THE UNIVERSITY OF WARWICK

Distributed Empirical Modelling and its Application to Software System Development

by

Pi-Hwa Sun

Thesis

Submitted to The University of Warwick

in partial fulfillment of the requirements

for admission to the degree of Doctor of Philosophy

Department of Computer Science

University of Warwick

July 1999

Contents

List of Tables

List of Figures

Acknowledgments

This thesis would not have been written without the encouragement and guidance of my supervisor, Meurig Beynon. Thank you Meurig for picking me up out of the depths of despair, for enlightening me the profound knowledge of Empirical Modelling, and for giving me invaluable help throughout the research and writing of this thesis.

I would also like to thank all the members of the Empirical Modelling Group at Warwick University for providing a stimulating research environment. To Steve Russ I owe extra special thanks for many discussions and comments during my research and on thesis drafts. To Suwanna Rasmequan and Ashley Ward I owe thanks for their friendly help in various aspects of my work and proof reading.

Finally, my deepest gratitude goes to my parents for their love and support throughout my work, and my wife, Lisa, who has cared for our daughters, Sophy and Mimi. Without their assistance and support, I could not possibly go through these most challenging years of my life.

* * * * * * * *

Further thanks goes to Diane Sonnenwald for her constructive comments during the viva and to Meurig Beynon and Steve Russ for their useful help on this final version.

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.

The perspective on distributed Empirical Modelling expressed in this thesis has been published in [BS99]. The various aspects concerning the application of distributed Empirical Modelling to requirements engineering has been represented in [SB98, SCRB99]. The view of Interactive Situation Model relating to software system development has been proposed in [BS98, BCSW99]. The example of the railway accident in the Clayton Tunnel has previously appeared in [SB98, BS99].

Abstract

Empirical Modelling (EM) is a new approach for software system development (SSD) that is particularly suitable for ill-defined, open systems. By regarding a software system as a computer model, EM aims to acquire and construct the knowledge associated with the intended system by situated modelling in which the modeller interacts with the computer model through continuous observations and experiments in an open-ended manner. In this way, a software system can be constructed that takes account of its context and is adaptable to the rapidly changing environment in which the system is developed and used.

This thesis develops principles and tools for *distributed* Empirical Modelling (DEM). It proposes a framework for DEM by drawing on two crucial theories in social science: *distributed cognition* and *ethnomethodology*. This framework integrates cognitive and social processes, allowing multiple modellers to work collaboratively to explore, expand, experience and communicate their knowledge through interaction with their networked computer models. The concept of *pretend play* is proposed, whereby modellers as internal observers can interact with each other by acting in the role of agents within the intended system in order to shape the agency of such agents.

The author has developed a tool called dtkeden to support the proposed DEM framework. Technical issues arising from the implementation dtkeden and case-studies in its use are discussed. The popular star-type logical configuration network and the client/server communication technique are exploited to construct the network environment of this tool. A protocol has been devised and embedded into their communication mechanism to achieve synchronisation of computer models. Four interaction modes have been implemented into dtkeden to provide modellers with different forms of interpersonal interaction. In addition, using a virtual agent concept that was initially devised to allow definitions of different contexts to co-exist in a computer model, a definitive script can be interpreted as a generic observable that can serve as a reusable definitive pattern. Like experience in everyday life, this definitive pattern can be reused by particularising and adapting it to a specific context. A comparison between generic observables and abstract data types for reuse is given.

The application of the framework for DEM to requirements engineering is proposed. The requirements engineering process (REP) – currently poorly understood – is reviewed. To integrate requirements engineering with SSD, this thesis suggests reengineering the REP by taking the context into account. On the basis of DEM, a framework (called SPORE) for the REP is established to guide the process of cultivating requirements in a situated manner. Examples of the use of this framework are presented, and comparisons with other approaches to RE are made.

Abbreviations

AI – Artificial Intelligence

DEM – distributed Empirical Modelling

EM – Empirical Modelling

ODM – Open Development Model

RE – Requirements Engineering

REP – Requirements Engineering Process

SPORE – Situated Process of Requirements Engineering

SSD – Software System Development

Chapter 1
Introduction

1.1 Research Motivation and Aims

This research is motivated by an interest in seeking an 'amethodical' approach[1] to software system development[2] (SSD). This interest stems from the author's past experience over nine years of developing Management Information Systems[3] (MISs). A MIS typically involves people from different backgrounds, with different knowledge and experience, and working in various organisational roles, and also includes their complex business processes that could be ill-structured and experience-oriented. In most cases, the developers[4] are totally unfamiliar with such working practices, and the users are not sure what they want, or even what they could have, from the system being developed. The

[1] The term 'amethodical' is used in this thesis to mean that no particular, strict method is used, rather than suggesting that no method is used. The term 'method' is used here as a synonym for 'methodology' and refers to a series of well-structured steps and procedures to be followed in the course of developing a software system [AF95].

[2] In this thesis, the term 'software system development' is preferred to the popular term 'information system development' in order to stress those information systems which are software-intensive. It refers to a process in which human agents construct and use a software system for their practices (cf. [Flo87, HKL95]). A broader perspective on SSD that cannot be completely addressed using conventional concepts of SSD is considered in this thesis. More details are elaborated in Chapter 3 (see especially p.49-50).

[3] A MIS is traditionally defined as "an integrated, computer-based, user-machine system that provides information for supporting operations and decision-making functions' [Awa88, p.5]. A modern MIS is a highly interactive information system that generates information for monitoring performance, maintaining coordination, and understanding problems and new situations [Alt96, p.223]

[4] In this thesis, the term 'developer' is applied to all people who are engaged in software system development, such as analysts, designers and programmers.

requirements for such a system are invariably unclear and volatile, in particular when the factor of reengineering the business process of users is taken into account. Accordingly, the intended system is ill-defined and open.

There are many textbook approaches for developing such a system. In the author's experience, however, most method-based approaches devised to date, from traditional waterfall methods [SS95, STM95] to the popular object-oriented methods [Boo94, CY90, Jac92], are difficult to use[5]. On the one hand, it is clear that the use of a method-based approach providing formally-defined methods, techniques and tools can in principle guide the development of software systems in a systematic and cost-effective fashion. On the other hand, such a technical process, which regards SSD as a manufacturing process within the engineering discipline and as therefore deterministic, mechanistic and rational, cannot easily adapt to a continuously changing environment [Fit96, Gog94]. This is because, in the case of an open, ill-defined system, it is very hard for developers to cope with diverse situations simply by following a set of steps and procedures imposed by a method. Instead, developers usually perform this task by sequences of what L. Suchman has called 'situated[6] actions' [Suc87], which are interactive actions undertaken by the actor in response to the situation of his/her external environment.

For example, object-oriented methods proposed in [Boo94, CY90, Jac92] require conservative assumptions about what is naturally an object. For instance, objects have a certain fixedness of roles and persistent inheritance of methods. This requires that each object must be subject to and must conform to well-defined and explicitly stated rules. In other words, the problem domain, objects and the interactions between objects become

[5] In this respect, the author's experience is in accord with the concerns of the research in [Fit96, RHHR98, And+90], where the practical difficulties of using a method-based approach for SSD are identified. It is also acknowledged that what developers do in the real world is often quite different from what method-based approaches suggest that they should do, but the discussion in this thesis does not take this into account.

[6] In [Gog96], J.A. Goguen identifies the qualities of situatedness as 'emergent, local, contingent, embodied, open and vague'. This description helps to clarify the meaning of the term 'situated'.

relatively fixed or static once they are assumed in a model. Therefore, within this approach, the real world, which is often chaotic and complex, is either ignored or expressed in a diagrammatic form, which is bound to be tangible constrained and formal. Obviously, such a mechanical approach cannot deal with contingent problems in a situated manner.

In addition, in the author's experience, the practice of interacting with users through documentation is not as effective as these traditional step-by-step methods claim. In reality, as indicated in [Fis91, DS97], interpersonal interaction through documentation is passive, error-prone and labour-intensive and has been recognised as the most difficult part of the process of SSD [Bub95, Eas93, Pot93, STM95, VPC98, Zav95]. This is because these traditional methods take it for granted that developers can collect and represent users' needs through text- and diagram-based metaphors (or even a prototype of the intended system), and that users can clearly express their needs and understand what these metaphors mean. Ironically, in such a framework, users are 'outsiders' in relation to the system being developed for them, and developers are at the centre of SSD. Figure 1-1(a) shows the relationship between developers, users, the used method, the used metaphors and the developing system in this developer-centred development[7] of software systems. Despite further improvements that have been proposed, such as user-centred design and Joint Application Design (JAD) [AF95], users are still outsiders, though they do have more opportunities to contribute to design decisions (see Figure 1-1(b)). In the context of both Figure 1-1(a) and Figure 1-1(b), developers still dictate the agenda of activities for ensuring the transition from users' needs to a software system.

[7] Strictly speaking, this developer-centred development should be regarded as method-centred, since the behaviour of each developer is restricted to following the steps dictated by the used method.

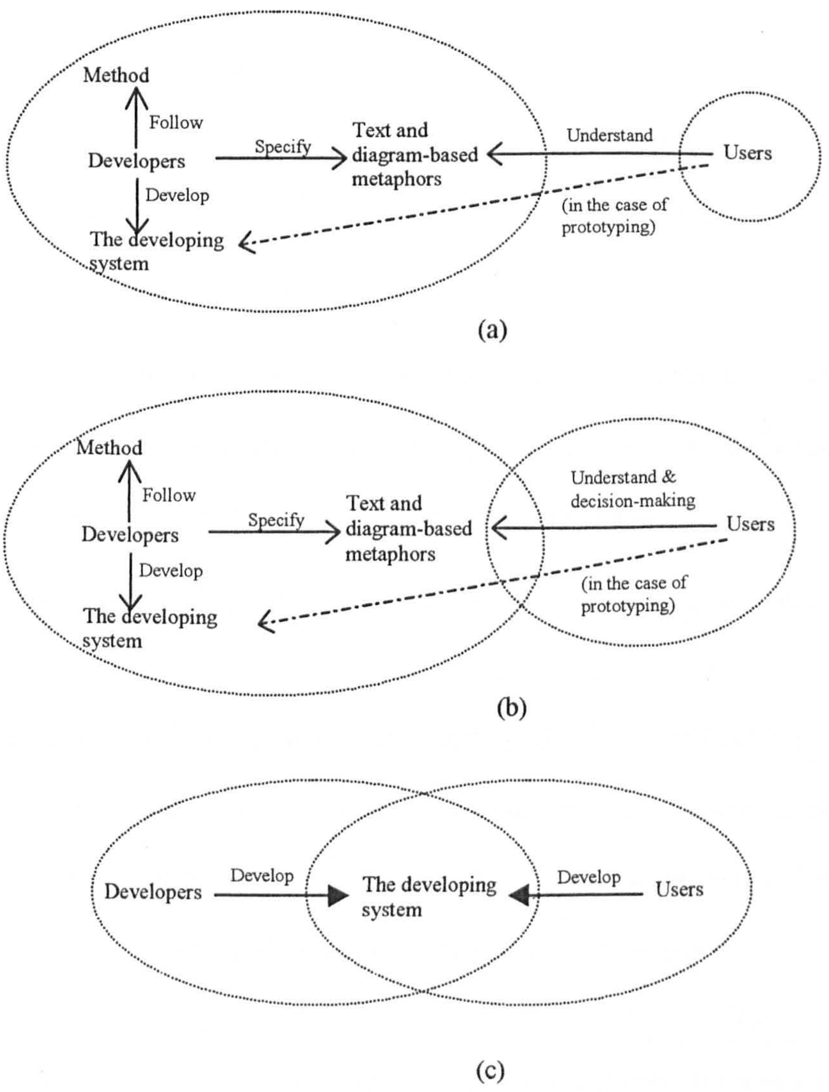


Figure 1-1.
(a) developer-centred SSD with users involved for consulting.
(b) developer-centred SSD with users involved for decision-making
(c) SSD with users involved for co-development

On the basis of this practical experience, the author has been led to ask two fundamental questions about the use of method in SSD:

- To what extent does a method really guide SSD and solve problems arising from the process of SSD?
- Is it possible to devise an amethodical approach[8] to SSD (that is, an approach that does not follow rigid activity patterns), that gives effective support to situated interpersonal interaction?

The first question is highly contentious, but its clear implication is that no method specifies *the* best way to develop software systems. It is beyond the scope of the present research to enter into a detailed examination of this question.

The answer to the second question is presently emerging. On the basis of an historic review of SSD methods, R. Hirschheim et al. point out that SSD is gradually being transformed from a technical process to a social process [HKL95]. This is because it has become increasingly evident that a technical process is unable to cope with social issues, such as job satisfaction, user resistance, learning and interpersonal interaction [CS90, HKL95, Mum95]. Hence, it now widely recognised that SSD is a human activity in which people attempt to make sense of their own and others' actions through interaction [HKN91, Flo95]. Accordingly, several approaches regarding SSD as a social process have been proposed, including SSM (Soft System Method) [CS90], ETHICS (Effective Technical and Human Implementation of Computer-based System) [Mum95] and MULTIVIEW [AF95]. Although they are based on different philosophies, principles and concepts, all these approaches put the emphasis on facilitating interaction between

[8] An amethodical approach should not be confused with a so-called contingent (or eclectic) approach. In a contingent approach, several methods, considered to be complementary to each other, are blended together from the diverse aspects of SSD in order to help developers deal with the various characteristics of the project or domain [AF95, McD90]. Such an approach is usually characterised by a hybrid model in which the behaviour of developers in each step is dictated by one particular method. It also puts the developers at the centre of the development of software systems and involves user participation to a limited extent only.

developers and users who participate in the process of SSD. Their justification for this is that the success of a software system is proportional to the degree to which its users are actually involved in the development of that system. They also point out that the barriers to interpersonal interaction for SSD can be removed by supporting learning through the mutual exploration of knowledge and understanding. These new approaches conclude that there is an alternative to the development of software systems by following fixed activity patterns, and that this alternative is anchored in the social process perspective. However, these new approaches are not the author's concern in the present thesis, since they are still restricted to rigid algorithms.

Two other approaches have greater relevance to the author's interest in seeking an amethodical approach: the professional work practices approach (PWP) [And+90] and cooperative design [Kyn91].

The professional work practices approach (PWP) aims to achieve successful SSD by improving developers' professional work practices. The improvement is to be achieved by learning that combines study (such as reading documents and taking part in academic training) with experience (e.g., of changing working practices). Since PWP recognises that theories and methods are subject to ambiguous interpretation and their true consequences cannot be understood simply through theoretical work, it encourages developers to carry out practical experimentation in their concrete working situations. For example, it proposes the application of different textbook-based methods to the same project in order to gain familiarity with the details of SSD and the use of these methods. In this way, the developers gradually build a thorough understanding of their work habits through practical experience, and as a result the improved development of software systems is supported.

Cooperative design, with its emphasis on user involvement, stresses the way in which users, as developers, actually engage in developing the required software system as shown in Figure 1-1(c). This approach argues that technology should not always be applied in ways that constrain human work, but instead it should encourage reciprocal learning, whereby users and developers teach one another about work practices and technical possibilities through joint experience [CWG93]. In this respect, it acknowledges that a high degree of user participation and well-developed interpersonal interaction between developers and users is imperative for the good design of software systems. In particular, cooperative design practitioners criticise the rationalistic approach of system development, with its roots in scientific objectivism, and specifically avoid presenting any 'step-by-step' method [CWG93, Kyn91]. For cooperative design, no particular activity pattern can be set to guide reciprocal learning and creative design. Methods are thus seen more as resources to use in order to cope with diverse situations, and are not gathered into a single coherent framework [cf. Suc87].

Neither PWP nor cooperative design is, strictly speaking, a method-based approach (more specifically, neither is based on a 'step-by-step' method) [HKL95]. They provide a set of principles, concepts and techniques to support their social process perspective and guide the development of software systems. At the same time, they do not degenerate into an *ad hoc* approach, which would threaten SSD with the same problems experienced prior to the advent of methods, as discussed in [DS97, Pre97]. On this basis, it seems that both approaches might be classified as amethodical.

However, as Floyd argues in [Flo87], a social process view of SSD means that software should be regarded as an *emergent phenomenon* taking place in an evolving world with changing needs, and as the object of the processes of *learning* and *communication* occurring in SSD. This observation indicates that the social process of SSD should not only be characterised by human learning and interaction - it should also enable software systems to evolve in practice in response to these social activities, and form the base of further activities. That is to say, the system being developed should always reflect interpersonal interaction and learning in a significant way. Taking this into consideration, it is not surprising that most approaches that focus purely on social action are criticised for their 'impracticality' and 'low cost-effectiveness' [AF95, HKL95], and involve a time delay in adapting software systems to their rapidly changing environments. It is helpful to regard the amethodical approach which the author is seeking in order to overcome these problems as *a social process for SSD, but with more technical practices to enhance the viability of given social goals.* From this perspective, the PWP approach takes insufficient account of interpersonal interaction [HKL95], and cooperative design provides limited technical practices [CWG93]; hence, neither of these alternatives qualifies as the desired amethodical approach.

Research into Empirical Modelling[9] (EM) at the University of Warwick [Bey97, Bey98, BCSW99] has shed valuable light on the search for an appropriate amethodical approach. Since EM does not involve a definition of well-structured steps by formalised rules and algorithms but rather emphasises the need to improve understanding and create experience through repeated experiments and observations, it should not be viewed as a technical process like that invoked by a traditional formal procedure for SSD. At the same time, by regarding the software system being developed as a computer-based model (cf. [Leh98]) and furthering the development of this software system through modelling, EM enjoys strong technical practices. Through the autonomous interaction with the computer-based model, the modelling process is advanced situation by situation, just as human agents solve problems in everyday life. The situated modelling process enables the modeller (that is, the user of EM) to design and maintain his/her own way of learning and thus enriches his/her experience, but also enables the modeller to adapt the software system (that is, the computer-based model) to its evolving world. Indeed, research into

[9] The details of EM are reviewed in Chapter 2, and Chapter 3 provides a discussion of fundamental concepts underlying EM and other methods for SSD.

EM and its application to a number of case-study software systems have demonstrated its useful technical capabilities and helpful support for learning [Bey97]. However, as indicated above, a social process must take sufficient account of interpersonal interaction between participants. Previous work on EM has yet to reveal the extent to which EM can serve as a social process to guide SSD in a distributed environment [cf. ABCY94, BNOS90]. Hence, it is important to clarify the characteristics of EM from the social process perspective, in particular in relation to user participation and interpersonal interaction.

Extending EM to serve as a social process for SSD is the main motivation for the present research. The aim is to integrate EM with social practices so that it can better support the social process of SSD in general and, more particularly, the interpersonal interaction between developers and users for the purpose of exploring, expanding, experiencing and communicating their knowledge. In motivating and justifying the application of EM to SSD, the thesis examines the following fundamental issues:

- **Essential character**

  What is the essential nature of SSD?

- **Real-world context**

  What is the relevance for SSD of the real-world contexts in which a software system is to be developed and used? How can sufficient account of these contexts be taken during the process of SSD?

- **Human factor**

  What are the roles for human agents in SSD? Are the developer and the user best seen as directors, guiding the process of SSD, or as actors, acting out the process by following the recipe of a method?

- **Social factor**

  What kind of interpersonal interaction in SSD is appropriate? In what respect can interpersonal interaction support SSD?

- **Computer support**

  What role can the computer play in supporting human agency and interaction in SSD?

As far as this thesis is concerned, the most significant issue is that of computer support for SSD. The proposed application of EM principles to SSD forms the main topic of the thesis (Chapter 4-7). It is the radical implications that this proposal has for SSD that motivate the broader agenda, and entail a comprehensive reconsideration of fundamental issues for SSD (see Chapter 2 and 3).

1.2 Thesis Outline

The principal aim of this thesis is to investigate distributed Empirical Modelling (DEM) and its applications to SSD. A framework for DEM is developed by drawing on two important theories in social science: *distributed cognition* [Hut95] and *ethnomethodology* [Gar67]. An application of this framework to requirements engineering is proposed. In addition, a tool has been implemented to support this framework. Also, several case studies are used to illustrate the concepts and principles of this framework. The rest of the thesis consists of seven chapters that are organised as follows.

Chapter 2 reviews and illustrates the concepts and principles of EM that form the basis of this thesis. First (Section 2.1), the concept of *situated activity* whereby human agents solve problems encountered in everyday life is identified. By using the computer as a modelling tool, this human-centred process can be enhanced. It is this kind of human-centred, computer-based process that EM attempts to invoke for SSD. Section 2.2 presents the framework of EM, including its basic concepts and principles, and the process of enacting EM. There then follows a discussion of technical issues supporting EM (Section 2.3). An example to illustrate the enaction[10] of EM is provided in Section 2.4.

Chapter 3 highlights the potential for using EM as an open development model for SSD. Two fundamental comparisons between EM and traditional phase-based process models for SSD, on the basis of their dynamic and static features, are given in the first two sections. One seeks to explore the differences between these two approaches from the perspective of process enaction, and the other focuses on the ways in which they manipulate the collected information. On the basis of these comparisons, the following

[10] As in [STM95, Tul88], the novel word 'enaction' is adopted in preference to 'enactment' to reflect the way in which human action can be closely integrated with computer execution in EM.

section (3.3) considers the use of EM for SSD. The advantages and limitations of using EM for evolving software systems which are ill-defined and volatile in their operational domain in the real world are discussed.

