott.ac.uk/optime/, 2000.

[Pae94] B. Paechter, H. Luchian, A. Cumming and M. Petruic, Two solutions to the general timetable problem using evolutionary methods, In *Proceedings of IEEE Conference on Evolutionary Computation*, 1994.

[Pai01] Richard Paige, Jonathan Ostroff, *Producing Reliable Software via the Single Model Principle*, Technical Report CS-2002-02, Department of Computer Science, York University, Canada, May 2001.

[Pan01] J. F. Pane, C. A. Ratanamahatana, B. A. Myers, Studying the Language and Structure in Non-programmers' Solutions to Programming Problems, In *International Journal of Human-Computer Studies*, v54n2, p237-264, February 2001.

[Ped97] Thomas Pederson, *A Cup of Tea & a Piece of Cake – Integration of Virtual Information Workspaces Inspired by the Way We May Think*, MSc thesis, Ericsson Media Lab, Umeå University report UMNAD 182.97, 1997.

[Pól45] G. Pólya, *How to solve it*, Princeton University Press, USA,1945.

[Pur00] Purchase H. C., Effective information visualisation: a study of graph drawing aesthetics and algorithms, In *Interacting with Computers*, 13, p147-162, 2000.

[Rah01] O. W. Rahlff, R. K. Rolfsen, J. Herstad, Using Personal Traces in Context Space: Towards Context Trace Technology, In *Personal and Ubiquitous Computing 2001*, Volume 5 Issue 1, Springer-Verlag London, 2001.

[Ram94] K. Ramamohanarao, J. Harland, *An Introduction to Deductive Database Languages and Systems*, VLDB Journal, 3, 107-122, 1994.

[Ras01] Suwanna Rasmequan, *An Approach to Computer-based Knowledge Representation for the Business Environment using Empirical Modelling*, PhD Thesis, Department of Computer Science, University of Warwick, UK, November 2001.

[Rat02] Software: Rose, http://www.rational.com/products/rose/, Rational Software Corp., 2002.

[Rob00] Jason Robbins, David Redmiles, *Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML*, Information and Software Technology, 2000.

[Rob92] Mike Robinson, "It's not a bloody typewrite you know" – Problems of Tailoring and Personalisation for Unanticipated User, Organisational Coherence and CSCW, *Oksnøen Symposium*, 23-28 May, 1992.

[Rob96] Jason E. Robbins, David F. Redmiles, Software Architecture Design from the Perspective of Human Cognitive Needs, In *Proceedings of the California Software Symposium 1996*, p16-27, 1996.

[Roe01] Chris Roe, *EM model: A digital watch model with activity diagram*, Reference Code: digitalWatchRoe2001, Model Online at EM electronic data repository: http://empublic.dcs.warwick.ac.uk/projects, 2001.

[Roe02] C. Roe, W. M. Beynon, Empirical Modelling principles to support learning in a cultural context, *1st International Conference on Educational Technology in Cultural Context*, University of Joensuu, Finland, 2002.

[Run02] Jaratsri Rungrattanaubol, *A Treatise on Modelling with Definitive Scripts*, PhD Thesis, Department of Computer Science, University of Warwick, April 2002.

[Rus95] Nancy L. Russo, *The Use and Adaptation of System Development Methodologies*, International Resources Management Association International Conference, Atlanta, Georgia, May 1995.

[Sch83] D. A. Schön, *The Reflective Practitioner*, Harper Collins, USA, 1983.

[Sch95] C. D. Schunn, D. Klahr, A 4-space model of scientific discovery. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, p106-111, 1995.

[Sch00a] Jennifer L. Schenker, Not Very PC, In *Time Europe*, vol.155 no.8, Feb 28, 2000.

[Sch00b] Schedule Expert, Online at http://scheduleexpert.com, 2000.

[Sen02] *Sentient Computing Project Home Page*, AT&T Laboratories, Cambridge, Online at http://www.uk.reserach.att.com/spirit/, June 2002.

[Sim68] H. A. Simon, Sciences of Artificial, Cambridge: MIT Press, 1968.

[Sim74] H. A. Simon, G. Lea, Problem solving and rule induction: A unified view, In *Knowledge and Cognition*, 1974.

[Sla90] M. Slade, *Definitive Parallel Programming*, MSc Thesis, Department of Computer Science, University of Warwick, 1990.

[Smo01] K. Smolander, K. Hoikka, J. Isokallio, M. Kataikko, T. Mäkelä and H. Kälviäinen, *Required and Optional Viewpoints: What is Included in Software Architecture,* Lappeenranta University of Technology, Telecom Business Research Center, 2001.