Chapter 4 aims to establish the framework for distributed Empirical Modelling (DEM), drawing on concepts from distributed cognition and ethnomethodology. Section 4.1 describes the reasons for developing DEM and the theoretical background to DEM. A detailed framework for DEM is then proposed (Section 4.2). This section introduces a central concept of DEM – '*pretend play*', whereby modellers shape the agency of agents within the system by pretending to act as these agents do in interacting with each other. One of the main issues of DEM concerns the method of shaping agency. The way in which DEM differs in this respect from AI and previous variants of EM is discussed in Section 4.3. Finally (Section 4.4), three strategies for developing software systems are identified and discussed. The use of different strategies distinguishes software system development by design (that is, developer-centred) from development by evolution (that is, participant-centred).

Chapter 5 is concerned with the implementation issues in creating a tool to support DEM. Section 5.1 discusses the creation of this supporting tool, called dtkeden. A distributed architecture with client/server communication is devised to implement this tool. Also, a new mechanism for supporting distributed synchronous communication in dtkeden is proposed. Then (Section 5.2), different interaction modes implemented in dtkeden for supporting different styles of interaction between modellers are discussed. In Section 5.3, the concept of *virtual agent* is introduced. The implementation of this concept in dtkeden provides a convenient way to specify several instances of a feature in a model by introducing the same definitive script in different contexts. It can also be applied to the dynamic reuse of a definitive script. A comparison between the proposed reuse in dtkeden and component reuse, based on abstract data types (ADTs), is also provided.

Chapter 6 illustrates the framework of DEM and the functionalities of dtkeden through case studies. The first section uses the example of an historic railway accident to demonstrate the concepts of pretend play and collaborative interaction between several computer-based models in a distributed environment. The main concerns of DEM, and the key functionalities of dtkeden are shown in this case study. Section 6.2 highlights the important concept of virtual agent through examples. One example involves the development of a new translator for generating Eden programs from ADM programs. Another two examples illustrate how this concept can be used to generalise reusable definitive scripts. Section 6.3 then focuses on the use of interaction modes provided by dtkeden. Examples of different interaction modes are presented.

Chapter 7 considers the application of DEM to requirements engineering (RE). In the first section, the basic concepts of requirements engineering are reviewed. Three kinds of model for the RE process (REP) are identified, and the difficulties of enacting them are discussed. Section 7.2 then considers the reengineering of the REP in order to reduce these difficulties. Context and human involvement are taken into account for this purpose. A human-centred framework for the situated process of RE, called SPORE, is proposed in Section 7.3. This framework regards requirements as 'solutions to identified problems' in the real world. These solutions are 'cultivated' by people taking part in the REP through their collaborative interaction with each other in an interactive, situated manner. Two examples of cultivating requirements in the framework of SPORE are given in section 7.4.

Chapter 8 brings the major findings and conclusions of the thesis together, discusses the limitations of the present research, and also examines the potential for further research.

1.3 Research Contribution

This thesis is intended to overcome some significant limitations of EM as currently practised. Its main objective is to construct a framework for distributed Empirical Modelling (DEM) that can support the modelling activities of several modellers in a distributed environment. By drawing on the concepts of distributed cognition and ethnomethodology, a framework that highlights not only the distributed perspective on EM but also the principles and concepts of EM is established for DEM. Within this framework, this thesis proposes the concept of pretend play to help modellers to shape the agency of agents within the intended system by acting in the role of such agents. In this way, each modeller's knowledge, associated with the software system being developed or used, can be conveniently explored, easily extended, substantially experienced and effectively communicated. Thus, the difficulties arising from ineffectual interaction between users and developers for SSD, and from discounting the context in which a software system is developed and used, can be significantly alleviated and, as a result, SSD can be better supported.

The work in this thesis also contributes a new process for integrating requirements with their real-world context in order to better support requirements development. The proposed situated process of requirements engineering (SPORE), which regards requirements as 'solutions of identified problems', allows the participants involved in this process to cultivate requirements in an incremental manner. As a result, the problem-oriented requirements development is intertwined with software system development. This close relationship not only facilitates the collaborative interaction between analysts, designers and users, but also promotes a seamless integration of specification, implementation and use of the software system. Furthermore, SPORE, by using computer-based models as a communication medium, also alleviates the communication bottleneck created by passive paper-based communication between participants.

In addition, for software system development, this thesis proposes a new strategy to support the evolution of a software system in the real world. This strategy requires a software system to be developed as an open-ended, computer-based model whereby not only developers but also users can adapt the system to cope with continuous changes, even in its operational domains, in an interactive, situated manner. Through this strategy, the drawbacks of developer-centred development, such as the problems of tacit knowledge, can hopefully be overcome.

In addition to these theoretical contributions, this thesis also develops the practical tool dtkeden for supporting DEM. This tool provides modellers with a distributed environment that has four different types of interaction mode in order to support their collaborative interaction through networked computer-based models. Within this collaborative work environment, a software system is distributed to connected computer-based models and can be incrementally developed in response to the understanding of modellers. The use of dtkeden illustrates in practice the advantages of an amethodical approach to the development of software systems. Moreover, the concept of virtual agent is developed and implemented in dtkeden in order to support the need to use the same definitive script in different contexts. This novel concept is also applied to reengineering an ADM-to-Eden translator to generate more readable Eden programs. It also enables the dynamic reuse of a definitive script, both in order to reduce the size of programs and to overcome the difficulty of maintaining these programs. The thesis includes several case studies to clarify the framework for DEM and demonstrate the functionality of dtkeden. Of these case studies, the animation of a historic railway accident is the most significant [Bey98, BS99, SB98].

Chapter 2
Empirical Modelling

Empirical Modelling (EM) is a novel approach to human-centred, computer-based modelling that has been developed at the University of Warwick over the last ten years. Its character and working principles embrace several different disciplines, especially psychology, cognitive science and computer science. Since the research for this thesis is based on the framework of EM, this chapter will begin by clarifying the fundamental principles and concepts of EM.

2.0 Overview

This chapter reviews and illustrates the concepts and principles of EM that form the fundamental basis for this thesis. Section 2.1 provides a general introduction to EM principles through an examination of scenarios that occur in everyday life. First, the concept of a *situated activity*, in which human agents solve problems encountered in everyday life, is introduced. A situated activity is to be distinguished from processes centred on traditional rationalistic[1] algorithms. Within a situated activity, the human

[1] The term 'rationalistic' is used in this thesis in the same way that T. Winograd and F. Flores use it in [WF86]. It denotes the view that the formulation of systematic rules can be used to capture the principles of an observed phenomenon in the real world.

factor, as the most important dimension of a soft process[2], is addressed. By comparison, traditional rationalism gives little attention to the role of human agents in dealing with diverse situations through the enaction[3] of a soft process in the real world. Moreover, the need of using the computer as a tool to facilitate the cognitive activity of a human agent is identified, in particular for a situated activity. Finally, the way in which EM research seeks to support a human-centred, computer-based approach through situated activity is highlighted.

Section 2.2 illustrates the framework of EM. Its basic concepts, observable, dependency, agent and agency, are defined in the first subsection (2.2.1). Subsection 2.2.2 then discusses the enaction of EM on the basis of constructing and maintaining the correspondences between the 'mental model' of its actor, called the modeller, the computer model, and its referent[4] in the real world. In establishing these correspondences, the modeller identifies primitive elements in the referent corresponding to the fundamental concepts above, records them by introducing appropriate definitions, functions and actions into the computer model, and also metaphorically explores, expands and experiences diverse states of the referent by interacting with the computer model. As a result, the enaction not only enhances the modeller's understanding of the referent, but also generates an interactive computer-based model as a by-product. With reference to software systems development, this by-product is exactly the evolving software system that is being constructed in the light of the modeller's current understanding of the referent. In other words, EM views a software system as a *computer-based model*, and the

[2] A *soft process* in this thesis refers to an intelligence-intensive process, such as developing a software system, designing a new car model and investigating an accident. By contrast, a *hard* process refers to a formally-defined and well-structured mechanism, such as the manufacturing process of an assembly line and the procedural instructions for operating a machine.

[3] The term 'enaction' rather than 'execution' is used in this thesis for the reason given in [STM95, p.17]: to highlight the embedding of human and computer-aided human activities in the model.

[4] The term 'referent' refers to the subject in the real world being observed by the modeller. From the perspective of EM, the referent is open and liable to change. In this thesis, the phrase "the referent associated with the subject in the real world" is often simplified as the term 'the referent'.

development of this system as *modelling*. More details will be given in the next chapter (Section 3.3).

Section 2.3 discusses technical issues supporting the enaction of EM. First, several tools previously developed for EM are reviewed (Subsection 2.3.1). By using these tools, the modeller can create a computer model and enact EM in an interactive and exploratory fashion. Amongst them, the tool tkeden has proved to be particularly successful in supporting the principles and concepts of EM. Underlying the tool tkeden is the concept of definitive programming [Yun92], explained in subsection 2.3.2. This kind of programming captures the dependencies between objects, and between objects and their properties, by declaring definitions resembling formulae in a spreadsheet. The use of definitive programming makes it possible to enact EM as a situated activity.

In the final section (2.4), the example of a hotel booking system is used to demonstrate the concepts and framework of EM. This example, developed using tkeden, also reveals the advantages of definitive programming in supporting the enaction of EM and constructing the intended software system as a computer-based model.

2.1 Meeting EM in Everyday Life

It may be difficult for a novice to understand the basic concepts and principles of EM. This is because EM is not an approach to solving a problem on the basis of traditional rationalism, where it is presumed that a good solution can be obtained by following a rigid process and abstract rules. Instead, within EM, the method of solving a particular problem is based on intelligence captured through practical experience [Bey94] – a method that human agents tacitly use to solve problems encountered in the real world.

In fact, the fundamental principles of EM are neither elusive nor intricate. Indeed, it is natural and essential for people to use these principles to solve problems in everyday life, even though they are rarely made explicit. However, in spite of their simplicity, these principles cannot easily be formulated as rigid processes and rules. One of the best ways to understand the main principles of EM, therefore, is by considering scenarios in everyday life. As examples of scenarios:

- A person is driving through London during the rush hour and intends to arrive home as early as possible.
- Friends meet each other in the street and carry on a conversation.
- A student is using a word processor to edit a text file into a particular format.

Although such scenarios involve different situations and serve different goals, they have one important thing in common: a coherent sequence of situated actions, called a *situated activity* in this thesis, that is being constructed by the interaction between a human agent and his/her environment[5]. An action is situated if it involves conscious reference to context and choice of course of action. An action is not regarded as situated if it takes the form of a prescribed response (that is, "I am not responsible for my choice of action") or if it is an unconscious automatic response (that is, "I am not aware of my choice of action"). For example, in the first scenario, the situated activity for the driver can include overtaking other vehicles, speeding up when the traffic is good, changing to an alternative route when the traffic is too busy, and so on. In the same manner, the situated activity for one of the friends in the second scenario involves listening and replying to the speaker, changing the subject, getting distracted by other people or things, and so on.

These scenarios show that a situated activity is very different in character from an activity that is specified by a formal algorithm (such as the operation of a machine by following its instructions). Within a situated activity, each situated action, described by L.

[5] Although the term 'environment' can be used in a very broad sense, which incorporates external surroundings and the internal mind, it is used here to refer only to the external surroundings of an individual, unless otherwise indicated.

Suchman as a dynamic interaction with the actor's external environment [Suc87], is very difficult to prescribe in advance[6]. Examples can be readily found in these scenarios, such as overtaking other vehicles in the first scenario, answering a question in the second scenario and relocating a heading that appears at the bottom of a page in the third. Unpredictable events require human agents – through uniquely human capacities[7], such as intelligence, experience, knowledge and the ability to use tools – to deal with each emerging situation in ways that cannot be preconceived. For this reason, it is in general hard to prescribe a situated activity by means of a formal (or semi-formal) algorithm through which a human agent can interact with a specific environment through preconceived, fixed and well-defined methods or rules.

In fact, one of the problems with any activity formally defined by an algorithm, if it is to address the need for greater flexibility and realism, arises from its adherence to certain rigid steps or fixed methods [Gog94, Tul95]. In particular, the rationalist emphasis on regarding a specific situation as simply an instance of a more general class of similar situations abstracts an activity in the real world from its context and turns it into a routine mechanism. Because of this abstraction, and because of the inherent openness of the real world, it is hard for a formalised process to express contingent knowledge of the real world in terms of inductive inference and predetermined stimulus-response patterns [Agr95, Fey75, Suc87, WF86].

Most software process models based on a formal method require developers to follow a set of sequential activities that are well-structured and formally defined [Boe88, Boo94, STM95]. However, it has been increasingly recognised that developers in practice

[6] As Suchman argues in [Suc87, p.52], plans that are prescribed can be regarded as "resources for situated action, but do not in any strong sense determine its course".

[7] It is very difficult to find the right word to include all details of these capacities, since they are all intertwined and interdependent. For the sake of convenience, the term 'knowledge' will be used to exemplify these capacities in this thesis. However, this is not intended to suggest that knowledge is the only capacity of human agents.

have difficulty in respecting such rigid protocols [Fit96, Leh98, Rac97, SAGSZ97, Suc87, Tul95, WF86], since the real environment confronting them is usually intricate, chaotic and unpredictable. The real activities enacted by developers are to a large extent a form of situated activity. That is to say, they take situated actions without reference to a specific algorithm in order to cope with each emerging situation. From this perspective, the concept of situated activity is arguably necessary in supporting SSD (in fact, it forms the basis for the amethodical approach to be proposed by the author in this thesis (see Chapter 1)). More detailed discussion of this issue is provided in the next chapter.

A situated activity is more versatile than a formalised activity for solving problems in the real world. The most significant reason is because it is centred on human agents rather than on strict laws, algorithms or so called 'plans' arising from a particular account of the world. In effect, most plans are simply used by human agents as a resource rather than as a source of control in everyday life [Suc87]. In a situated activity, it is most appropriate to give human agents autonomy for problem-solving purposes. By reflecting on the surrounding resources, such as known information, individual experience and knowledge, and the current state of the environment, human agents can conduct reasoning in their minds to 'preview' possible results, and can consequently undertake corresponding action towards a new expected or unexpected situation. Each action undertaken, by promptly and tacitly affecting both the internal mind and the external environment, leads to a new situation and concurrently enables the situated activity to progress. In other words, situated activity, instead of prescribing preconceived activities and specifying the stimuli-response relations between human agents and their environment, highlights the importance of human agents coping with diverse situations in the real world by taking the context into account.

In short, typical problem-solving in a situated activity, as described here, reveals two features:

- The solution to a problem is *context-dependent*: it cannot be separated from the problem's context and then specified in a rigid way that does not take its situatedness into account.
- The solution is *human-centred*: human agents, whose capacities can still not be circumscribed or predefined through any formal logic or rules, play an essential role in providing a situated solution.

Certainly, a formalised process can enjoy the benefit of high quality assurance associated with an engineering discipline. However, the enaction of a situated activity that is context-dependent and human-centred is arguably necessary in dealing with real world complexity and uncertainty. Hence, a soft process is most appropriately enacted as a situated activity, that is to say, taking full account of human agents and the context.

Relying upon situated action definitely has its disadvantages. For one thing, by virtue of being human, a human agent at the centre of situated activity is inevitably error-prone and forgetful, learns slowly from experience, and can be seriously distracted by his/her environment [Hal89, Nor83, RB74]. These human factors are bound to influence not only the end-result but also the structure of the situated activity. In practice, these disadvantages caused by human factors also occur in most rationalistic models, but they are deemed to be too philosophical and open-ended to take into account. For this reason, most of these models leave the relationship between human agents and the enaction of situated activity open. In effect, such models take it for granted that the reasoning and thinking of human agents has the same internal coherence and consistency that would be expected of a mathematical model. However, this assumption is dubious when such models are interpreted in the real world, due to the openness of the environment and the inevitable fallibility of human agents.

The degree of insight the human agent has into his/her situation determines the quality of a situated action. This insight is expressed in terms of awareness of relevant factors in the situation, and appreciation of the probable implications of action. In effect, most of drawbacks of situated activity stem from limitations of human agents in respect of cognitive activities [Nor83], such as understanding, thinking and reasoning. Fortunately, history shows that the effective use of tools can to a large extent assist human agents in performing these activities. For example, pencil and paper facilitates reasoning for most people [FP88], LOGO games facilitate the thinking of children [FSCSF88], and a physical model facilitates the understanding of physicians and chemists in solving a problem [RB74, diS88]. Today, it is widely believed that the computer is one of the best tools for human beings to facilitate the performance of these cognitive activities [Cro94, DS97, FP88].

However, it is exceedingly difficult to make effective use of the computer to support cognitive activities. For example, even though computer-based tools are already used for many rationalistic models, they can only provide limited help. This is because activity based on a formalised process is dominated by its algorithms independently of its context. Most tools developed on the basis of the algorithm for supporting the process cannot help but be context-independent. They are limited to dealing with the static information prescribed in advance rather than capturing the dynamic information emerging from the process itself. In other words, any information must be perceived and specified in the early stages of the process; otherwise the tools can take no account of it. This prohibits the tools themselves from coping with any unpredictable situation, a norm in the real world, and inevitably limits their advantages. Diverse CASE (Computer-Aided Software Engineering) tools exhibit this limitation. In view of the practical evidence, some researchers doubt whether these tools, based on specific algorithms, can provide sufficient support for software development in the real world [Blu93, BD93, Bub95].

From this perspective, it is important to use the computer in ways that best support the cognitive activity of human agents in response to the openness and unpredictability of situated activity. Recognising this, EM seeks to provide an approach that enables a human agent engaging in a situated activity *to use the computer as an open-ended artefact to explore, expand and experience his/her understanding of a situation, as gauged by their ability to construe phenomena and anticipate the consequences of action.* This human-centred, computer-based approach has been promisingly applied to AI [Bey98], educational technology [Bey97], concurrent engineering [ABCY94], creative software development [Nes97], geometric design [Car98], and requirements understanding [SB98]. Ongoing research is applying this approach to decision support systems, business process modelling, program comprehension [BS98] and software reuse.

2.2 The Framework of EM

EM is associated with enacting a soft process characterised by the features of situated activity, but also drawing on the special capabilities of the computer to overcome the limitations of human cognition. This section gives more details of what EM is and how it works. First, the basic concepts of EM are identified. Then, the process of enacting EM is described, and close attention is given to two key activities involved in this process: observation and experiment.

2.2.1 The Basic Concepts of EM

Due to the openness of the real world, it is very difficult and provides little help to specify a situated activity in a preconceived form. For example, in the scenario of driving home (see Section 2.1), it is not sensible to preconceive the presence of a dog on the driver's way home or that the radio broadcasts that a world crisis is over. For this reason, it seems to be plausible that a situated activity can only be described in a situated manner, that is, situation by situation. In other words, a situated activity can only be understood by modelling the interaction between its enactor and his/her environment with reference to the situations that pertain moment by moment rather than by appealing to an abstract conception of his/her behaviour. For this purpose, it proves useful to *construe a situation* in terms of the following concepts: observables, dependency, agency and agent.

- An **observable** is a characteristic of a subject to which an identity can be attributed.
- A **dependency** represents an empirically established relationship between observables.
- An **agent** is an instigator of change to observables and dependencies.
- An **agency** represents an attributed responsibility (or privilege) for a state change to an agent.

The above concepts are very general and broad. For example, the highway code can be regarded as accounting for car-driving in terms of observables (such as traffic signs, indicators, and traffic lights), dependencies (such as the relationship between the car's speed, and the speed-limit signposts and traffic lights), agents and agency (for example: a driver is responsible for stopping his/her car when he/she sees a traffic light on red). It is not too difficult to identify similar concepts in methods for SSD: for example, entities and relations in an entity-relation model [Che76], and objects and classes for an object model [Boo94, CY90]. However, as in the highway code, the intention behind these models is to use these concepts to specify a process that is in essence a situated activity in a preconceived form. As explained above, this is inadequate and provides limited help for the real process, which cannot be specified in advance. In contrast, EM makes effective use of these concepts in an open-ended fashion.

An observable in EM can be physical or abstract in nature, as illustrated by the following examples: the power of the newly designed engine, the position of the approaching aeroplane, the cry of the white seagull, and the time on Big Ben. In conceiving a situation encountered in a situated activity, a family of relevant observables is implicated. In modelling situation-by-situation, the presence of observables can be intermittent rather than persistent, so that an observable can appear or disappear at any moment in response to each situation that is being construed. For instance, in the driving scenario, an observable, such as the dog, appears to the driver only whilst it is running past his/her car.

A dependency in EM is not merely a constraint upon observables, but reflects how the act of changing the value of one particular observable is perceived to change the values of other observables predictably and indivisibly. In this respect, dependencies play a significant part in construing a phenomenon [Bey98]. For example, in the driving scenario, it is found that the view in the rear mirror is determined by following traffic, and the car's acceleration depends on the position of the accelerator pedal. In a procedural interpretation, dependencies invoke hierarchical processes that can propagate the effect of redefining the state of any observable to all relevant observables directly or indirectly dependent on this redefined observable. For example, the brake lights are on when the brake pedal is depressed, and the brake pedal is depressed when the driver's foot pushes down on the pedal. Moreover, like an observable, a dependency need not be permanent but can instead be provisional. For example, on an icy road, the direction of motion of a skidding car no longer depends on the position of the steering wheel.

In EM, identifying agents and their agency is "associated with attributing state-change to what is construed as their primary source" [BeyMsc]. Beynon argues that agency is "in the mind of the external observer" and is "shaped by the explanatory prejudices and requirements of the external observer, and by the past experience of the system" [BeyMsc]. A typical question that helps to identify agents and agency is: "who are/is responsible (or who have/has privilege) for this state-change?" In the driving scenario, the dog and the driver are agents when the responsibility (or privilege) for state changes, such as control over their movement, is attributed to them.

It should be noted that the concepts of agent and agency in EM are quite different from traditional agent models in the AI field, where an agent is generally defined as an entity and its ability to perform a preconceived behaviour is called agency. The specific entity is often granted or ascribed human-like mental states and is capable of interacting with its external environment in terms of these mental states [DBP93a, Rao94, Sho93, WJ95, BT94]. Hence, these models stress the conceptualised mechanism of an agent. In contrast, EM merely acknowledges the fact that agents come to be recognised by the modeller, and regards agency as being shaped by repeated observations, interactions and experimentation (see Section 4.3 for more details).

When these concepts are used in a situated, open-ended manner, the challenge of construing a situation is to provide for their computer support. In EM, definitive notations have a basic role in providing such support. A *definitive notation* is a simple programming notation for formulating definitions. A *definition* is a formula of the form x = f(y1, y2,...) similar in character to a formula in a spreadsheet[8] [Yun92]. The value of the variable x (dependent) is always equal to the evaluation of the user-defined function f with the current values of these variables y1, y2, ... (the dependees[9]). Any change to the value of a dependee will give rise to a re-evaluation of the value of the dependent. For example, the definition 'A is B+C' indicates the dependency of A on B and C so that any change in the value of either B or C will cause a re-evaluation of A.

[8] More complicated definitions could be considered from the perspective of higher-order dependency [GYCBC96], which is being investigated by D. Gehring. This kind of definition is not taken into account in this thesis.

[9] The new word 'dependee' corresponding to 'dependent' is made up to denote that the value of a dependent variable is determined by those of its dependee variables.

The observables and dependencies associated with the current situation can be expressed by a set of definitions called a *definitive script*. The ordering of definitions in a script is unimportant. A redefinition of a variable automatically brings all its dependents in a definitive script to a new state through a propagation process that re-evaluates all its dependents. For example, in the definitive script 'X is Y+A; A is B+C', any change in the value of either C or B will cause a re-evaluation of X (more details of implementation issues are given in the next section).

2.2.2 Enacting EM

EM is a powerful form of interactive modelling. It allows the modeller to use the computer to create an artefact[10], an interactive computer-based model with something of the character of an engineering prototype. In order to enact EM, the modeller must first build up a virtual correspondence (as shown in Figure 2-1) between the computer model and its referent. In enacting EM, the modeller 'embeds' knowledge about observables, dependencies, agents and agency in the computer model. Interacting with the computer model allows the modeller's insight into the situation to be accessed.

Such a computer model of a situation is versatile – it can be used in dealing with many different subjects. The subject is typically an intelligence-intensive process (a soft process in the terminology of this thesis), such as developing a software system, understanding requirements, designing a geometric model, or investigating an accident. In using EM in SSD, it is envisaged that the same model may be used for understanding requirements of a software system and for its subsequent development.

[10] An artefact used for a situated activity can be a physical model, a graphic drawn on a piece of paper, a computer model, and so on. However, within the framework of EM, it is recommended that the artefact can be constructed as an interactive computer model in order to make the best use of the computer's advantages, as discussed in the previous section. Therefore, the terms 'artefact', 'computer model' and 'computer-based model' are used interchangeably in this thesis.

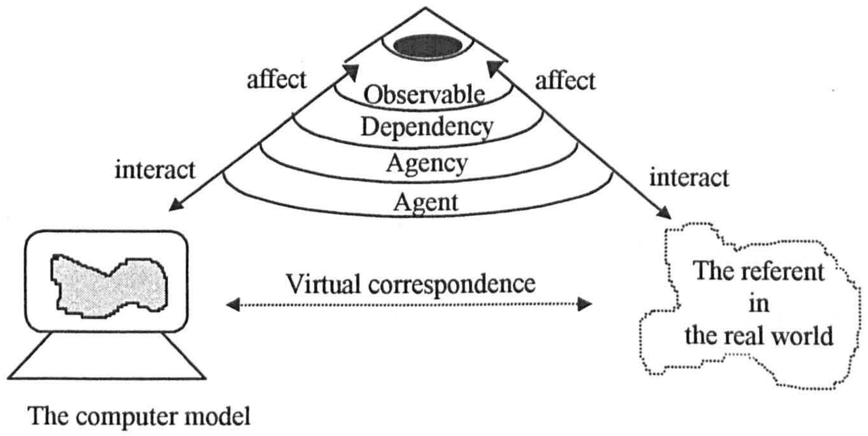


Figure 2-1. The virtual correspondence within EM

It is convenient to conceive the virtual correspondence in Figure 2-1 as established by two auxiliary correspondences. These auxiliary correspondences connect the modeller's 'mental model' of the current situation with the computer model and with the referent respectively. The concept of a mental model is introduced to acknowledge the fact that the modeller generally has insights, beliefs, and expectations of the situation (cf. the characterisation of knowledge in footnote 7) that have yet be taken into account or are in conflict with the computer model or the referent. The correspondence between the mental model and the referent is established by the interaction depicted on the right-hand side of Figure 2-1. This interaction between the modeller and the referent enables information exchange and creation. Each change that occurs in the referent, whether it is triggered by the modeller or not, may affect the mental model. The correspondence between the mental model and the computer model is established by the interaction depicted on the left-hand side of Figure 2-1. In this case, the modeller is empowered to interact with the artefact, and may also be affected by any change in the artefact. The insight gained by the modeller through establishing the virtual correspondence is expressed in coherence between an abstract explanatory model – or *construal*[11] – in the modeller's mind, the physical embodiment of this construal in the computer model, and a situation in the referent.

In constructing the virtual correspondence, the modeller can identify primitive elements in the domain being modelled corresponding to the fundamental concepts above, and then record them by introducing appropriate definitions, functions and actions[12] into the computer model. A typical step in this process involves the identification of a dependency and the introduction of the definition into the computer model. From the modeller's perspective, a definition is "recording a dependency between the observables". From a computational perspective, the abstract semantics of introducing such a definition is typically similar to that of introducing a new formula into a spreadsheet to which a visualisation of cell values is attached. In particular, the dependencies amongst observables are automatically maintained (that is, any change in a dependee is propagated to all its dependents. More details are given in Section 2.3.2). Unlike traditional programming codes, definitions do not have to be entered and organised sequentially. Because of these properties, the construction of such computer-based artefacts is a useful vehicle for exploring and developing insight.

EM is a means of constructing knowledge in an experiential rather than a declarative fashion: the modeller's insight is expressed as coherence between expectations in the mind and the experimentation that can be performed on the computer model and/or in the referent. The principle resembles 'what if' experiments with a spreadsheet. The modeller introduces new definitions to impose a change of state upon

[11] D. Gooding introduces the term 'construal' in analysing Faraday's experimental practices. He regards a construal as "a means of interpreting unfamiliar experience and communicating one's trial interpretations" [Goo90, p.22] and argues that "a construal cannot be grasped independently of the exploratory behaviour that produces it or the ostensive practices whereby an observer tries to convey it" [Goo90, p.88].

[12] Actions are specified as procedures that are triggered by changes in the values of a particular variable. [Bey97].

the embodied construal, that is, the computer model. Almost simultaneously, the new state of this construal is mediated to the user through the visual interface, and evokes a change of state in the mind of the modeller. When this change of state is consistent with the modeller's expectations, it serves to reinforce the modeller's confidence in the way in which a situation has been construed. When the change of state confounds expectations, the modeller must determine whether the situation has been construed in an inappropriate way, for example by giving an incorrect definition, or whether a hitherto unsuspected behaviour has been identified. In the latter case, there is a creative and often surprising element of discovery that is rarely encountered in conventional modelling.

In fact, the modeller not only enriches but also to some degree embodies his/her mental model through the continuous interactions with the computer model. This is because the computer model is incrementally developed to correspond to the mental model and then to the referent. More details are provided in the next chapter.

Theoretically, the enaction of EM is unbounded since it is not possible to take all situations associated with a particular subject into account. As in everyday life, the modeller continually confronts different unpredictable situations. A new situation can cause a discrepancy between the modeller's mental model, the computer model and the referent in Figure 2-1. The modeller may interact with both the computer model and the referent in order to resolve such discrepancies. Situated interaction of this nature reflexively constitutes the situated activity of enacting EM, which cannot be prescribed by algorithms in advance. It also accounts for the openness of EM itself.

Obviously, the main crux of enacting EM lies in maintaining the virtual correspondence. With reference to the right-hand side of Figure 2-1, the modeller's interaction with the referent is not constrained by an explicit interface. Like an experimenter, the modeller may not be aware of what actions can affect the states of the referent. The interaction with the referent is open subject to empirically established knowledge of the observables that can be changed, and the associated dependencies. Interaction with the computer model on the left-hand side must be supported in the same manner. For this purpose, the computer model must have automatic dependency maintenance, that is to say, the model must be appropriately restructured in an automatic fashion in response to any change to its elements. This feature allows the realisation of the maintenance of the virtual correspondence. The necessary supporting technique is provided in the next section.

EM does not claim that using a computer model as an artefact is the only way to model a situation. After all, the human brain, supported by paper and pen, has performed the same task quite effectively for hundreds of years. However, as explained in the previous section (2.1), the computer has unusual potential as a supporting tool for helping to overcome human cognitive limitations in information processing. Many cognitive activities of human agents, such as reasoning and remembering, can be greatly improved by externalising them to the computer. For example, in a 'what if' experiment in a spreadsheet, the modeller can 'observe' rather than 'imagine' or 'conjecture' possible results from the artefact. This helps the modeller to reason more quickly and with more confidence. In this sense, the computer model does serve as an artefact for improving human cognition [Nes97, Rus97].

However, it is evident that most cognitive activities of human agents are too complicated and sometimes insufficiently predictable to be completely automated. Recognising this fact, EM makes best use of the capacity of the computer by delegating to it routine and structured tasks that involve complex calculation and huge demands on memory. At the same time, EM highlights the role of human agents in a situated activity, allowing the modeller to carry out intuitive and situated procedures, such as the identification of observables, dependencies, agents and agency. In this respect, the enaction of EM is consistent with C. Tully's concern about the mechanism of enacting a software process model, which on the one hand is "a symbiosis of human agent and computer" and on the other hand should be such as "not to hint at particular roles for either partner" [Tul88, p.3].

Both auxiliary correspondences are reached through various interactive activities, such as observation, experimentation, creation and so on. Since the first two are the primitive and critical activities in EM, a more detailed explanation of them is given here.

- Observation

In EM, observation, which refers to the modeller's ability to apprehend features of a particular situation directly, is vitally important. Without it, cognitive activity reverts to a traditional form: the modeller relies on imagination or conjecture without any assistance from suitable tools. Observation can be invoked on both sides of Figure 2-1, that is, to observe both the referent associated with a subject in the real world and the computer model. At least two correlated psychological events relating to the enhancement of the modeller's insight are necessarily involved: perception and connection (cf. [Hal89]).

Perception is concerned with identifying features in the computer model and/or in the referent. Connection involves associating these features with the mental model. The performance of each event is deeply bound up with factors affecting cognition, such as past experience, subjective belief, the understanding of a situation, and so on. At the same time, the result of performing both events leads to an alteration in the modeller's mental model and to the formation of a new state which influences subsequent activities. In other words, through observation, the modeller can not only construe the current situation but can also enrich his/her resources for dealing with future situations.

Perception and connection play significant roles in establishing the virtual correspondence between the computer-based model and the referent in the real world. They account for the way in which information about the referent is propagated to the computer model via the mental model, and vice versa. By this means, the referent can be metaphorically represented by the computer-based model. For example, placing a lamp on a desk can be represented and understood as placing a circle (representing the lamp) inside a rectangle (representing the desk). In the same manner, the state of the computer model can be referred to the state of the referent. For example, changing the position of the rectangle in the computer model can be referred to moving the desk.

It may be claimed that observations are also carried out in enacting traditional process models. In a narrow sense, a kind of observation is indeed performed. However, in these models, observation is intended to pin down elements whose nature is context-dependent within a particular context. These elements, e.g. entities and relations for an entity-relation model [Che76], and objects and classes for an object model [Boo94, CY90], are preconceived, prescribed and then isolated from the proceeding process until – in view of a new functionality or context – a further change of these elements is required. In other words, this kind of observation serves to draw a line to separate the developed model from the referent. Accordingly, the developed model, which prescribes a frozen domain, becomes well suited for the use of orthodox tools and methods that are devised for implementation. This separation can make the implementation more effective and robust, but at the price of being less adaptable (details are given in Chapter 3).

- Experimentation

The choice of the epithet empirical reflects the pivotal role that experimentation plays in EM. In effect, it plays a 'creator' role for modelling a situation in EM, since it always 'creates' diverse new states with surprising discoveries that can enrich the procedure of modelling a situation. Without experimentation, modelling a situation will be reduced to 'imagining' reliable patterns of state change in the same way that behaviour in conventional programming is preconceived in response to each particular situation. In this case, the method of modelling will degenerate into what Feyerabend in his book *Against*

Method has characterised as a 'scientific' approach, which in fact is not easily capable of discovering new ideas [Fey75].

Modelling a situation is the most elusive but fundamental aspect of the EM approach. As Beynon argues [Bey98], a situation should not be "interpreted as referring to an abstract computational state, but to something resembling a 'state of mind' that derives its meaning from a relationship between a human agent and an external focus of interest and attention". Modelling a situation involves devising diverse interpretations of the relationships between this situation and its diverse state changes. To do this, an animation of knowing-by-doing through 'what if' experiments is introduced. Instead of reasoning (or imagining) possible results in his/her brain according to the current situation, the modeller changes the state of the computer model (doing) to bring about a new state of his/her mind (knowing). In this way, the modeller can enhance his/her understanding of the situation and perhaps even make surprising discoveries by exploring unfamiliar territory.

Theoretically, experimentations can be invoked both in the computer model and the referent. However, EM puts greater emphasis on the computer model. This is partly because in many cases performing an experiment in the real world is very difficult and expensive, as is illustrated by the example of developing a new air traffic control system. More importantly, the modeller can make the best use of the power of the computer as an interactive modelling medium to achieve the principled theme of EM: to explore, expand and experience the modeller's understanding associated with the subject.

It should be noted that although observation and experimentation have been discussed separately here, they are inseparably invoked by the modeller in order to maintain the correspondences between the modeller's mental world, the computer-based model, and the referent in the real world.

2.3 Technical Issues of EM

Since EM highlights the importance of empirical experience arising from repeated observation and experimentation, it fulfils the requirements for *enactability* described by Tully in [Tul88]:

> If we set out to develop models, formalisms or representations [for SSD], then there is a strong case that they should be enactable – that is, should take form of 'process programs'. Enactability simply means that human beings involved in the software process receive computer guidance and assistance in what is an extremely complex activity. Put another way, models are not just used 'off-line', as a means of studying and defining processes, but also 'on-line' while processes are being carried out, as a means of directing, controlling, monitoring and instrumenting them.

In order to support the *enactability* of EM, the technical issues of supporting the principles and concepts of EM, especially the development of computer-based tools, must be considered.

2.3.1 Tools for Supporting EM

EM aims to enable the modeller to extend, expand and experience his/her mental world through the interaction with the computer model. To build up such an interactive artefact, several tools have been developed and these will be summarised briefly.

LSD (Language for Specification & Description) [Bey86] is an open-ended notation used to account for the referent in the real world. It provides a description of "those observables that are bound to an agent (*state*), those that it is conditionally privileged to change (*handle*), and those to which it responds (*oracle*)" [BR94]. It also includes an account of the dependencies between observables perceived by the agent (*derivates*) and of the actions it is conditionally privileged to perform (*protocol*). It should be noted that this description indicates the modeller's provisional construal of subjects and accordingly should not be viewed as a circumscribed specification, such as a requirements specification as defined in [LK95].

Within the enaction of EM, the LSD notation is useful for recording the identification and classification of agents, agency and observables associated with the modeller's observation of the referent and the model. In effect, an LSD account records the modeller's construal of state changes. Many different possible state changes and patterns of behaviour may be consistent with this construal. For this reason, an LSD account is not executable. To interpret an LSD account, the modeller needs interactive tools to realise and explore state changes consistent with the description. The tools ADM and tkeden, complementary to LSD and necessary for the enaction of EM, serve this purpose.

The tool ADM (Abstract Definitive Machine) is used to study parallel state-change, synchronisation of agent actions and openness in an LSD account [Sla90]. The modeller can manually transform an LSD account to an executable program in the ADM. An animation is then devised to give operational meaning to interaction between the LSD agents (such as what an agent can refer to in a particular state and how it can act to change the state). In this way, the modeller using the ADM can dynamically intervene and redirect the execution of this animation by interacting with this model in order to improve his/her understanding.

Another tool tkeden, one of the most successful tools for EM, is developed on the basis of the fundamental principles of EM. Its basic architecture is shown in Figure 2-2. In tkeden, there is a window-based interface based on a Tcl/Tk interpreter. This interface provides the modeller with an interactive environment to introduce new definitions into the computer model, and thus to observe their influence on the model's visualisation. Each definition is read into the Tcl/Tk interpreter as data and is stored prior to further manipulation. Visualisation of the computer model is established through two observational tools: DoNaLD [ABH86] and Scout [Dep92]. The former is a two-dimensional line drawing tool, and the latter deals with the issues of screen layout.

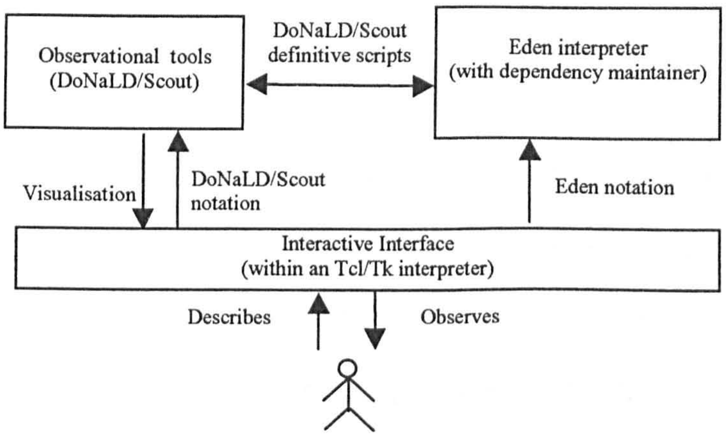


Figure 2-2: The architecture of tkeden

The core part of tkeden is an interpreter called Eden [YY88, Yun90]. This is both a definitive language for specifying definitive state transitions and also a virtual machine for maintaining the dependencies of given definitions in an interactive way. Each definition is maintained in the form of formulae resembling those in a spreadsheet. When the value of a dependee is changed, EDEN automatically propagates the change to its dependents and re-evaluates the values of these dependents. The premise that dependencies are automatically maintained is related to an indivisible state change propagated so as to reflect change in the referent rather than in a control mechanism in the programming sense. Nothing in such a model is preconceived, because no one knows what definitions will subsequently be introduced by a user.

Both tools, the ADM and tkeden, are interactive tools for supporting EM. They can be used independently and serve different purposes. The ADM focuses on the concurrent systems modelling needed in order to exhibit appropriate behaviours consistent with the LSD account. In contrast, the tool tkeden is more concerned with the visualisation of state-changes to observables and dependencies. It is often useful to combine concurrent systems modelling with visualisation. For this purpose, translators from the ADM to tkeden (adm and adm3 – see Section 6.2.1) have been developed. These allow an LSD account to be semi-automatically translated to a tkeden model.

2.3.2 Definitive Programming

One of the most important techniques behind tkeden is definitive programming, contributed by Y.P. Yung's PhD research [Yun92]. This technique, which refers to definition-based programming, seeks to "capture the dependency information of the properties within an object and between objects by means of definitions" [Yun92, p.5]. As described earlier, a definition is a formula of the form x = f (y1, y2, ...). The value of the variable x is always obtained by evaluating the formula.

Definitive programming uses definitive notations to establish a state-transition model in the computer. A definitive notation is a programming notation[13] that can be used for formulating a set of definitions. DoNaLD and Scout are two examples of definitive notations. A state of the model is then represented by a set of definitions – a definitive script – and a transition is accomplished by modifying the definitive script. This modification can involve overwriting an existing definition (redefinition) or just adding a new definition. Each such modification changes the current state of the model and leads to a new state by automatically re-evaluating the script.

To illustrate the concept of state transition in a model, consider the computer model that is constructed by EDEN by building up a definitive script step-by-step (in the syntax of EDEN and followed by its output) by introducing the following sequence of dependencies:

[13] It is described as a 'programming notation' rather than a 'programming language' because it represents only part of the information needed for general-purpose programming [Yun92, p.6].

1. Rectangle-area is Rectangle-length * Rectangle-width; writeln(Rectangle-area);

   → @[14]

2. Rectangle-length is 10; Rectangle-width is 20; writeln(Rectangle-area);

   → 200

3. Rectangle-length is 15; writeln(Rectangle-area);

   → 300

4. Cuboid-volume is Rectangle-area * Cuboid-high; Cuboid-high is 10; writeln(Cuboid-volume);

   → 3000

5. Rectangle-length is 12; writeln(Rectangle-area, ",", Cuboid-volume);

   → 240, 2400

The initial state of the computer model is established by giving a definition of observable *Rectangle-area* (step 1). By adding new definitions (step 2) and redefining an old definition (step 3), the state of the model is changed. Three different outputs, resulting from the first three steps, for the same observable *Rectangle-area* indicate that the re-evaluation is automatically executed. Given state 4, in addition to the explicit dependency between *Cuboid-volume, Rectangle-area,* and *Cuboid-high*, an implicit dependency between the observables *Cuboid-volume*, *Rectangle-length* and *Rectangle-width* is also established. This implicit dependency is demonstrated in the output of the last step in which the change to observable *Rectangle-length* (step 5) is propagated to the observable *Cuboid-volume*.

In other words, when a redefinition of an observable in a definitive script is given, a re-evaluation of the observables that are dependent on this observable is automatically

[14] In tkeden and LSD, the symbol '@' denotes 'undefined'.

invoked. The automated mechanism of maintaining the dependency between observables provides a very important basis for programming in an interactive and exploratory fashion [Yun92].