[Str01] Norbert A. Streitz, Mental vs. Physical Disappearance: The Chanllenge of Interacting with Disappearing Computers, In *Workshop Proceedings of Distributed and Disappearing UIs in Ubiquitous Computing*, CHI, 2001.

[Suc87] L. A. Suchman, *Plans and Situated Actions: the Problem of Human-Machine Communication,* Learning in doing: Social, cognitive and computation perspectives, CUP 1987.

[Sug81] Sugiyama K., Tagawa S., Toda M., Methods for Visual Understanding of Hierarchical System Structures, In *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. Smc-11, No.2, Feb 1981.

[Sun98] Pi-Hwa Sun, Richard Cartwright, A Quick Guide to dkeden, Department of Computer Science, University of Warwick, UK, 1998.

[Sun99a] Pi-Hwa Sun, *Distributed Empirical Modelling and its Application to Software System Development*, PhD Thesis, Department of Computer Science, University of Warwick, UK, July 1999.

[Sun99b] Pi-Hwa Sun, *The Clayton Tunnel Railway Accident Simulation,* Reference Code: claytontunnelSun1999, Model Online at EM electronic data repository: http://empublic.dcs.warwick.ac.uk/projects, 1999.

[Tan01] Peter Tandler, Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices, In *Proceedings of Ubicomp 2001: Ubiquitous Computing*, Heidelberg: Springer LNCS 2201, p96-115, 2001.

[Ten00] David Tennenhouse, Proactive Computing, *Communications of the ACM*, v43n5, May 2000.

[Tod76] S. Todd, The Peterlee Relational Test Vechicle – a System Overview, *IBM Systems Journal,* p285-308, v15n4, 1976.

[Tog02] Software: ControlCenter, http://www.togethersoft.com/products/controlcenter/, TogetherSoft Corp., 2002.

[Tol96] Tollis, I. G., 1996. Graph Drawing and Information Visualization, Strategic Directions in Computing Research: Working Group on Computational Geometry, *ACM Computing Surveys*, 28a(4), December 1996.

[Tru00] Duane Truex, Richard Baskerville, Julie Travis, Amethodical system development: the deferred meaning of systems development methods, In *Accounting, Management and Information Technologies*, p53-79, Elsevier, 2000.

[Tuk96] M. Tukiainen, Basset: a structured spreadsheet calculation system, In *Machine-Mediated Learning*, v5n2, p63-76, 1996.

[Tur96] M. Turner, *The Literary Mind*, Oxford University Press, 1996.

[Tur00] Neil Turner, *EM model: Distributed five a side football*, Reference code: footballTurner200, Online at EM electronic data repository: http://empublic.dcs.warwick.ac.uk/projects, May 2000.

[Uml02] *Introduction to OMG's Unified Modeling Language (UML)*, Online at http://www.omg.org/gettingstarted/what_is_uml.htm, February 2002.

[Weg97] P. Wegner, Why interaction is more powerful than algorithms, In Communications of the ACM, v40n5, May 1997.

[Wei91] Mark Weiser, The Computer for the Twenty-First Century, *Scientific American*, pp. 94-10, September 1991.

[Wei93] Mark Weiser, Some Computer Science Issues in Ubiquitous Computing, *CACM*, July 1993.

[Wes97] Dave West, Hermeneutic Computer Science, *Communications of the ACM*, v40n4, April 1997.

[Win01] Terry Winograd, Architectures for Context, In *Human-Computer Interaction*, 16, 2001.

[Won98] Allan K. T. Wong, *Implementing a Definitive Notation for Windowing and Graphics using Modd*, Undergraduate Final Year Project Report, Department of Computer Science, University of Warwick, 1998.

[Woo95] M. Wooldridge, N. R. Jennings, Intelligent agents: theory and practice, In *The Knowledge Engineering Review*, v10n2, p115-152, 1995.

[Yun88] Y. P. Yung, Y. W. Yung, *The EDEN handbook*, Department of Computer Science, University of Warwick, 1988 (updated in 1996).

[Yun89] Yung Edward Yun Wai, EDEN: *An Engine for Definitive Notation - Design, Implementation and Evaluation*, MSc Thesis, Department of Computer Science, University of Warwick, UK, September 1989.

[Yun90] Edward Yung, *EDEN: An Engine for Definitive Notations*, MSc Thesis, Department of Computer Science, University of Warwick, September 1990.

[Yun93] Simon Yung, *Definitive Programming – a Paradigm for Exploratory Programming*, PhD Thesis, Department of Computer Science, University of Warwick, January 1993.

Index