For SSD, this interactive programming technique enables the modeller to establish an incomplete computer-based model and improve the model incrementally. In this sense, programming becomes a matter of solving a problem rather than translating a specification. The translation of a specification can be accomplished in a single pass by a top-down or bottom-up approach without referring to the knowledge emerging from the on-going process. Instead, the problem-solving process as a situated activity must make progress incrementally in response to its situation and the emerging knowledge. Solving a jigsaw puzzle is a good example of the piece-by-piece solution of a task. A divide-and-conquer strategy is often used in this case. The most easily identified and more geometrically significant pieces of a jigsaw, such as the four corners, might be put in place first in order to provide further valuable information. Most solvers continue to add to the jigsaw piece by piece in response to the current state arising from the completed segments, rather than complete the jigsaw in a particular sequential order, such as from the upper-left corner to the lower-right corner, without referring to the emerging knowledge.

The value of a variable within definitive programming can be undefined and can be automatically revised. This feature enables the programmer to define variables in accord with their semantics. For example, even though its two dependees (length and width) are not defined yet, the definition of the area of a rectangle can be given (as in step 1 above). Hence, the task of programming can be accomplished by local adjustment (that is, by a piece-by-piece strategy). The incremental development feature is difficult to achieve for traditional programming, in particular for procedural programming. This is because, for traditional programming, any change to a program typically must be performed from a global viewpoint, since it could affect other parts of this program. However, with definitive programming, each local change is propagated to the whole program and leads to the necessary re-evaluation. For example, changing the value of *Rectangle-length* (in steps 2 and 3) will cause the values of *Rectangle-area* and *Cuboid-volume* (in steps 4 and 5) to be re-evaluated.

For most traditional programming methods, an undefined variable is not allowed. For example, step 1 can cause an error of data type during compilation in most traditional programming languages [Ous98], if both variables *Rectangle-length* and *Rectangle-width* are undefined. In addition, according to the sequential algorithm provided by any of these methods, the definitions given in steps 2 and 3 do not change the value of *Rectangle-area*. In other words, dependency maintenance is not supported by these methods.

Moreover, the exploratory programming empowers the modeller to experiment with the computer model in order to enhance his/her understanding. Design is a trial-and-error learning activity [Som92, Vli93]. It is valuable to explore diverse situations in order to capture a deeper understanding of a problem and its solution through a variety of experiments. Giving a redefinition, that is, an experiment, the modeller can see – *experience* – a state change in the computer model [Bey94]. In the light of these immediately experienced state changes, the modeller can modify or qualify the virtual correspondence and, more significantly, reconstruct his/her understanding. In this respect, the theme of EM to a large extent accords with the concerns of constructivism[15]: knowing-by-doing, a very important concept widely used in education [Puf88]. The invocation of experiments on the computer model, as actions on the subject in a constructive model, is an interactive, situated mechanism whereby the understanding can be extended, expanded and experienced.

[15] Constructivism is characterised as "the continual restructuring of the relation between self and world, where world implies both palpable and ideational reality" [Puf88, p.17].

In summary, definitive programming is very helpful for the modeller seeking to enact EM in the form of situated activity. As described earlier, the openness of situated activity enables human agents to cope with varied situations by taking situated actions [Suc87]. In the same manner, it is necessary for the modeller to interact with the computer model in an open-ended manner. Moreover, in order to ease the limitations of human cognitive activities, it is helpful to use the computer as an artefact to improve understanding of a problem and its solution. With the aid of the interactive and exploratory features embedded in definitive programming, EM can serve these purposes in a significant way.

2.4 An Example illustrating EM

In order to illustrate the concepts of EM and the use of those tools mentioned in the previous section, the example of developing a hotel booking system is discussed here. To make a reservation, the customer must tell the hotel receptionist both the arrival date and the departure date. At the same time, the receptionist checks the availability of all room slots (12 rooms) in the reservation tables for those dates. If there is an available room, the receptionist puts the customer's name into the room slot for each day of the intended stay. According to the observation in the real world, two agents are identified by the modeller in response to his/her observation from the process of making a reservation:

- The customer who wants to make a reservation;
- The receptionist who is dealing with the customer's request.

Next, the modeller defines each of the agents in LSD. The LSD account of the customer and the receptionist agents could be as follows:

```
agent customer (c) {
  state customer_giveDate (c) /* the customer c intends to make a reservation */
  oracle reservation_ok (c, d1, d2) /* the reservation for the customer c during d1 and d2 is ok */
  handle customer_giveDate (c) /* the customer c provide dates of arrival day and departure day */
 protocol
     reservation_ok (c, d1, d2) == FALSE   => customer_giveDate (c) = TRUE;
     reservation_ok (c, d1, d2) == TRUE    => customer_giveDate (c) = FALSE;
}

agent receptionist {
  oracle customer_arrival_day (c, d1), customer_depart_day (c, d2), room_slot (n,d)
  handle
     customer_arrival_day (c, d1) /* customer c is expected to arrive at day d1 */
     customer_depart_day (c, d2) /* customer c is expected to depart at day d2 */
     room_slot (n, d) /* the content of room slot n at day d */
     reservation_ok (c, d1, d2)
  derivate
     customer_arrival_day (c, d1) = customer_giveDate (c) ? input(c, d1) : @
     customer_depart_day (c, d2) = customer_giveDate (c) ? input(c, d2) : @
     room_availability (n, d1, d2)
            = (∃d, d1<=d<d2, room_slot (n, d ) ! = "" ) ? "reserved" : "available"
  protocol
    customer_arrival_day (c, d1) != @ && customer_depart_day (c, d2) != @  &&
       reservation_ok (c, d1, d2) == @ && (∃n, room_availability (n, d1, d2) == "available")
         => reservation_ok (c, d1, d2) = TRUE; (room_slot (n, d) == c, ∀d, d1<=d<d2)
    customer_arrival_day (c, d1) != @ && customer_depart_day (c, d2) != @  &&
       reservation_ok (c, d1, d2) == @ && (∀n, room_availability (n, d1, d2) == "reserved")
         => reservation_ok (c, d1, d2) = FALSE
}
```

It should be noted that these definitions are personal and subject to revision during the enaction of EM.

Now, the modeller can create a computer model corresponding to the LSD account. A general rule is to replace derivates by definitions and protocols by actions. For example, the following Eden definitions are given:

```
all_rooms_availability is [roomAV_101, roomAV_102, roomAV_103, roomAV_104,
                           roomAV_201, roomAV_202, roomAV_203, roomAV_204,
                           roomAV_301, roomAV_302, roomAV_303, roomAV_304];
roomAV_101 is check_room_availability(1, d1, d2);
roomAV_102 is check_room_availability(2, d1, d2);
...
roomAV_304 is check_room_availability(3, d1, d2);
```

Also, visualisation using the observation tools is taken into account during the creation of the model. For example, a Scout screen corresponding to the reservation table is created as shown in Figure 2-3.

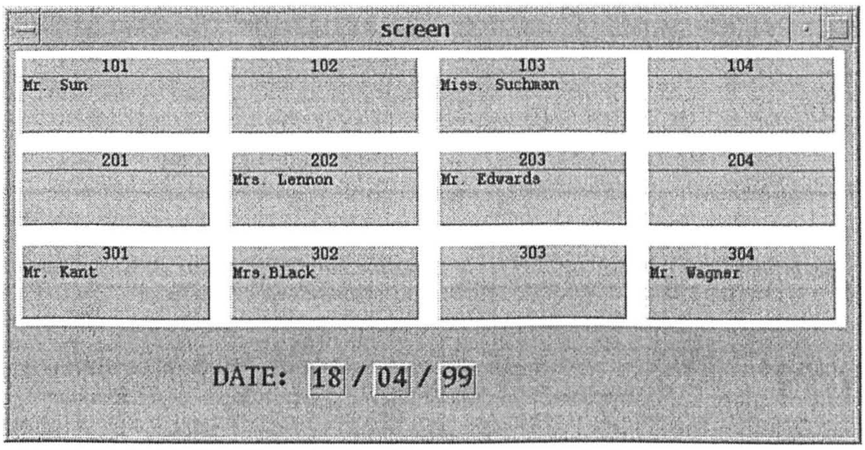


Figure 2-3. A snapshot of the computer model for the hotel booking system

To animate the process of making a reservation as described above, the modeller can select a date and a room slot and then enter the customer's name. After the interaction, it is found that data lists in the form of a reservation book used by the receptionist are needed in order to save the entered data. Hence, the following definitions are given[16]:

```
Year99 is [Jan99, Feb99, Mar99, Apr99, May99, Jun99, Jul99, Aug99, Oct99, Nov99, Dec99];
Jan99 is [day010199, day020199, ..., day310199];
Feb99 is [day010299, day020299, ..., day280299];
...
Dec99 is [day011299, day021299, ..., day311299];
day010199 is array(12);
...
day311299 is array(12);
```

Thus, the modeller can create procedures to store the customer's name in the data lists above when the name is entered into a room slot for a specific date.

Moreover, it is found that identifying the reservation table as an agent is helpful in introducing automatic checks on data integrity. For example, the agent's protocol can be configured to prevent the receptionist from mistaking the availability of a room slot when

[16] The current EM tools do not support the functionality of a database, so data manipulation is difficult. This so far is a limitation of EM (see Section 3.3). For the sake of simplification, the details of data manipulation are omitted here and in most examples given in this thesis.

making a reservation. An appropriate LSD account for the reservation table agent is defined as follows:

```
agent reservation table (d1, d2) {
  state
    room_slot (n,d)  /* the availability of room n at date d */
    room_availability (n, d1, d2)  /* the availability of room n during the period of d1 and d2 */
  handle reservation_error  /* the same room slot is allocated to different customers */
 derivate
   room_availability (n, d1, d2)
           = (∃d, d1<=d<d2, room_slot (n, d ) ! = "" ) ? "reserved" : "available"
 protocol
   room_slot (n, d) != c && reservation_ok (c, d1, d2) != TRUE && input_roomslot (n, d, c) == TRUE
                            => reservation_error = TRUE;
}
```

At the same time, for the LSD account of the receptionist agent, the definition of the observable *room_availability(n, d1, d2)* is removed and the following protocol is given in response to the new added agent.

```
handle input_roomslot(n, d, c) /* the receptionist intends to input the cusomer name c into room slot n for day d */
protocol
  reservation_error == TRUE   =>  input_roomslot (n, d, c) = FALSE;
```

The decision made by the modeller to identify a reservation table agent indicates that the modeller views making a reservation and the rules of making a reservation as conceptually distinct.

The modeller can continually use the computer model to explore, expand and experience his/her understanding of the intended system through 'what if' experiments for diverse purposes. For example, the following experiments have been conducted:

1. What if each of the room slots is coloured to indicate its availability?
2. What if the customer changes his/her arrival date?
3. What if the receptionist needs to reallocate one or more reserved rooms to make optimal use of the hotel's accommodation?

4. What if the customer asks for information about a room, such as price, bedding style, windows' orientation, and so on?

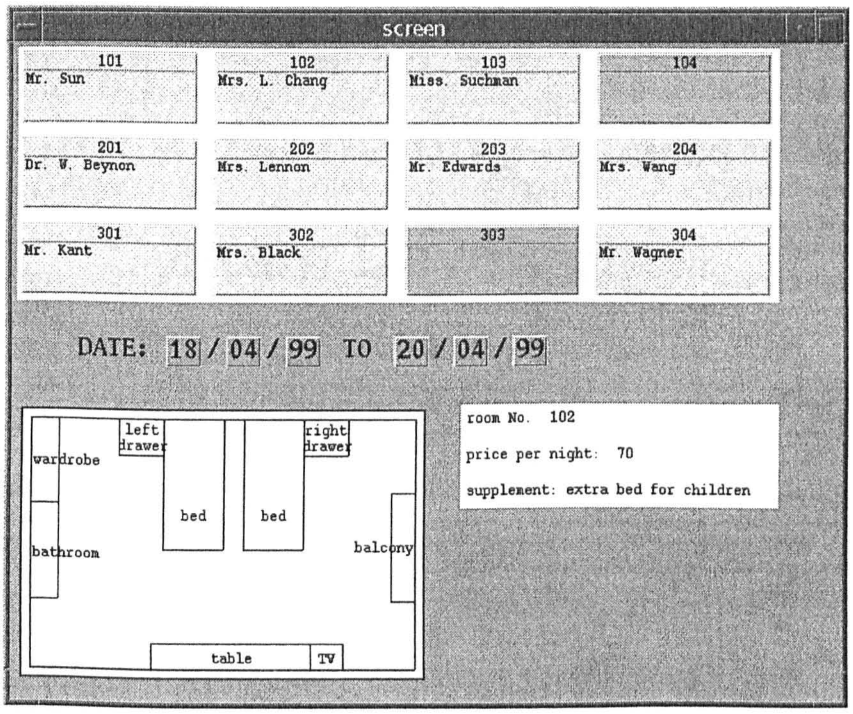
5. What if a reserved room slot is re-reserved for another customer?

Each experiment could change both the LSD account and the computer model. In the case of experiment 1, the following is added into the LSD account of the reservation table agent:

```
state room_slot_colour (n, d1, d2)
  derivate
    room_slot_colour (n, d1, d2) = (room_availability(n, d1, d2) = = "available" )? "grey" : "yellow"
```

In addition, the computer model is modified for invoking experiments 1 and 4. Figure 2-4 shows a snapshot of the revised computer model.

In the same manner, more experiments can be invoked by means of definitive programming for improving the understanding of the intended system. This understanding helps to maintain the virtual correspondence between the computer-based model and the system used by the receptionist in the real world. In most cases, the exploratory and interactive process of enacting EM must be stopped at some moment in order to deliver the developed system to the user. However, the openness of the developed system can still be persistent, and this in turn allows the user as the modeller to continue the enaction of EM during the use of the system in the real world. (More details are provided in the next chapter).



room No. 102

price per night: 70

supplement: extra bed for children

Figure 2-4. A snapshot of the computer model for a hotel booking system after further experiments

Chapter 3
Empirical Modelling and Software System Development

Software system development (SSD) is a process whereby human agents – including developers, users and other participants – construct and use a software system for their practices. The basic approach of traditional software engineering (SE) is broadly to formalise the development process abstractly without reference to the particular characteristics of the product to be developed. This promotes a conventional perspective on SSD where there is a clear separation between the abstract development process and the developed product (that is, the software system). To formalise this process, it is divided into several phases to be performed in a linear order[1]. Each phase is characterised by engineering practice, that is, it involves the application of proven methods, techniques and tools in a systematic and cost-effective fashion. Through such a well-defined phase-based process, it is expected that high-quality software can be produced with finite resources and to a predicted schedule. When a system being developed is well defined, its complexity is relatively low, and the overall project and technical risk are reasonably well understood, such a linear process meets this expectation. Unfortunately, these conditions are rarely achieved in the real world, and the worst thing is that changes in the real world,

[1] In a broader sense, a spiral model [Boe88] is also enacted in a linear order.

including people, information, technology and the process itself, are rapid and inevitable. Thus, a key challenge of SSD, especially for ill-defined and volatile software systems, is not to develop high-quality software product in a one-off shot but rather to adapt its process and its products to a rapidly changing environment [cf. Flo87, Fis93].

This thesis challenges the separation between the development process and the developed product associated with the conventional perspective on SSD. It proposes a radically different strategy for SSD in which the product-under-development is concretely represented by interactive artefacts throughout the development process. The presence of this concrete representation of the product-under-development leads to a development process that is no longer abstract, broadly preconceived and product-independent, but is situated, open-ended and product-specific in nature. For this reason, this thesis does not attempt to achieve greater understanding of the development process through formalisation and description in order to gain better control of SSD[2]. Instead, the development process in this thesis is viewed as a collection of situated activities that arise in the construction and use of the required software system in the real world. From this perspective, the investigation of SSD in this thesis is focused on the interactions between human agents, and between human agents and the product-under-development, as represented by interactive artefacts that reflect the evolving software system (cf. [Flo87, Gog97, Leh98]). Although this broader view of SSD is still in its infancy, the discussion about SSD in this chapter and other chapters (Chapter 4, 5 and 7) exhibits its potential to enhance the suitability and adaptability of the development process and the software system to be developed.

[2] Such attempts have led to the investigation of what are currently called *information systems development methodologies* [AF95, HKL95].

3.0 Overview

The purpose of this chapter is to highlight the powerful potential of using EM as an open development model for SSD. As explained in the previous chapter, EM is, in essence, a situated activity: an interactive and situated process without presumed sequential phases and rigid algorithms. In contrast to the 'linear thinking' (in Pressman's term [Pre97]) of the phase-based development process, EM entails 'experimental thinking' whereby iterative experiments are invoked to adapt the system to the changing environment in an interactive and exploratory fashion. This experimental process is suitable for developing open, ill-defined software systems, whilst linear thinking has difficulty in adapting systems to a rapidly changing environment.

Section 3.1 compares EM with a phase-based process model, with particular reference to the enacted process itself and its enactor. This comparison initially focuses on the fundamental principles of both EM and a phase-based process model. On the basis of traditional rationalism, a phase-based model is concerned with setting patterns of activities by applying engineering principles and concepts for guiding and managing the process of SSD. It is assumed that the enacted process can benefit from these formally-defined, well-structured activities, but this process also suffers from the difficulties of adaptation to the rapidly changing real world due to its adherence to these fixed rigid activity patterns. In contrast, EM concentrates on creating and communicating experience for the modeller through the process of constructing a computer-based model for informing the development of the software system [BCSW99]. Because of its situatedness and openness, EM activity is highly adaptable to the real world, though sometimes at the cost of efficiency and effectiveness.

This section also takes the human dimension of enacting a process model into account. Since the subjectivity of a human agent can influence the enacted process to a significant extent, most models are intended to minimise this influence in order to ensure the expected quality of the developed system. These models stipulate fixed patterns of activities with rigid algorithms that have paramount importance in guiding and controlling the enactor's behaviour. In contrast, EM is dominated by the modeller and his/her experience. No pattern of activities is given for guiding the process of EM, so that the modeller is able to undertake activities in a situated manner in order to cope with diverse situations arising from the process.

Section 3.2 explores the issue of knowledge manipulation in process models for SSD and in EM. Most traditional process models can be seen as emphasising either *knowledge representation* or *knowledge construction*. Process models oriented towards knowledge representation, such as the object-oriented model [Boo94], structured analysis and design [SS95, Pre97] and the entity-relation model [Che76], focus on the process of recording and communicating the knowledge of human agents (including the developer and the user). They seek to capture the knowledge in advance as completely and accurately as possible and to specify the captured knowledge by using context-free abstractions, for example through textual and diagrammatic metaphors. Process models oriented towards knowledge construction, in contrast, do not rely on the completeness and consistency of the knowledge that is represented in advance and separated from the context of the enacted process. These models, such as prototyping [Rei92] and Rapid Application Development (RAD) [Mar91], seek ways to construct knowledge with reference to the context, and hence focus on the process that enriches the knowledge of human agents in a situated manner. Attempting to take both knowledge representation and knowledge construction into account, EM aims to create a computer-based model to represent the modeller's knowledge associated with the intended system and to use situated modelling to enrich the knowledge. Graphical metaphors and dynamic interaction between the modeller and the computer-based model are exploited for the purpose of knowledge representation. More significantly, EM enables the modeller to facilitate knowledge construction in a situated manner by using the computer-based model as an interactive, open-ended artefact.

EM is in accord with new trends in process models for SSD, such as prototyping [Rei92, Mar91, And94] and scenario-based analysis [DF98, RSB98, SDV96, WPJH98], in seeking to improve the knowledge of the developer (and/or the user) rather than to prescribe the developer's behaviour [DF98]. Such an improvement can only be achieved to a limited extent, however, if the construction of the developer's knowledge and the construction of the represented knowledge in a representational medium are accomplished independently (cf. throwaway prototyping in [And94]). Instead, EM enriches the modeller's knowledge by 'what if' experiments resembling sensitivity analysis in a spreadsheet and at the same time interactively reconstructs the represented knowledge in response to changes in the modeller's knowledge. This enrichment by means of an interactive representational medium (that is, a computer-based model) gives EM the potential to enhance the developer's knowledge to an even higher degree.

Section 3.3 considers the use of EM as an *open development* model (ODM) for SSD. First, three kinds of software classified by M. M. Lehman are discussed: S, E and P-type [Leh94b]. Special attention is given to the E-type software, which is unbounded, ill-defined and liable to change in its operational domain in the real world. Lehman argues that the software itself is a model whose development and maintenance (that is, its *evolution*, to use his term) must be performed through feedback emerging from its operational domain. Unlike Lehman's feedback system, EM creates the software as a computer-based model in which not only feedback but also experience gained through experiments can be used as resources for the evolution of the software. Moreover, with the user of the developed software system in the role of the modeller, and with the aid of definitive programming, EM enables the software system to evolve in its operational domain. In this way, the problem of tacit knowledge can perhaps be resolved to a significant extent, and the gap between the developer's and the user's views of the system can be greatly narrowed. This section ends with a discussion of the limitations of EM as an ODM for SSD.

3.1 Open Development versus Closed World

SSD is a soft process referring to the entire life cycle of software production and evolution from the initial concept through definition, design, programming, implementation, operation, maintenance and enhancement, to the eventual retirement of the software [STM95]. In order to enact this complicated process, a flexible and practical process model is needed.

As already explained, most process models for SSD are established on the basis of linear algorithms derived from traditional rationalism [WF86]. The difficulties involved in enacting these models in the real world have been identified in [CS90, PR95, Rac97]. In contrast, EM treats the software system being developed as a computer-based model, and thus develops this software system by situated modelling (a form of situated activity, as introduced in Section 2.1). By means of this situated modelling, the knowledge associated with the system is incrementally embedded into the computer-based model through successive experiments and observation. When the modeller is satisfied that the knowledge embedded in the model is well-matched to the real world domain, the software system presented by the computer-based model is exactly the intended system.

It is very difficult to answer the question: is EM a process model? On the one hand, EM meets the demand that a process model can be enacted to construct a software process for SSD [Rol93]. On the other hand, unlike a process model, EM does not circumscribe the structure of the process by using rigorous algorithms. Apart from the principles and concepts described in Chapter 2, no instrument-like guideline is given for enacting EM. To clarify the difference between EM and most traditional process models, it is helpful to compare their essential foundations. This comparison is unfolded at two different levels: one is concerned with the process *per se* and the other is associated with the modeller, that is, the enactor of the process.

First, a useful starting-point is provided by P. Brödner's observation concerning two cultures in engineering:

> One position, ... the 'closed world' paradigm, suggests that all real-world phenomena, the properties and relations of its objects, can ultimately, and at least in principle, be transformed by human cognition into objectified, explicitly stated, propositional knowledge.
>
> The counterposition, ... the 'open development' paradigm, does not deny the fundamental human ability to form explicit, conceptual and propositional knowledge, but it contests the completeness of this knowledge. In contrast, it assumes the primary existence of practical experience, a body of tacit knowledge grown with a person's acting in the world. This can be transformed into explicit theoretical knowledge under specific circumstances and to a principally limited extent only ... Human interaction with the environment, thus, unfolds a dialectic of form and process through which practical experience is partly formalised and objectified as language, tools or machines (that is, form) the use of which, in turn, produces new experience (that is, process) as basis for further objectification. [Brö95]

Although this observation is concerned with the contrast between two cultures and their paradigms in engineering, the distinction may also be applied to models for SSD. The 'closed world' paradigm is characterised by the tradition of rationalism and logical empiricism that can be traced back to Plato. This tradition has been the mainstream of Western science and technology, and has demonstrated its merits in 'hard sciences'[3]. A

[3] T. Winograd and F. Flores define 'hard sciences' as "those that explain the operation of deterministic mechanisms whose principles can be captured in formal systems" [WF86, p. 14]. This thesis also uses this term in the same sense.

major influence on computer research into process modelling was Miller, Galanter and Pribram's famous book, *Plans and the Structure of Behaviour* [MGP60]. The authors examined everyday life and tried to represent it in a formal way. They proposed the concept of a Plan to explain their observation:

> A Plan is any hierarchical process in the organism that can control the order in which a sequence of operations is to be performed [MGP60, p.16].

This definition highlights the view that a Plan for everyday life is a process controlling the behaviour of both the human and the machine. They maintained that the behaviour of the human could be represented in a hierarchical structure as a program in a computer. This understanding obviously corresponds to the theme of the 'closed world' paradigm and has been widely accepted as the rationale of many process models in computer science. Also, it can be found in the simulation of human behaviour in the AI field [Agr95, Dre79, Hau97] and in the modelling of the software development process in the Software Engineering (SE) field [STM95, Som95, Pre97]. These models begin with the interpretation of a Plan as "a relatively fixed repertoire of commonly employed structure of action" [Agr95]. Then, they investigate the possibility of abstracting or formalising a process as a hierarchical plan for guiding the computer and the human as well.

A process that follows a plan in this sense should be predictable and repeatable. That is, by following the same plan, the actions in each process should generally be of the same nature and should lead to a similar final result [Jal97]. However, the continuously changing environment and uncertainty of human agents make the predictability and repeatability of the 'closed world' paradigm untenable[4]. This central problem becomes even more significant as more far-reaching research attempting to automate complicated

[4] The lack of repeatability has been used to criticise the view that SSD is an engineering discipline [DS97, Ste94, XIA98].

processes in the real world is undertaken. As argued by P. Feyerabend in [Fey75], a fixed plan decreases the freedom for taking actions and accordingly blocks the emergence of new concepts. He highlighted the disadvantages through an examination of historical episodes and an abstract analysis of the relations between idea and action [Fey75]:

> ... the principles of critical rationalism ... and the principles of logical empiricism ... give an inadequate account of the past development of science and are liable to hinder science in the future [Fey75, p.179].
>
> Modern science has developed mathematical structures which exceed anything that has existed so far in coherence and generality. But in order to achieve this miracle all the existing troubles had to be pushed into the *relation* between theory and fact, and had to be cancelled, by *ad hoc* approximations and by other procedures [Fey75, p.64, original emphasis].

Similarly, L. A. Suchman also illustrates the impotence of a Plan in coping with unexpected real-world situations by examining the interaction between the human and a photocopier with embedded instructions. She argues that such an attempt to abstract a process away from the particular environment in which it is situated is of limited applicability in the real world [Suc87]. J. A. Goguen also maintains that "rigidly following a fixed process model can severely limit adaptation" [Gog94]. Similar criticisms can be found in [Agr95, Dre79, Kir91, Rei65, RK95, Tul95, Weg97].

In accordance with the 'open development' paradigm, the above authors all address the inadequacy of setting fixed patterns of activities with rigid algorithms in guiding or even controlling human behaviour in the real world. To overcome this drawback, some researchers argued that the focus of a process model should be on the interaction between the human and the environment rather than on the activities of the human in a particular environment [Agr95, Bro87, Flo95, LR98, RB74, RK95]. They suggest that a process focusing on this kind of interaction should be able to involve as much improvisation as possible in coping with a wide variety of contingencies. In other words, SSD should not follow fixed activity patterns but instead be freely carried on by the interaction between the enactor and his/her environment[5].

In the same manner, EM rejects fixed activity patterns for SSD and centres on the interaction between the modeller and his/her environment (including the computer-based model and the referent in the real world). In each situation that is encountered, the modeller in EM is empowered to interact with his/her environment in an open-ended manner in order to maintain the virtual correspondence between the computer model and the referent in the real world as shown in Figure 2-1. Through such situated interaction, the computer-based model that is built up in the process of understanding the software system can come to fulfil the functionality of the required software system.

Apart from the openness and situatedness of the process itself, another key factor affecting the process of SSD is its enactor. Most process models pay less attention to this factor[6]. In EM, each state change of the enacted process is due to the invocation of an improvised interaction between the modeller and the computer model (or the referent in the real world) rather than the execution of prescribed activities. In this activity, no situation encountered in the enacted process is predictable in detail. The modeller has to advance the process by means of situated activities that construct the process of SSD. This human-centred concept is in harmony with the increasingly recognised fact that the human being is an important factor leading to the success of SSD [Pre97, LR98, You98].

In fact, as further examination discloses, when a process model is enacted, a complementary process that resides inside the mind of the modeller is simultaneously

[5] The enactor's environment could involve human agents, such as other developers and users. In addition, in a broader sense, both the developer and the user can be an enactor and affect the process of SSD (cf. participatory design in [Mum95]).

[6] Although some models, such as ETHICS [Mum95], have highlighted the importance of participants in the process of SSD, they are more concerned with formalising the process to be followed by participants rather than reflecting what participants do in their practices for SSD.

developed. To clarify this, the former, which changes the state of the environment (including the referent and the computer-based model), is called the *external* process, and

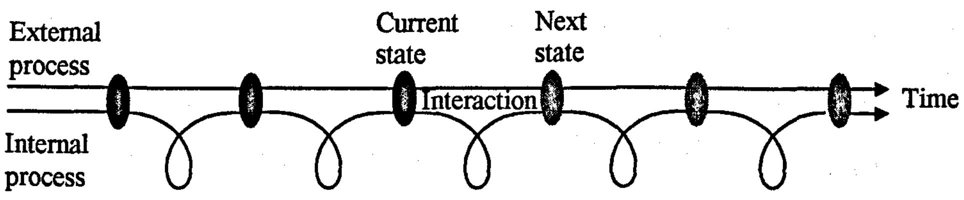


Figure 3-1. The interdependent and inseparable relationship between an internal process and an external process.

the latter, which affects the modeller's knowledge, is called the *internal* process. It is self-evident that both processes are intertwined and inseparable as illustrated in Figure 3-1. This close relationship highlights the fact that the modeller's knowledge (or experience) guides the external process in response to a situation in the real world, and the result from the external process in turn improves the modeller's knowledge through the internal process. Both processes affect each other, and this gives rise to changes in the modeller's mental model, the computer-based model and the referent in the real world.

Hence, the modeller's role in EM is indispensable in construing the phenomena occurring in the referent, in constructing the computer-based model, and, more significantly, in interacting with both the referent and the model in order to maintain their virtual correspondence. It is important to note that the emphasis on the human dimension should not lead to the formalisation of the modeller's behaviour, since this contradicts the principle of open development. Instead, the focus must be on the modeller *per se*. This shift is supported by J. Radford and A. Burton in their comments on simulating human behaviour: "if our aim is to simulate, and thereby gain more insight into, human behaviour, we should begin with the human rather than his behaviour" [RB74, p.349]. That is to say, the primary emphasis should be on the cognitive activity that underlies behaviour rather than on human behaviour itself.

3.2 Knowledge Construction versus Knowledge Representation

SSD is knowledge[7]-intensive [Rob99]. From conceptualisation, description and organisation to transmission, the enaction of a process model is concerned with the manipulation of the knowledge associated with the system being developed. Knowledge manipulation for SSD generally involves two processes: *knowledge construction*, which captures knowledge associated with the system for the developer, and *knowledge representation*, which records the developer's knowledge by means of representational media[8] such as documents and programs. Figure 3-2 depicts the relationship between the developer and these knowledge manipulation processes for SSD. It should be noted that these processes can operate in parallel and without synchronisation.

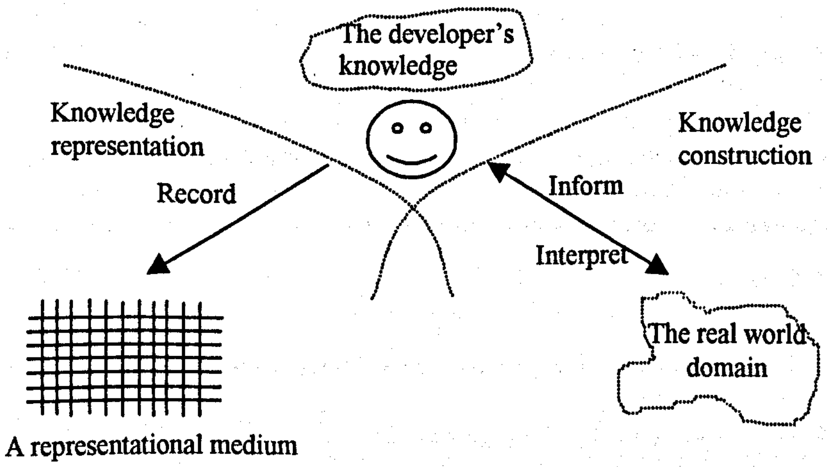


Figure 3-2. Knowledge representation and knowledge construction for the developer

[7] The term 'knowledge' is used loosely to include any structure of information which is constructed by coupling information obtained in one context with other information obtained in a different context [Pre97].

[8] The term 'representational medium' is used as in [Hut95], in a very general way, to indicate any form of media that can be interpreted as a representation of something.

Most traditional process models tend to concentrate on knowledge representation. They take it for granted that the developer can construct knowledge by successfully collecting and interpreting the information in the real world domain. They exploit systematised algorithm-based formats, for example in the form of an entity-relation model [Che76] or object model [Boo94], to represent the constructed knowledge. These representational media, that rely mainly on textual and diagrammatic metaphors[9], can specify the software system and its behaviour in a relatively context-free manner. If the specification fails to reflect the real-world context, this indicates that something in the real world domain is misunderstood. A reinforcing backtrack for locating and correcting errors should then be invoked. In this respect, the representational medium serves as a metaphorical presentation of the real world domain and representation of the modeller's knowledge associated with the system being developed.

However, in practice, there are debates about the completeness and consistency of the knowledge embedded in a representational medium. For example, on the basis of biological experiments, H. R. Maturana argues that symbolic representation cannot serve as the knowledge of an organism to control the way the organism behaves [Mat80]. He claims that an organism can adapt by coupling its structure with its external environment to generate its behaviour. That is to say, the knowledge that controls the behaviour of an organism is context-dependent and open to change. T. Winograd and F. Flores provide a similar argument for the design of an intelligent system that is restricted to representing knowledge by the acquisition and manipulation of the adopted facts. They remark that "knowledge is *always* the result of interpretation, which depends on the entire previous experience of the interpreter and on situatedness in a tradition" [WF86, p.74, original emphasis]. C. Crook also argues that "knowledge is not so neatly circumscribed as to

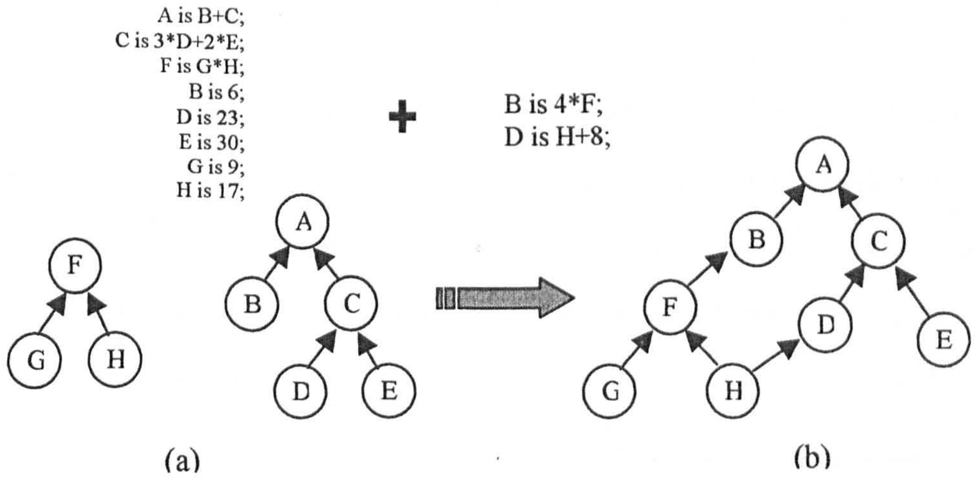
[9] In practice, metaphors in the form of texts and diagrams have been widely used in computer science for describing and sharing knowledge [Joh94]. Specification is a well-known example.

allow complete and unambiguous stuffing under some human lid" [Cro94, p. 95]. Similar arguments can be found in [Slo90, Cla97, Suc87]. The evidence of these researchers suggest that any attempt to give a full account of knowledge using representational media is inadequate.

Recognising the inadequacy of representing the developer's knowledge by context-free abstractions, some process models, such as prototyping [Rei92, And94] and the spiral model [Boe88] shift their focus from knowledge representation to knowledge construction, where the developer's knowledge is informed by its context in the real world domain. These models indicate that the developer must construct knowledge in a situated manner. From this perspective, knowledge evolves and is open to change. However, specification-based models can only support the openness and evolution of the developer's knowledge to a limited degree. In part, this is because documentation is mainly used for recording and is awkward to change. More importantly, it is difficult to describe context-dependent knowledge adequately by using text-based documentation. Hence, these models tend to contribute to the developer's implicit knowledge, based on practical experience, rather than to an explicit detailed specification. The concept of knowledge construction is consonant with A.diSessa's concept of *knowledge in pieces* (knowledge can only be constructed piece by piece) [diS88], and the theme of constructivism: *knowing-by-doing* (knowledge is gained through practical work) [Puf88].

Knowledge construction is useful for enriching the modeller's knowledge in a situated manner, and knowledge representation is helpful for organising the collected information into relations. Accordingly, EM regards the two approaches as complementary and seeks to take them both into account. To do this, EM uses the computer-based model as an interactive, open-ended artefact both to represent and also to enrich the modeller's knowledge in a significant way. For knowledge representation, it exploits graphical metaphors used in the visualisation of the computer-based model together with definitive scripts (described in Chapter 2). More importantly, the knowledge represented in the computer model can be explored through interaction with the model. The interactive exploration not only discloses the system's behaviour and the relationship between components, but also enables the modeller to connect knowledge with experience. This practical experience is more useful than the text- and diagram-based metaphors in other models for understanding the represented knowledge in the representational medium (that is, the computer model).

Moreover, the computer-based model in EM is more powerful than the text- and diagram-based metaphors in other models in dealing with changes in the represented knowledge. This is because any change is liable to have implication for the whole system, and the scale of these changes is likely to be reflected in revising the representational media (for instance, in editing a document). However, EM can deal with this problem in a significant way. In EM, any change to the model automatically and interactively leads to a structural change to the model that couples the update (that is, the added definitive script) with the current structure using dependency maintenance, as described in Section 2.4. For example, the structure of observables shown in Figure 3-3(a) is reconstructed to



```
A is B+C;
C is 3*D+2*E;
F is G*H;
B is 6;
D is 23;
E is 30;
G is 9;
H is 17;
```

```
B is 4*F;
D is H+8;
```

Figure 3-3. The situated structural coupling of observables

Figure 3-3(b) when a new definitive script is added. In other words, given the computer-based model, the modeller can revise the representational medium in an interactive manner. This is very hard to do in most traditional models.

In effect, this structural coupling of the computer-based model is very useful for supporting knowledge construction. For knowledge construction, the modeller's knowledge associated with the developing system continually changes in response to the emerging information from the real world domain. It is very difficult for most traditional representational media, such as documents and prototypes, to keep up with this rapid change (cf. the throwaway prototyping in [And94]). With the aid of structural coupling, EM can to a large extent support the interactive reconstruction of the computer-based model.

In addition, EM aids knowledge construction in a significant way. With reference to the openness and evolution of the modeller's knowledge, the structural coupling in EM, as described above, can provide the modeller with sufficient support. The focus here is on the enrichment of the modeller's knowledge through EM. Through 'what if' experiments resembling sensitivity analysis in a spreadsheet, the modeller can produce sufficient resources for the refinement of his/her knowledge (see Section 2.2.2). In this context of modelling a situation, the relationship between the computer-based model and the modeller in knowledge construction is exceedingly subtle. On the one hand, the modeller creates the computer model to represent his/her knowledge associated with the referent in the real world. On the other hand, corresponding to the interaction with the computer model, the modeller gains additional resources to (re)construct his/her new knowledge. In other words, both the represented knowledge that is embedded in the computer model and the modeller's knowledge are intertwined and complementary.

In practice, a benefit of knowledge construction is that it links the modeller's knowledge to experience rather than simply informs the modeller's knowledge from the context. This is highlighted in A. diSessa's discussion of science education. He stresses that the use of the computer for improving the student's learning involves building and integrating pieces of knowledge in order to achieve the best connection to experience [diS88]. Obviously, the enrichment of the modeller's knowledge by means of interacting with a computer-based model accords well with this constructivist outlook. It also meets the need, identified by Naur [Nau95], for incorporating experiential knowledge to complement the knowledge that is defined and processed by 'logic and rules'. Similar arguments can be found in [Bur91, Cro94, LW91, Rei97, Sal87].

The enrichment of the developer's knowledge through practical experience has increasingly attracted attention in SSD, as software systems have become bigger and more complicated. Prototyping [Rei92, Mar91, And94] and scenario analysis [DF98, RSB98, SDV96, WPJH98] are very popular illustrations of this theme. However, both approaches, by making use of static rather than interactive representational media, inevitably isolate the represented knowledge from the developer's knowledge. In contrast, EM enables the modeller to interact with the computer-based model in a situated manner. In this way, the computer-based model that serves as an open-ended artefact not only facilitates the construction and integration but also the exploration and extension of the modeller's knowledge. In turn, the enriched modeller's knowledge can to a large extent enhance the knowledge represented by the computer-based model. EM thus supports SSD that is guided by the progressively enriched knowledge of the developer. This meets the need, highlighted by J. Goguen in his discussion of requirements understanding [Gog96], for development techniques that harness rather than reject subjectivity.

3.3 EM as an Open Development Model for SSD

In [Leh94b, Leh97, LR98], M. Lehman identified three types of software system: S-type, E-type and P-type. S-type software has a well-defined domain which can be completely represented by a fixed specification. Correctness is absolutely needed for the specification and its implementation. On the other hand, for an E-type software system, its application domain is, of necessity, bounded to programs, but its operational domain in the real world is unbounded and keeps changing. Such a system cannot be developed completely and precisely, and thus correctness becomes meaningless [Leh94a]. Consequently, the criterion of E-type software acceptability becomes user satisfaction with each execution of this software system rather than absolute correctness to a fixed specification. Other details are summarised in Table 3-1.

S-type	• It is completely defined by a fixed specification. • When revision is required, it is viewed as a new specification, and the resulting program is viewed as a new program. • It needs absolute correctness with respect to the specification.
E-type	• It is a model of the application in its real world domain; as such, the solution system contains a model of itself. • It and its operational domain are conceptually unbounded and continually change. • Its consequences under execution are unpredictable. • Human involvement in the application process and its computerised model excludes precise and complete theories/models of domain and application properties, and makes the change in the process and the model unpredictable. • Its development process is an evolutionary process which requires interactions between many human-populated agencies involving a wide variety of knowledge, understanding, experience and authority.
P-type	• It is used to solve specific problems. • It is intermediate between the S- and E-types.

Table 3-1. The summarised features of S-, E- and P-type software [Leh94b]

Traditional process models, which emphasise correctness, are obviously well suited to developing a S-type software system. According to prescribed, fixed specifications, these process models seek to develop the right software (validation) and the software right (verification). They provide proven methods, techniques and tools for optimising the process of developing such a system in order to ensure the completeness and accuracy of the developed software. However, for an E-type software system, these models can only provide limited help, since its specification provides only a provisional description and is liable to change. In particular, human involvement makes such change much less predictable.

Hence, Lehman suggests that it is helpful to view an E-type software as "a model of the application, its participants (human and mechanical), the operational domain, and activities in that domain" [Leh98, p.41]. Such a model is incomplete, unbounded and easy to change. There is inevitable and irresistible pressure on this model for change on an increasingly extensive scale; hence, one must regard the evolution of this model (that is, its development and maintenance) as a system in the system-theoretic sense. Lehman argues that the system behaves as a self-stabilising feedback system in which feedback, as the most important resource for evolving the model, is derived from the operational domain of the software in the real world. Through feedback leading to corrective or adaptive changes to the E-type software, one can obtain a degree of intellectual control over the software's evolution, thereby mastering it and achieving its sustained improvement.

From this perspective, it is clear that EM is well-suited to be employed for the evolution of E-type software. For example, EM appreciates that a software system is unbounded and apt to change. No fixed, complete and precise specification is needed to guide the evolution of the software system. Instead, the evolution is driven, controlled and directed by the modeller. Also, any feedback emerging from its operational domain can be captured and incorporated into the software in an interactive, situated manner. In other words, EM offers the support for change management and human involvement that are important features for the evolution of E-type software highlighted in Table 3-1.

More importantly, EM does not simply *regard* an E-type software system as a model, but *creates* it as a computer-based model. This shift is very powerful in supporting Lehman's concern for evolving a software system as a feedback system. A seed model is created in the computer at the beginning in response to the modeller's initial understanding of the software system being evolved. Like a developer's initial understanding, this computer-based seed model is also incomplete, imprecise and liable to change. When feedback in the form of knowledge emerges from the operational domain of the software system, EM enables the modeller to incorporate the captured knowledge into the computer-based model (that is, the software) by definitive programming and structural coupling described in the last section. In this way, a feedback mechanism is provided by EM in an interactive, situated manner. More significantly, this mechanism leads to the evolution not only of the computer-based model but also of the software that is to be developed, which is presented in this model.

The computer-based model created by EM can also serve as an open-ended artefact for the modeller to facilitate the evolution of the software. Through 'what if' experiments, EM enables the modeller to explore, expand and experience his/her understanding of the software through situated modelling (see Section 2.2). As a result, this improved understanding guides the evolution of the software. It should be noted that a computer-based model is very different in character from a computer program. Whereas the latter fulfils a preconceived and specified function, the former can serve as an open-ended aid to conception and design [BCSW99].

A SSD model for open development, in P. Brödner's sense (see Section 3.1), must be able to tackle the continuous change to the software system in order to interactively manage the knowledge emerging from the process. In this regard, EM has a potential to cope with continuous change at least as effectively as the feedback system proposed by Lehman.

An open development model (ODM) must also support the interaction between an individual developer or user of the system and his/her external environment, and must allow him/her to first capture the practical experience emerging from the interaction and then embed this experience into the system to inform subsequent interaction. In particular, an ODM should allow the software system's user to guide the evolution of the software in response to his/her practical experience. If it fails to support this user-centred evolution, a model cannot be an ODM. This is because, whilst the developer must struggle to elicit the user's practical experience, the problem of tacit knowledge still exists. Unfortunately, most conventional process models, even the feedback system discussed above, take insufficient account of user-centred evolution. Within these models, the developer is still regarded as the only person who is empowered to shape the software system. This developer-centred bias is evident from the fact that most software systems are provided to the end-user in the form of execution codes. Performance considerations apart, the key reason is to prevent the end-user from modifying the system since it is assumed that the end-user is not competent to do so (more details are given in Section 4.4). On this account, the developer-centred models are too weak to be an ODM for SSD.

In contrast, EM, which regards a software system as a computer-based, open-ended model, is able to support the user-centred evolution by means of definitive programming. Like the modeller, the software's user, if sufficiently qualified, is empowered to embed his/her practical experience into the software system simply by introducing new fragments of definitive script as described in Chapter 2. In this way, the software system is open to change in an interactive, situated manner in response to the captured knowledge emerging from practical experience, even in its operational domain.

In summary: EM treats a software system as a computer-based model that – in the light of definitive programming – can evolve through modelling. Knowledge arising from practical experience, including that generates from the operational domain (through feedback, in Lehman's term) and by 'what if' experiments, is interactively incorporated into the system by situated structural coupling (see Section 3.2). The system evolves with the modeller's understanding, and is always liable to undergo further evolution. For this reason, it is plausible to say that EM serves as an ODM for SSD, in particular for E-type software systems.

Not surprisingly, EM, and its supporting tool tkeden in particular, has its limitations as an ODM for supporting SSD. Some of these limitations have been completely or partially overcome by the author's research, but the others still require further work. These limitations are summarised as follows.

1. EM does not support the interaction between multiple modellers.

In a sense, EM can be viewed as a modelling process for an individual, as described in section 2.2. The modeller is the unique user in the enaction of EM. In the same manner, the tool tkeden, supporting EM, is also developed for individual modelling. However, as highlighted by Lehman, the evolution of E-type software generally requires the interaction between many human agents [Leh98]. This limitation motivates one of the main research tasks in this thesis: to extend the framework of EM to a distributed environment.

2. EM provides no formalised method.

In general, EM is a sort of experience-based modelling technique. In order to free the modeller from rigid algorithms imposed on his/her activities, no formalised method for SSD, apart from some fundamental principles and concepts, is given in EM. On the one hand, the experienced modeller can benefit from the freedom to cope with diverse situations in practice. On the other hand, the naive modeller is often puzzled by the enaction of EM. This dilemma is concerned with a trade-off between the needs of both kinds of modellers. In fact, according to the discussion earlier (Section 3.1 and 3.2), it is very difficult to say whether or not this is really a limitation for SSD.

3. EM does not support project management and quality control.

The most important purposes for using a phase-based process model are to manage the project of SSD and promote the quality of the developed software system [Blu94a, Gib94, Som95]. EM takes no account of either purpose. In the software industry, this limitation might discourage many practitioners from using EM as a model to develop software systems, especially those software systems which have constraints, e.g. time and budget, even though EM provides the advantages of open development.

4. EM has difficulty in supporting a large-scale project.

An immediate cause of this limitation is the supporting tool tkeden. Since tkeden is the subject of ongoing research, insufficient account has yet been taken of its scaleability, and complex dependency between observables, e.g. higher-order dependency such as is discussed in [GYCBC96], is not supported. It is clear that both problems could be relieved if alternative advanced techniques were exploited. However, such relief would only be partial if the heavy load of modelling for a large-scale project is still attributed to only one modeller. It is clear that distributing the heavy load of modelling to many modellers (see Section 4.1), and improving the modelling technique, for example, by providing reusable definitive scripts (as proposed in Section 5.3), are useful ways to overcome this limitation.

5. Tkeden does not support component reusability.

Since no kind of modularity is enforced on the computer-based model in tkeden, reusable components are difficult to establish. Typically, definitive scripts must be given piece-by-piece, even though some pieces are very similar. For example, in the case of a hotel booking system (described in section 2.4), each room slot has almost the same description, but to a large extent the modeller cannot reuse a definitive script to generate the needed scripts. This limitation prevents the modeller from structuring the developing system and leads to an increase of program size. Hence, maintaining the developed system becomes much harder. The author's research in this thesis helps to address this limitation.

6. Tkeden does not offer powerful tools to support data manipulation.

The only data structure supported by tkeden is the list. For data-intensive software systems, the weak support for data manipulation leads to complications. For example, in the case of the hotel booking system, the reservation data for each room slot on each day is stored in a list. Any manipulation of these lists requires extra effort from both the modeller and the computer. Without the support of a powerful tool for data manipulation, e.g. through the use of a database, it is not easy to use tkeden for developing data-intensive software systems.

7. Tkeden provides limited support for user interface design.

User interface design has increasingly become one of the most important requirements for supporting SSD. Most modern programming languages, such as VB and Java, take this support for granted. In addition, since the interaction between the modeller and the computer-based model is the most important activity in enacting EM, support for user interface design can provide the modeller with useful benefits. Unfortunately, tkeden can only provide such support to a limited extent.

Chapter 4

Distributed Empirical Modelling

A prevalent view in modern industrial societies is that group activities are an economically necessary and efficient means of production, and that such societies could not survive on the basis of individual effort alone [Smi97]. Group activities are particularly important in developing information systems, since this task is bound to involve the co-operative effort of users, developers and designers [AC98]. Any approach for supporting this social process should therefore take group activities into account. On the other hand, though EM has promising potential for software system development (SSD) as discussed in Chapter 3, the practical work of EM so far has focused on the role of the individual[1] modeller. In a typical application, the modeller, in the role of the only external observer, builds up a computer model for SSD by modelling. The model often converges to a state that represents the cognitive insights of the modeller. In order to apply EM to SSD, convenient support for group activities, in particular group interaction, should be provided. To achieve this, it is necessary to clarify the distributed perspective on EM and enhance the framework of EM to create a distributed modelling environment where interpersonal interaction between modellers (such as exploring shared knowledge and shaping agency of agents) can be effectively carried out.

[1] The emphasis in EM until now has been on groups of people creating a single model, or creating individual models that are not closely integrated.

4.0 Overview

This chapter aims to establish a framework for distributed Empirical Modelling[2] (DEM) by drawing on two important theories in social science: *distributed cognition* [Hut95] and *ethnomethodology* [Gar67]. Two topics associated with shaping agency and developmental strategy for SSD within DEM are also discussed.

The first section seeks to explain why - from both a practical and theoretical point of view – the distributed perspective on EM needs to be highlighted. In practical terms, these factors are considered: alleviating the overloaded responsibility of the modeller in EM, adjusting a critical bias with reference to the balance of subjectivity and objectivity in EM, catching up with the trend towards techniques for developing distributed software systems and providing an alternative to the traditional communication between users and developers. In the theoretical analysis which follows, the concept of distributed cognition is introduced to explain that meanings of group work are distributed across all participants rather than only inside any individual [Hut95]. Next, the concept of ethnomethodology [Gar67] is elaborated. It highlights the inadequacy of investigating the interaction between members of society simply from the viewpoint of an external observer. Both concepts indicate that the modelling activities for developing multi-agent systems should be invoked not only from the system level but also from the component level. This fact motivates the construction of a framework for DEM.

In Section 4.2, a framework for DEM is established. An approach, called E-modelling and based on a familiar approach to concurrent engineering, is first considered. Although modellers can perform E-modelling activities from both the system and the component perspectives in a distributed environment, they act as *external* observers so

[2] The term 'DEM' is used in this thesis for highlighting the distributed perspective on EM. It should not be misunderstood as if it is different from EM.

that the application of EM principles becomes more obscure. This is because the context in which each modeller interacts with the computer model is separated from the context in which he/she interacts with other modellers. Another approach called I-modelling is then proposed to highlight not only the distributed perspective on EM but also the principles and concepts of EM. New modellers as internal observers, called A-modellers, are introduced to shape the agency of agents in agents' customary context. This is reached by 'pretend play' whereby modellers act as agents to interact with each other. At the same time, a modeller called the S-modeller is involved. The S-modeller as the super agent shifts his/her super-agency from shaping the agency of agents to creating the context for agent interaction. With the involvement of the S-modeller and A-modellers, the framework that integrates the cognitive processes of modellers with their social process is constructed for DEM. This section ends with a discussion of the collaborative relationship between modellers. The relationship between modellers is analogous to that of participants in Gruber and Sedl's shadow-box experiment, where observers and experimenter collaboratively work together to construct a consensus between them [Goo90].

Section 4.3 focuses on distinguishing DEM from AI and previous variants of EM where shaping agency is concerned. First of all, the popular definitions of *agent* and *agency* in the AI field are briefly reviewed and their literal meanings (based on dictionary definitions) are explained. Next, the definitions and features of agent and agency in DEM are discussed. Within DEM, it is convenient to adopt the convention that an observable becomes an agent as soon as the modeller attributes a particular state change to it, and reverts to an observable when its agency disappears. With this interpretation of agent and agency, DEM highlights the fact that shaping agency is a cognitive process but also a social process where modellers interact with each other by acting as agents. Through the combination of both social and cognitive processes, modellers can adapt the agency of agents to change practice and at the same time make it accountable to each other (in the sense introduced in ethnomethodology).

Three strategies for SSD are discussed in Section 4.4. The *intentional* strategy, proposed by Dennett in [Den87] and underlying most conventional design models, is the strategy most commonly used by the developer of a software system. Through this strategy, the intentionality of the developer is embedded in the behaviour of the developed system. To complement the intentional strategy, the *experimental* strategy whereby the developer explores the system behaviour through exploratory experimentation is introduced. When both strategies are combined, the developed system reflects not only the intentionality but also the experimental experience of the developer.

Intentional and experimental strategies put the central focus on the developer, which gives rise to a gap between the developer's system and the user's system. *Evolutionary* strategies for SSD have been proposed as a way to narrow the gap, but these are traditionally developer-centred. DEM attempts to bridge this gap in a more significant way by adopting an evolutionary strategy that involves users. It treats a software system as an evolving system whose behaviour is adapted to its rapidly changing environment in an interactive and situated manner. In this way, the task of system development is no longer an exclusive prerogative of developers. Instead, users, based on their professional background and contextual needs, are also amongst the most important 'developers' of their system as it operates in the real world.

4.1 The Need for DEM

Research into supporting a group activity in EM stems from the need to develop multi-agent systems [ABCY94, BR94, BNOS90]. In [Bey97], two scenarios where EM are applied to develop a multi-agent system are described:

- **Scenario 1** (called *S1-modelling*): the modelling activity is centred around *an external observer who can examine the system behaviour*, but has to identify the component agents and infer or construct profiles for their interaction.
- **Scenario 2** (called *S2-modelling*): *the system can be observed from the perspectives of its component agents*, but an objective viewpoint or mode of observation to account for the corporate effect of their interaction is to be identified.

Due to the lack of a suitable tool that can support S2-modelling in EM, the main focus of EM in previous research work and most case studies has been only on S1-modelling[3]. Although S1-modelling can be used alone for the development of multi-agent systems, S2-modelling – from both practical and theoretical perspectives – is more useful for developing such systems. Four reasons for this are discussed here: the need to decentralise the modelling activity, to redress individual bias, to develop distributed systems, and to support interpersonal interaction.

Decentralisation of the modelling activity: In practical terms, S1-modelling empowers one modeller as the super agent to step into the computer model in order to explore unknown territory. It is fair to say that after accomplishing the modelling task the

[3] A variant of S2-modelling is found in some case studies (such as in [ABCY94, BR94]). With this variant, each modeller at the lower level can concurrently describe the system behaviour from the perspectives of component agents by using LSD, and a high-level modeller (that is, the super agent) performs the modelling activity. Since the lower-level modellers do not enact EM in a typical fashion, this variant is not regarded as S2-modelling in this thesis.

modeller should become an expert in the specific domain. However, this also means that the modeller must be capable of deriving sensible meanings from the model. For example, in modelling a vehicle cruise control system [BR94], there could be a number of roles for the modeller to play for the purpose of interacting with the computer model. In the role of a car engineer, the modeller should be aware that a transducer on a wheel emits one pulse per revolution; the speed is measured by a counter/timer that estimates pulse-rate. At the same time, being in the role of a driver, the modeller should be aware of the relationship between the operation of setting the cruise speed and the output power of a car engine.

In other words, the modeller in S1-modelling is responsible for playing various roles from different viewpoints in order to promote the enaction of EM. The magnitude of this responsibility leads to overloading, with implications for cost-benefit and productivity, when the modelled subject is very complex and involves a variety of knowledge. To overcome this problem of overloading, it is necessary to decentralise the modelling task from the sole modeller to a team.

Redressing individual bias: Gruber and Sehl's shadow-box experiment reveals the bias of an individual's perception and the advantage of teamwork [Goo90]. This experiment highlights how easily different observers can have different perceptions of a single situation, as illustrated in Figure 4-1. It also indicates that the individual's perception is often informed only by a part of the whole subject. In such a situation, according to Gruber and Sehl's explanation, observers have to communicate co-operatively with each other to identify a mutually agreed possible object hidden in the box. They exchange tentative constructs – or *construals* – of their personal experience in order to make more sense of the hidden object. A social process that moves private perception toward public consensus in the light of interaction with others is then commenced for construing or re-constructing each observer's own experience. In short,

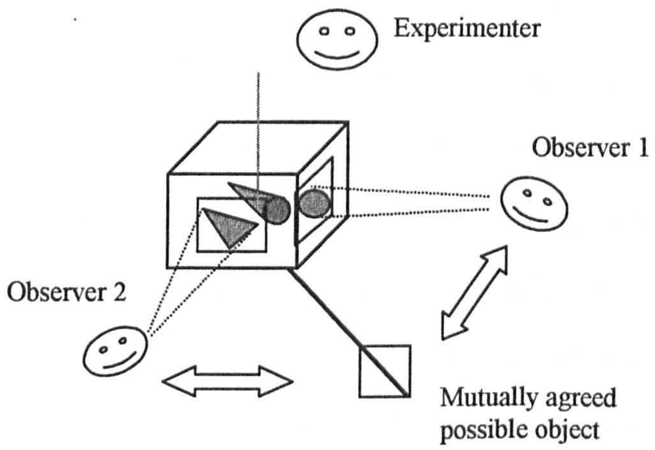


Figure 4-1. Gruber and Sehl's shadow-box experiment
(from [Goo90, p. 22])

this experiment highlights the fact that cognition based on the individual is very likely biased due to different situations and personal construals. To overcome this bias, it is necessary to integrate a social process with a cognitive process in order to construct a public, objective and universal construal between individuals. From this perspective, it is often not appropriate to model a multi-agent system by using S1-modelling alone.

The need to develop distributed systems: Distributed systems are increasingly popular in SSD. The splendour of centralised processing systems is fading away from the area of SSD because of up-to-date information technology and ubiquitous network communication. Instead, distributed systems that support a multi-user environment now appear to hold the spotlight. This new trend towards developing distributed software systems requires the support of a distributed modelling environment. In other words, if the intention is to apply EM to SSD, as explained in Section 3.3, then S2-modelling must be supported.

Support for interpersonal interaction: A very important motivation of highlighting S2-modelling comes from the research of D. Sonnenwald on 'communication in design' [Son93]. Sonnenwald pointed out that many design situations include a number of participants who need to interact with each other for building up an evolving artefact to support patterns of work activities, social groups and personal beliefs. Indeed, the importance of interpersonal interaction and of mutual understanding between all participants in SSD has been widely recognised [Bos89, Sal87, VF87, VPC98]. The development of a software system is not regarded as an act of individual creation but instead as the product of social interaction in a community [AC98, Flo87, XIA98]. Information arising from the interpersonal interaction between the user and the developer forms the foundation of SSD, and is therefore the key factor in determining the success or failure of a SSD project. For such interpersonal interaction for knowledge exploration and creation in SSD, it is obvious that S2-modelling can provide more support than S1-modelling.

At the same time, Sonnenwald's research also concludes that a collaborative tool is needed to support the mutual exploration of participants in order to facilitate interpersonal interaction and the creation of knowledge amongst participants. However, support for interpersonal interaction in SSD is limited and restricted to the use of traditional methods, for example, paper documents and conversational dialogue. Paper is passive and can only serve as a repository for information [Fis91, DS97]. Dialogue is liable to encounter those so called 'within', 'among' and 'between' communication obstacles[4] [VF87]. Although Sonnenwald gives few details about the nature of the collaborative tools that she has in mind, a computer-based system should be very promising in comparison with traditional methods[5]. This is because a computer-based system can be interactive and can facilitate human agents in searching, understanding and creating knowledge in a significant way [Cla97, Fis91, HM96, Rei97, FP88]. In this respect, with the features explained in Chapter 3, EM has proved its significant merits in improving an individual's

[4] The 'within' obstacles are those cognitive and behavioural limitations within the individual. The 'among' obstacles arise when participants have inconsistent or conflict viewpoints on the same thing. The 'between' obstacles are then mainly due to the fact that the involved parties speak two different languages.

[5] The computer-based interpersonal interaction should not be regarded as a replacement for conventional means of communication but as an important enhancement of existing means for communication.

understanding, creativity and learning [Bey97, Bey98, Nes97, Rus97, SB98]. In order to support knowledge exploration and creation between participants, the distributed perspective on EM needs to be clarified and highlighted in order to enable component agents to collaboratively interact with each other. In other words, it is necessary to provide support for S2-modelling.

Apart from these practical needs, an ontological debate that is associated with the use of S1-modelling and S2-modelling also indicates that S2-modelling is needed for developing multi-agent systems. With S1-modelling, the sole modeller as the super agent is empowered to interact with his/her computer-based model and the referent in the real world in order to explore, expand and experience his/her cognitive states. This modelling process is concerned with the construal of phenomena, empirical evidence and immediate experience, and is characterised by commitments based on the modeller's experience and knowledge [Bey98, Bey94, Rus97]. Whatever the system through which EM is enacted, the interaction of agents is viewed as a phenomenon experienced (or known) by the modeller and is construed in his/her particular context. Hence, the agency of agents (that is, the interaction between agents) is conceived in the privacy of the modeller's mind (cf. [BeyMSc, Bey98]). This inevitably amounts to a private, subjective and provisional interpretation of the modeller, that is, a first-person perspective.

In [Bey98], Beynon explains that this first-person perspective can be projected to the second- and third-person perspectives in order to build up common experience and public knowledge. The essential principle behind the projection of agency from first to second person is that "through experiment and perception of dependency I can identify families of observables (constituting a 'you') who can be construed as acting in ways congruent to my own" [Bey98]. And, from a third person perspective, an observable must be an element of "our experience that empirically appear to be common to all other human agents subject to what is deemed to be the norm" [Bey98]. Hence, Beynon argues that "objectivity is empirically shaped concurrently by our private experience, and our experience of other people's responses [in a common environment]" [Bey98]. S2-modelling is crucial if this process of projection of agency is to be empirically investigated.

Without S2-modelling, ontological problems obstruct any insight into the projection of agency from first to second and third person. The first-person perspective, that focuses only on private experience of the modeller as an external observer, cannot account for the interpersonal interaction between agents that involves aspects independent of private experience. The second-person perspective raises the issue of how an identified agent (constituting a 'you') – in particular if this 'you' is another person – can be deemed to gain experience and knowledge in the common environment that is congruent to the modeller's own. Similarly, the third-person perspective raises the issue of how interpersonal interaction that is inherently unpredictable can be construed as 'persons following standards or rules without referring to their own circumstances and contexts'.

Two important theories in social science are very helpful in clarifying the above issues: distributed cognition [Hut95] and ethnomethodology [Gar67]. The concept of *distributed cognition*, proposed by E. Hutchins in [Hut95], represents a synthesis of cognitive, anthropological and social scientific approaches to the study of collaborative work. Its central theme involves "locating cognitive activity in context, where context is not a fixed set of surrounding conditions but a wider dynamical process of which the cognition of an individual is only a part" [Hut95, p. xiii]. Hutchins argues that interpersonal interaction should be seen as an activity that is undertaken in social settings by using various kinds of tools rather than as a solitary mental activity. His concern is that overemphasis on representing or describing a 'knowledge structure' that is somewhere 'inside' the individual overlooks the fact that interpersonal interaction is always located in an intricate social world from which it cannot possibly escape. Hence, Hutchins emphasises that cognition for a group activity is a process socially distributed between individuals and artefacts, that is, "the group performing the cognitive task may have cognitive properties that differ from the cognitive properties of any individual" [Hut95, p.176]. Any analysis of such a group activity thus has to account for the interaction between participants. This account softens the traditional boundary between individuals and their environment, in which individuals are separated from their environment and at the same time each individual is isolated from others.

Ethnomethodology[6] is "the empirical investigation ('-ology') of the methods ('method-') people ('ethno-') use to make sense of and at the same time accomplish communication, decision making, reasonableness, and action in everyday life" [Rog83, p. 84]. In his major work, H. Garfinkel, the originator and guiding figure of this approach, indicated that ethnomethodology concerns "practical activities, practical circumstance, and practical sociological reasoning as topics of empirical study" [Gar67, p1]. He argued that social actors are, through their own actions, unavoidably engaged in producing and reproducing the intelligible characteristics of their own circumstances [Gar67, p23]. It is therefore not satisfactory to describe or interpret their circumstances by reference to rules or algorithms which are external to, or independent of, the ways in which its characteristics are recognised, used and produced through practical actions in a contingent manner. In this sense, Garfinkel affirmed that "social facts are the accomplishment of the members" [GS70, p.353, quoted in [Cou95]]. Social actors embody those rules of social reality in the process of their everyday social practices. In the process, their concrete activities give sense to their surrounding world and naturally exhibit the social competence that affiliates themselves with this society, allowing them to be recognised and accepted [Cou95]. In other words, ethnomethodologists do not want

[6] A very wide range of issues has been discussed in ethnomethodology, such as how the feeling and emotion of a human agent in everyday life affect his/her behaviour [AA87]. Only those aspects that are described in this paragraph are taken into account in this thesis.

to import any of their own assumptions of social facts as objects into their descriptions. By contrast, they make social facts observable and capable of being described and constituted through naturally organised ordinary activities. Through this process of reflexive constitution, social facts are accomplished in the ongoing process of the interaction between members. In addition to studies of social problems, such as education, medical practices and organisational processes, the concept of ethnomethodology has been applied to scientific research, for example, to the work of mathematicians on mathematical proofs [Liv87].

According to the concepts of distributed cognition and ethnomethodology, the projection of agency from first to second and third person hence has to take sufficient account of the social dimension of understanding and the commitment it entails. T. Winograd and F. Flores argues that it is very important to shift knowledge and understanding from an individual-centred conception to one that is socially based:

> Knowledge and understanding (in both the cognitive and linguistic senses) do not result from formal operations on mental representations of an objectively existing world. Rather, they arise from the individual's committed participation in mutually oriented patterns of behaviour that are embedded in a socially shared background of concerns, actions, and beliefs. This shift from an individual to a social perspective – from mental representation to patterned interaction – permits language and cognition to merge [WF86, p.78].

In other words, cognitive properties that are distributed across members should refer not only to each individual cognition (a cognitive process) but also, more importantly, to the interaction between all individuals and their environments (a social process) [Hut95, WFH96].

The ontological issues raised by the projection of agency, and the relationship between the S1-modelling and the S2-modelling perspectives, are familiar in sociology.

For instance, Trigg (cf. [Tri85]) refers to ‘a vexed issue in the social sciences’: whether there is something to be discovered at the social level or whether everything of significance can be dealt with at the level of the individual. The discussion of this sociological problem is beyond the scope of this thesis. The emphasis given here is on reconciling both global (that is, social) and local (that is, individual) perspectives on social interaction in a society [Tri85]. After all, most human-centred activities are so complex that both perspectives are intertwined and interdependent. In the same manner, it becomes apparent that S2-modelling and S1-modelling should be supported concurrently in order to reconcile the global and local perspectives on the development of a multi-agent system [Bey97].

The need to support both S1- and S2-modelling concurrently highlights distributed perspectives on EM that operate in two different dimensions, and are embedded in both kinds of modelling activity. In the ‘vertical’ dimension, individual modellers that perform modelling activities from the system level and the component agent level should be involved. In the ‘horizontal’ dimension, modellers that observe the system from the perspectives of different component agents are also needed.

4.2 A Framework for DEM

This section aims to construct a framework in which not only S1-modelling but also S2-modelling can be supported concurrently. A framework for distributed Empirical Modelling (DEM) that highlights the distributed perspective on EM (described in the last section) is established in the first subsection. Two methods for constructing this framework are considered and examined. In the second subsection, the collaborative interaction between participants is discussed. This is very important for SSD (relevant discussions are given in Section 4.3 and 4.4) and requirements development (discussed in Chapter 7).

4.2.1 Constructing the Framework of DEM

From the distributed perspective on EM, modellers for both S1- and S2-modelling activities at the system level and at the component agent level should be involved. At the system level, the modeller, called the *s-modeller* (the small s is used to denote its role as the super agent), is involved for enacting S1-modelling in order to examine the whole system behaviour (as shown in Figure 4-2). As explained in Section 2.2, the s-modeller

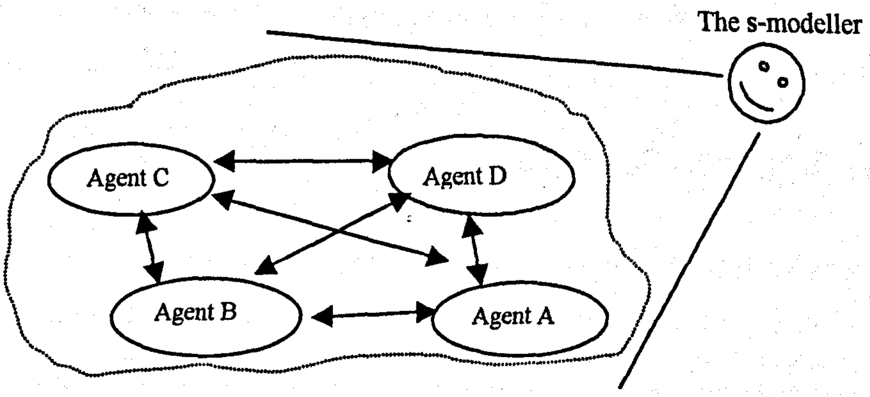


Figure 4-2 The relationship between the s-modeller and agents (the sole modeller in the system level)

has super agency to attribute the privilege of state change to agents and intervene in the interaction between agents.

In addition to the s-modeller as the super agent, modellers (called *a-modellers*) that can perform S2-modelling activity from the perspectives of component agents are also involved. Unlike S1-modelling, that is to a considerable extent understood, S2-modelling has thus not been explicitly discussed in previous work in EM[7], due to the lack of supporting tools. For instance, insufficient attention has been given to how a-modellers enact S2-modelling activity in a distributed environment in terms of observables, dependency, agents and agency.

To motivate DEM as introduced in this thesis, previous work on applying EM to S1- and S2-modelling is first discussed. This is based on an approach that is often used in concurrent engineering for decomposing a complex task to several small parts to be shared by a group people working together [DS94, Yeh92]. Within this approach, called *E-modelling*, each modeller is an *external* observer who observes the system as if from

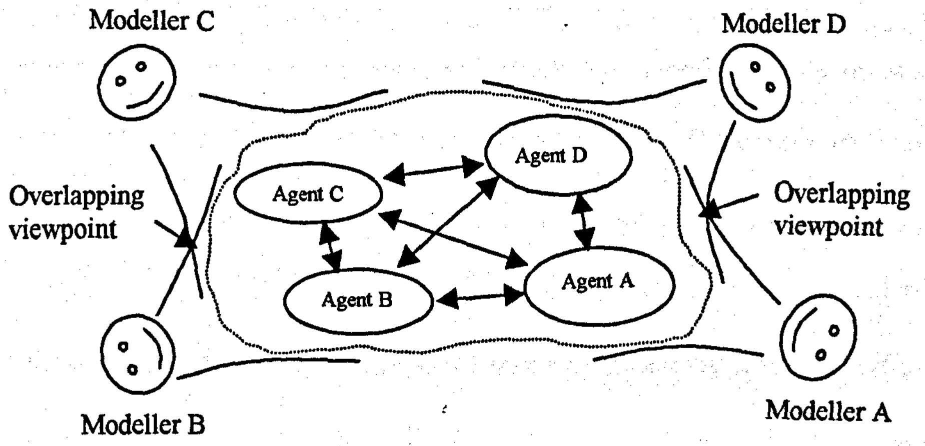


Figure 4-3 The relationship between a-modellers and agents (modellers in the component agent level)

[7] For example, it is not clear if all modellers use a shared computer model or each of them has his/her own computer model. For convenience, it is assumed in this thesis that each modeller has his/her own computer model and can enact EM in ways that are described in Section 2.2.

the perspective of a particular agent (as shown in Figure 4-3) and can enact EM individually (cf. [ABCY94, BR94, BNOS90]). For example, for developing a vehicle cruise control system (VCCS), a-modellers might be a dashboard designer and a car

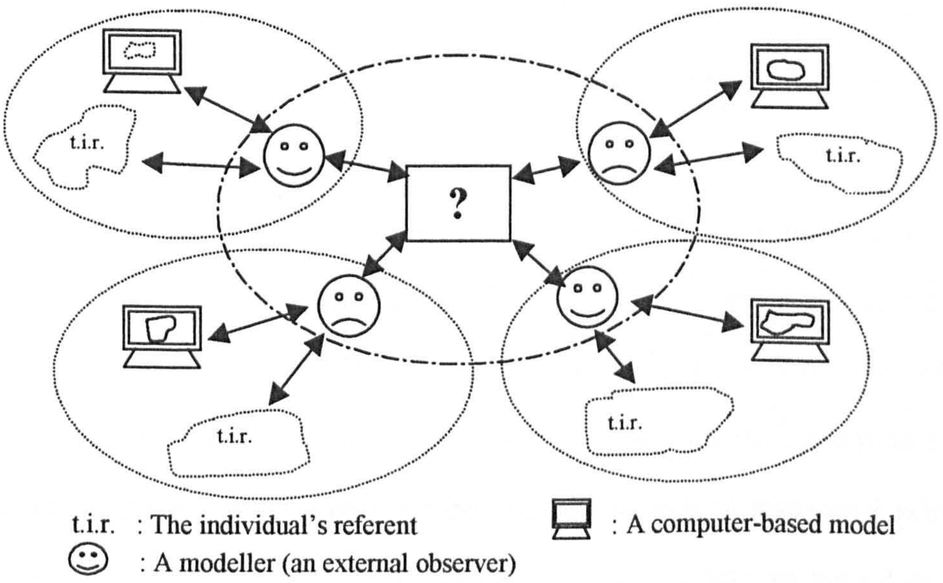

t.i.r.
t.i.r.
t.i.r.
t.i.r.
?
t.i.r. : The individual's referent
: A computer-based model
: A modeller (an external observer)

Figure 4-4. A framework for DEM based on E-modelling

engine designer who observe the system to take account of the perspectives of the car driver and the car engine respectively (cf. [BBY92]). This leads to a framework that supports – to some degree – both S1- and S2-modelling activities in a distributed environment (as illustrated in Figure 4-4).

Within the E-modelling framework, each modeller (including the s-modeller and a-modellers) can enact EM as an external observer by observing the system that is being developed from a particular context (that is, the s-modeller observes the whole system, but an a-modeller observes a part of the system). By enacting EM, each modeller interacts with his/her own computer model and his/her own referent in order to explore, expand and experience his/her own mental model. To enable these modellers to enact their modelling activities together, an agreement about naming and sharing common data, such as observables and definitions in EM, is often established. With this agreement, all modellers can interact with each other to shape their overlapping – but potentially conflicting or inconsistent – viewpoints, for example, through managerial negotiation or mediation. As a result, once the conflict between these computer models is eliminated through such social interaction, the intended multi-agent system could be obtained by integrating together all definitive scripts, for example, into the s-modeller's computer model[8].

Although this framework enables modellers to develop a multi-agent system by enacting both S1- and S2-modelling activities in a distributed environment, it does not provide modellers with sufficient support for their interaction. For example, inconsistencies or conflicts in shaping the agency of agents with the application are often eliminated through social activities between modellers, such as managerial negotiation and mediation. The involvement of many social interactions between modellers implies that the advantages of enacting EM will be largely reduced, since the context for modelling their computer models must be separated from the context in which they interact with each other. For example, modelling a situation through 'what if' experiments for the modeller from the component agent level will become very difficult and make little sense, since only a part of the system is taken into account. In other words, E-modelling activities do not accord well with the principles and the concepts of EM.

In addition, according to the concept of ethnomethodology, external observers find it difficult to understand the real context of an agent on the basis of their individual viewpoints. For example, the modeller for the driver in VCCS expects to have a digital

[8] In this framework, a communication network that connected the computer models of all modellers could be used to help information processing, for example, the collection and integration of definitive scripts from different computer models. From the perspective of modelling, the modelling activities between modellers are still conceptually independent. A broader use of a communication network will be given later.

speedometer, but the modeller for the speedometer designer in VCCS takes the normal type of speedometer into consideration. After mutual interaction, an agreement (for example, the need to build a digitalised speedometer) could be gained. However, when the driver actually interacts with this kind of speedometer, as when glancing at it while driving, he/she discovers that it is hard to read the displayed figure because of sunshine. Modellers who view the system as external observers can only make sense of the interaction between agents from their own context rather than from the contexts of the component agents themselves. After all, performing as an agent and being able to describe and analyse the methods used by an agent are not the same thing. Although external observers can change their roles in order to capture more information, each changed role as an external observer can only understand the interaction between agents by imposing his/her own context. Hence, from the perspective of ethnomethodology, it is not appropriate to account for the interaction between agents through E-modelling.

Accordingly, a framework based on E-modelling is not well suited for DEM, even though it can integrate S1- and S2-modelling together in a distributed environment for the development of multi-agent systems. A new approach (called *I-modelling*) that highlights not only the distributed perspective on EM but also the principles and concepts of EM is proposed by the author in this thesis (see Table 4-1 for a summary of the descriptions of S1-, S2-, I- and E- modelling). This approach introduces a distributed environment in which the roles of modellers in relation to component agents are both externalised and internalised. Through externalisation, the super-agency of an external observer in DEM is shifted from shaping the agency of agents to creating the context for agent interactions. Internalisation then enables modellers to improve their understanding by enacting the interaction between component agents in its natural context as internal observers.

In I-modelling, *internalisation* means shifting the role of a modeller from an external observer of the system inwards to that of an agent inside the system. By drawing on the concept of ethnomethodology [Gar67], new modellers at the agent level are introduced. Like modellers in E-modelling, these modellers are responsible for establishing the correspondences between their own computer models and referents to explore, expand and experience their mental models. But unlike modellers in E-modelling who are in the role of external observers, these modellers are placed in the role of agents within the application. In other words, each of these modellers, called an *A-modeller*[9], can act as an agent from the perspective of an *actor*, not only from the perspective of an observer.

S1-modelling	Modelling activity that is centred around the system behaviour	
S2-modelling	Modelling activity from the perspectives of component agents	
E-modelling	An approach to modelling a software system whereby modellers invoke S1 and S2 modelling activities by acting as external observers	
	s-modeller	The modeller who enacts S1-modelling
	a-modeller	A modeller who enacts S2-modelling
I-modelling	An approach proposed in this thesis that requires most modellers to model a software system by acting as internal observers	
	S-modeller	The modeller who creates diverse situations for A-modellers to reflect the different possible contexts for their interaction (this can be viewed as S1-modelling)
	A-modeller	A modeller who can act as an agent to shape the agency of that particular agent (in a sense, this activity can be regarded as S2-modelling)

Table 4-1 A summary of the descriptions of S1-, S2-, E- and I-modelling

As an actor, an A-modeller can pretend to be an agent within the application by enacting the ordinary interaction with other agents. This *pretend play* relies on an important belief in the ethnomethodology that each agent is capable of managing the world and of 'being-in-the-world' [Gar67]. Hence, A-modellers can either wait for

[9] The capital A is used to denote the role of the modeller in acting for an agent.

stimuli before they react or can autonomously trigger action to interact with others in their customary context, that is, from the viewpoint of agents themselves rather than from the viewpoint of the role of a particular external observer. In this way, A-modellers can get insight into the interaction between agents in the agents' context.

Through pretend play, A-modellers are regarded as *internal observers* rather than external observers, as shown in Figure 4-5. An external observer focuses on attributing state changes to a particular agent, but an internal observer pays more attention to appreciating the situations in which interactions between an agent and other agents occur. For an external observer, the interaction between agents is regarded as a form of state change of agents in the application system. The mechanism of stage change is often vindicated through observation and experiments from a specific viewpoint, e.g. an engine designer's viewpoint. Even though an external observer can 'experience' such an interaction by interacting with the computer model and/or the referent, this interaction is still based on a private and subjective perspective outside these agents, such as that of a designer or a user.

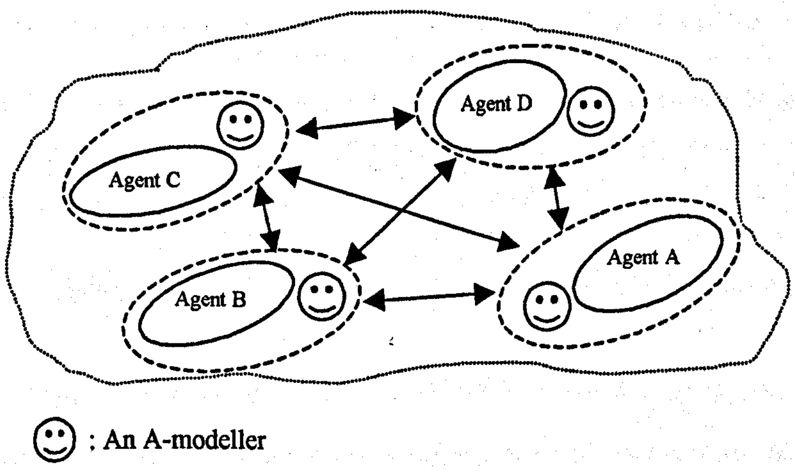


Figure 4-5 A-modellers acting as internal observers in a being-participant-observer way

In contrast, being an internal observer means interacting with other agents by acting as an agent. This kind of interaction that is exercised by internal observers has a strong contextual dependency, and can only be understood in a situation where agents are engaged in activities that they regularly and ordinarily perform. For example, in I-modelling for the VCCS, an A-modeller, who acts as the *car engine* agent rather than observing the system from the viewpoint of a car engine designer, can be introduced (cf. [BR94]). In comparison with a car engine designer acting as an A-modeller, such an A-modeller can gain much more insight into how the car engine interacts with other agents, such as the driver and the environment. By acting as a car engine, this A-modeller can wait for stimuli, such as those from the driver who starts the car engine or from its surrounding temperature, before responding. The A-modeller can also trigger action to interact with other agents, for example, by making itself break down due to unknown reasons. From the viewpoint of a car engine designer, these situations may not be important enough to be taken into account, but they are ordinary activities for a car engine. In particular, due to the uncertainty of incoming stimuli triggered by other agents, the context is usually situated beyond the imagination of a car engine designer.

This *being-participant-observer* way of acting as agents provides modellers themselves with important resources to account for their (agents') activities[10]. As the ethnomethodologist A. Coulon explained in [Cou95, p.23]: to describe a situation is to

[10] According to the way in which an observer participates in the activities of members, three membership roles have been proposed in [AA87]: peripheral, active and complete membership. The *peripheral* membership role allows the observer to participate in the activities of members without engaging in the most central activities. As an *active* member, the observer can participate in the core activities in much the same way as members, but he/she must hold back from committing themselves to the goals and values of members. With *complete* membership, the observer is expected to participate in the activities of members from the perspective of a full member. In I-modelling, A-modellers might adopt different membership roles that change over time for gaining different resources, but they become involved in complete membership roles for performing pretend play.

constitute it. Through this kind of reflexivity[11], the interaction between A-modellers can be conflated with the procedures that agents carry out in order to make their interaction accountable. In other words, by acting as agents to modelling situations, internal observers are able to describe the ways in which agents themselves make sense of their interaction and consequently constitute the reality of the interaction between agents.

From this perspective, the proposed interaction of agents through the enaction of EM in pretend play can enrich both the context of the agents' interaction and the mental model of the A-modeller. More significantly, when A-modellers interact with each other in the customary context of agents, they can observe, experience and account for the interaction between agents. This enables them to shape the agency of agents with reference to the interaction between A-modellers. This echoes W. Sharrock and G. Button's concept of 'acting as a member of a group to contribute positively to the creation of social interaction' [SB91].

The proposed pretend play raises a fundamental question: can an A-modeller, being a human agent, pretend to act as a non-human agent, such as a machine? An affirmative answer to this question is justified because the mechanism of a non-human agent is conceived in terms of human intentionality [Den87]. It is on this basis that the activities constituting the mechanism are prescribed, are well-defined and – more significantly – are made accountable to other agents. By appealing to such an account, it is natural for an A-modeller to engage in these activities in a mechanical manner. Even in the case of unpredictable activities, this accountability is still relevant. This is because it is plausible to attribute human uncertainty and unpredictability to a non-human agent. The attribution

[11] This is the feature of social order that "presupposes the conditions of its production and at the same time makes the act observable as an action of a recognisable sort" [Cou95, p.23]. It refers to "the equivalence between describing and producing interaction, between understanding and articulating the understanding" [Rog83, p.94]. For example, while people are talking, they are building up – at the same time that their words are uttered – the meaning of what they are saying.

of such failings to a non-human agent exactly highlights the possibility of its failure to confirm to the designated mechanism.

This way of attributing human-like mental features to machines is not a new concept. It has been broadly adopted in the development of AI systems [McC78, Sho93, WJ95, Rao94]. Most of them believe that to ascribe mental notions, such as beliefs, free will, intentions, abilities, autonomy, and so on, to a machine is legitimate and useful. More details will be given in the next section.

In practice, it is not necessary to involve as many A-modellers as there are agents when enacting their interaction. For instance, there may be no need to introduce an A-modeller for an agent whose interaction with others is entirely predictable. Such an agent can be temporarily regarded as a reliable machine that is responsible for specific state changes. In other words, the extent to which enaction by an A-modeller is appropriate depends largely on the character of the interaction between the agent to be enacted and other agents. An A-modeller can be introduced into or withdrawn from I-modelling at any moment in order to reflect the predictability of this agent's interaction with others.

In addition to the agents mentioned above, another very important 'agent' is taken into account in I-modelling. This agent is the rapidly changing environment of agents that is beyond the control of any agent. This omnipresent agent, that might for some reasons be regarded as 'God' [Fey75], can create contexts for agents in an open-ended manner in order to intervene in their interaction. This super intervention provides an effective way to express the uncertainty of agents themselves and their surroundings. By failing to take this into account, traditional rationalism restricted to rigid algorithms is hard to adapt itself to the emerging context in the *real* world [WF86, Suc87, Fit96].

For EM, super intervention is embedded in the modelling activities of the super agent, that is, the modeller. With this super-agency, the modeller can shape the agency of agents by directly attributing the privilege of state change to agents and performing 'what if' experiments. To enable A-modellers to shape the agency of agents in the agents' context, I-modelling *externalises* the relationship between the modeller (as the super agent) and agents by shifting the focus of the super-agency from shaping the agency to creating contexts for agent interaction. A modeller called the *S-modeller* is introduced in I-modelling. Even though the S-modeller still has the authority to perform the individual modelling task, more emphasis is placed on modelling the context of agents by creating diverse situations that can affect the interaction between A-modellers in the role of agents. In this way, the responsibility for modelling the agency of agents is largely shifted to the A-modellers, and this agency emerges from their interaction with each other.

The S-modeller, acting as the rapidly changing environment, can then create more various situations for A-modellers than A-modellers themselves can do. For example, a car engine may suddenly break down, or the driver may fall half-asleep. These situations, which are usually unexpected, enrich the context of the agents' interaction, and as a result improve the understanding of the S-modeller and A-modellers of the agency of agents. It should be noted that this improvement in I-modelling is reached by means of collaborative interaction between modellers (including A-modellers and the S-modeller) rather than the interaction between the S-modeller as the super agent and his/her individual computer-based model. More details of the collaborative interaction are given in the next subsection.

Network communication has an enabling role to play for I-modelling. The computer models of the S-modeller and A-modellers must be connected together in order to support the interaction between them in the form of pretend play. The society of agents is indivisible, though each agent may *just* be a part of this society. Each change triggered by an agent may cause effects on other agents so that the propagation of the change to others becomes very important. By using a communication network that connects all computer models together, this propagation can be achieved even though agents are separated in a distributed environment. With the aid of a communication network, a framework that integrates the S-modeller and A-modellers and supports S1- and S2-modelling concurrently in a distributed environment can hence be constructed. This framework (as illustrated in Figure 4-6) highlights not only the distributed perspective on EM (as described in the last section) but also the principles and the concepts of EM (as described in the Chapter 2). More technical details are discussed in the next chapter.

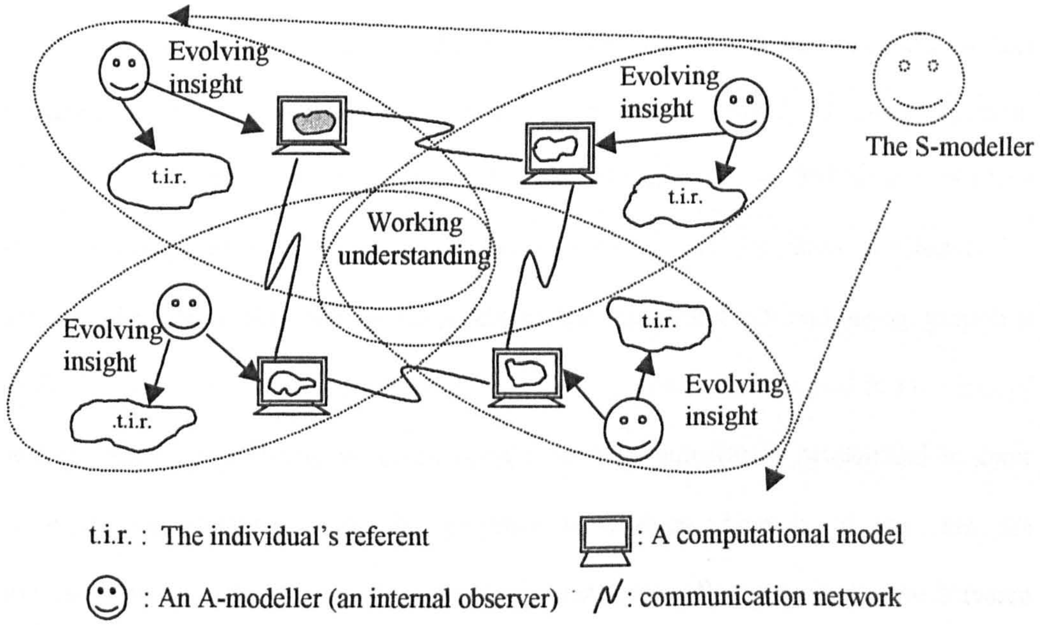


Figure 4-6 A framework for DEM based on I-modelling

The connection of modellers' computer models offers the additional advantage of improving the interpersonal interaction between modellers. The complexity and difficulty of interpersonal interaction in design involving participants from different groups have been revealed in [Son93]. Conversational dialogue on the basis of mental models is one of the easiest and most popular approaches to the interaction between human agents, but it does result in a number of interpersonal interaction problems, such as misunderstanding, confusion, and excessive time-consumption [Bos89, VPC98].

To overcome these problems, a lot of communication media, for example, formal notations, diagrams, and charts, have been devised to complement this social interaction in practice. However, most of these media, used as means of knowledge representation, are inactive and cannot support the social interaction efficiently enough to cope with dynamic change [DS97, Fis91]. Some computer models with static visualisation have been suggested for improving the method of human communication [LL94, LR91], but they can only provide limited help because they focus on knowledge representation.

In contrast to these models, the computer model with interactive visualisation has the potential to facilitate interpersonal interaction in an open-ended, interactive fashion. Models of this kind embodying individual construals are used in EM as a modelling medium [Rus97] and a knowledge construction medium (as described in Chapter 2). Their visualisation is obtained by representing the real-world referent using graphical metaphors. Since the internal representation of these graphics is expressed in the form of definitive scripts, any change in these scripts can be immediately propagated to their higher-level representation, viz. the graphics themselves. That is to say, via the communication network, the visualisation can interactively reflect the interaction between agents. Once a computer model is changed, this change can be propagated to other computer models, and this in turn will affect their modellers. By such interaction mediated by visualisation, the interpersonal interaction between modellers can potentially be improved[12]. In this respect, these computer models promote interpersonal interaction amongst modellers and serve as 'equipment for language' in the sense highlighted by T. Winograd and F. Flores [WF86, p.79].

[12] Due to the limitation of the author's research time and the lack of suitable projects, this improvement has not yet been evaluated in practice. An evaluation of the impact of computer-mediated interpersonal interaction would be helpful and is proposed as future work (see Section 8.3).

4.2.2 The Collaborative Relationship between Modellers in DEM

The proposed way of applying the ethnomethodological method to construct interaction between agents from an empirical environment is subtle. Clearly, the progress of this approach relies very much on the modellers involved. To illuminate the collaborative relationship between modellers, it is helpful to review the Gruber and Sehl shadow-box experiment described in the previous section.

Within this experiment, observers construe personal experience by simultaneously interacting with their own internal mental model and external environments in a way discussed earlier (Section 3.2). Both kinds of interactions are important resources for observers in making sense of the projection seen and the object hidden. Through both, observers can not only construct their own individual local knowledge but also contribute to the emergence of the distributed global knowledge amongst them. The former is associated with the projection seen and reports of others' construals. The latter, underlying the shared understanding of observers, is concerned with the agreed object that is compatible with the imaged objects of observers. It is not inside any observer's individual minds, but rather it emerges from distributed cognition across observers. That is to say, given the ability to interact with each other, observers collaboratively work together in order to make progress in constructing both the individual local knowledge and the distributed global knowledge.

In fact, another particularly vital participant in the shadow-box experiment is the experimenter, although he/she receives very little attention. It is the experimenter who sets up the context of experimentation for observers in order to examine the construals of observers. The experimenter is empowered to intervene with observers' contexts by altering the seen projection, for example, by changing the orientation of the hidden object. This intervention serves to provide observers with diverse resources to refine their construals, and this in turn leads to the reconstruction of both local and global knowledge mentioned above. In other words, the experimenter performs the role of a resources-for-knowledge-construction supplier and thus enables the observers to construct both kinds of knowledge. With the participation of the experimenter, the construction of both kinds of knowledge can be more effective and efficient, though it also depends on the experimenter's ability to create the context for observers. Therefore, the importance of the experimenter in this process of knowledge construction should not be underestimated.

On the other hand, the experimenter is also a constructor of knowledge, just like any observer. As described above, through the context created by the experimenter, observers are able to construct or reconstruct their knowledge and also to enrich their own contexts with surprising discoveries. This enriched context in turn provides the experimenter with useful resources with which to construct the individual global knowledge about this experiment. Theoretically, this global knowledge can include anything relating to the experiment, but usually its focus is on the experimenter's insight into the whole experiment. Its content could be very high-level and multifaceted. For example, the knowledge can concern how the new context affects observers' construals and what object, compatible with individual perceptions, is agreed by observers. These concerns are relevant to the experiment itself, but are also subject to the egocentric perspective of the experimenter.

Clearly, even though this global knowledge is always private and subject to the experiment, it indeed provides observers with situational contexts and as a result hopefully enriches both kinds of knowledge at the observer level, that is, the individual local knowledge and the distributed global knowledge. In most cases, the more situations that the experimenter creates, the more resources are provided to observers and the more feedback the experimenter obtains. This fact indicates the symbiotic relationship between the experimenter and the observers in constructing knowledge.

The way in which the A-modellers and the S-modeller work together collaboratively within DEM is much the same as the way in which the experimenter and the observers work together in the shadow-box experiment. Within DEM, A-modellers acting as agents can interact with each other to construct distributed cognition amongst them as described above. For example, an A-modeller can be introduced to act as a driver who interacts with both a brake and an accelerator to control the required power supply of a car engine. Given the visualisation of his/her computer model as shown in Figure 4-7,

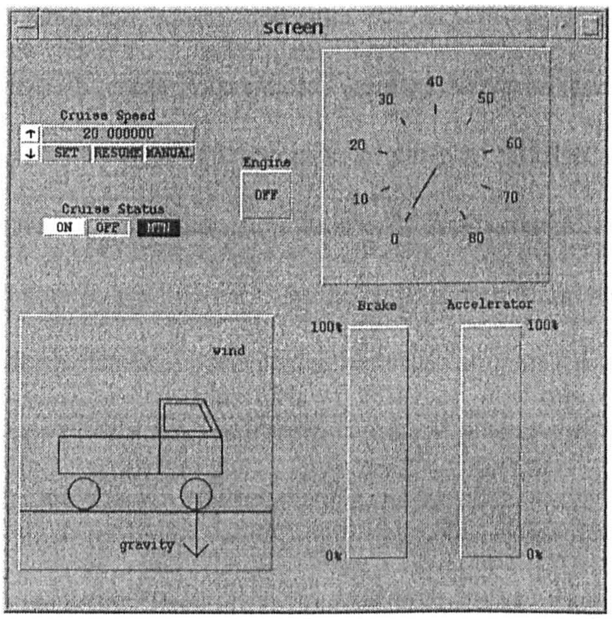


Figure 4-7. A visualisation of a vehicle cruise control system

the A-modeller can click on the box representing the accelerator to request the car engine to increase its power output. In this case, another A-modeller acting as a car engine could respond to this request in any chosen fashion, for example, by complying with the request, or making no response at all. In this way, each interaction between A-modellers becomes a constitutive part of the interaction between agents.

In other words, through interaction between modellers that takes the same form as that occurring in the ordinary life of agents, distributed cognition among A-modellers –

and also in the similar manner among agents – can be established reflexively. The difference between the observers in the shadow-box experiment and A-modellers is that the latter interact with each other through computer models connected together via a network rather than through linguistic dialogue[13] (the resulting implications for human communication have been explained in the previous section). Connecting computer models in this fashion makes individual contexts combine with each other to create a social context corresponding to Hutchins's model of what actually happens in a human society [Hut95].

At the same time, the S-modeller, like the experimenter in the shadow-box experiment, can intervene in the A-modellers' context to enrich not only his/her own global knowledge but also the local knowledge of each A-modeller and the distributed knowledge of A-modellers. The computer model of the S-modeller is established by integrating the computer models of all A-modellers. This integration is in accordance with the idea that the experimenter has a global view that embraces all components in the experiment. This view reflects the distributed cognition amongst agents, and describes how this is constituted through the interaction of A-modellers.

When the S-modeller intervenes in the computer model of any A-modeller, the new context can be propagated to other computer models through dependency maintenance and the network connection. This propagation can thus effect a change to these models and give rise to further interaction between A-modellers. As a result, the situational context of agents can be described and constituted through these interactions. The influences of the S-modeller in the situational context of agents should not be overlooked. It should be noted that the individual global knowledge of the S-modeller is not

[13] This should not be misunderstood as proposing that conventional communication between human agents can be completely replaced by the interaction between these connected computer models.

necessarily identical with the distributed knowledge of A-modellers, though both are closely interrelated.

In summary, the relationship between the S-modeller and the A-modellers in DEM is on par with that between the experimenter and the observers in the shadow-box experiment. On the one hand, all modellers construct their own individual knowledge by interacting with their own computer models. On the other hand, the distributed global knowledge is constructed by enacting practical interaction between modellers and by means of the network connection between their computer models. On the basis of distributed cognition, the knowledge is socially distributed across modellers. More importantly, this constructive process also describes and constitutes the social reality of agents by making their interaction observable and accountable. In this sense, this proposed framework, integrating cognitive and social processes, enables several modellers in a distributed environment to shape the agency of agents within an application.

4.3 Agency in AI, EM and DEM

Agent and *agency* are amongst the most important concepts in EM. As explained in Chapter 2, an agent can be an observable or a family of observables responsible for a particular state change. This definition is very different from the general usage of this term 'agent' in computer science, particularly in the Artificial Intelligence (AI) field. When several modellers are involved in a distributed environment, as in a real-life society, the definitions of agent and agency in EM furnish more practical and commonsense meanings.

The term 'agent' in computer science, particular in the AI field, generally refers to an entity which is granted or ascribed human-like mental qualities and is capable of interacting with its external environment [DBP93a DBP93b, KJ98, Rao94, Sho93, WJ95]. J. McCarthy argued that it is legitimate to ascribe human-like mentalistic notions to a machine when such an ascription expresses the same information about the machine that it expresses about a person [McC78, quoted in [Sho90]]. Similarly, D. Dennett has coined the term 'intentional system' to describe entities "whose behaviour can be predicted by the method of attributing belief, desires and rational acumen" [Den87, p.49]. These arguments underlie the fundamental concepts of agent and agency in the AI field.

As Wooldridge and Jennings discussed in [WJ95], descriptions of AI agents involve two kinds of notion: *action-based* and *mentalistic*. Action-based notions, such as autonomy, social ability, reactivity, and pro-activeness, refer to the ability of an agent to take actions. Mentalistic notions, such as goals, beliefs, plans, intentions, capabilities, knowledge, desires, and choices, refer to human-like concepts concerned with mental processes. Both kinds of notion are considered as the most critical factors affecting an agent's actions. Different agent models provide characterisations in terms of different notions. For example, Rao regards plans, beliefs, goals and intentions together as the mental states of an agent [Rao94], but Shoham takes beliefs, capabilities, choices and commitments into account [Sho93].

Although different mentalistic notions may be applied to an agent, it is generally agreed that given specifications prescribing their own notions directly control what an agent can do. In other words, these kinds of agents, here called AI-agents, are restrained from acting or causing effects within an unexpected context in order to make it possible to describe, explain and predict their behaviour in formal terms. From this standpoint, agency is interpreted as the capability attributed to an agent as specified by prescriptive mechanisms [WJ95, LdI95], and is granted by designers in ways that separate it from its context. Such agency is expressed through activity that, like reasoning in formal logic or a search for anticipated facts, is determined by a context that, though yet to be specified completely, is of a preconceived type.

An emphasis on knowledge prescription that does not take sufficient account of context is not without its problems. M. Minsky remarked that "formal logic is a technical tool for discussing either *everything that can be deduced from some data* or *whether a certain consequence can be so deduced*; it cannot discuss at all what *ought* to be deduced under ordinary circumstances." [Min74, p. 141, original emphasis]. In [Dre79], H. Dreyfus argued that what ought to be deduced can never be represented in the form of elementary context-free features characterising any type of content and structure of knowledge needed for this deduction. He also claimed that anticipating facts about a subject's total knowledge is an infinite task which must be situated rather than prescribed in a way of presupposing a background of cultural practices and institutions.

Indeed, it is increasingly recognised that knowledge prescription (or representation) has run into serious trouble in seeking to develop an intelligent machine that can act as a human being capable of adapting to its environment [Agr95, Cla97, WF86]. The context-free feature simply indicates the flaw of having far too little adaptability to accommodate frequently changeable practices. In this respect, the agency of an AI agent is rigidly limited to predictable description and cannot easily be adapted to a changeable environment.

Because of this lack of adaptation, the definitions of the terms 'agent' and 'agency' from AI field are not suitable for the framework of DEM. To give a suitable definition for both terms in DEM and EM, it is appropriate to review the following literal, dictionary definitions [Oxf89, Web61]. An 'agent' is defined as:

- One who (or that which) acts or exerts power, as distinguished from the patient, as distinguished from the instrument.
- A means or instrument by which a guiding intelligence achieves a result.
- Science: Any natural force acting upon matter, any substance the presence of which products phenomena.
- Of persons: One who acts for another; one entrusted with the business of another; a substitute; a deputy; a factor.
- Of things: The material cause or instrumentality whereby effects are produced; but implying a rational employer or contriver.
- The part of the system that performs information preparation and exchange on behalf of another. Especially in the phrase 'intelligent agent' it implies some kind of automatic process ... to perform some collective task on behalf of one or more humans.

The term 'agency' is defined in the following ways:

- The faculty of an agent or of acting; active working or operation; action, activity.
- Working as a means to an end; instrumentality, intermediation.
- Action or instrumentality embodied or personified as concrete existence.
- The office or function of an agent or factor.

These definitions indicate that an agent can be physical or abstract in nature, but it must be able to act or cause effects as granted agency by, and for, others or itself. From this point of view, an entity is an agent because of its agency. For example, the corner of a table can be an agent when it hurts someone's fingers or is used to open a bottle of wine. In the similar manner, a book can also be an agent when it changes the mind of the readers, increases the author's reputation or is used to knock on a door. These examples show that the agency of an entity cannot be specified with reference to its intrinsic features, since it is so widely open and undetermined.

In this sense, that an entity becomes an agent is due to the attribution of particular activities or causing effects. It does not matter whether it is active or passive, or whether or not it is able to perform these attributed tasks autonomously. At the same time, the consequence of acting or causing effects need not be guaranteed, since it depends highly on the combined context established by the agent's own status and the prevailing situation in its environment. In other words, anything is eligible to be an agent if its potential agency is granted (cf. the concept of view 1 agent in [Bey97]).

From this viewpoint, agent and agency are associated with situated activities rather than activities prescribed by rigid algorithms. For an AI-agent, the agency representing its capabilities is determined in a prescribed manner. As previously explained, this circumscribed specification is not sufficiently adaptable to a rapidly changing environment. By contrast, an alternative definition emphasising the aforementioned situatedness of agents is given in EM: *an agent is an observable or a family of observables responsible for particular state changes*[14].

Given this definition, the boundary between an agent and an observable hinges on the responsibility of the former for particular state changes. The responsibility attributed by the external observer to an agent has no intrinsic relation to either the agent's capability or its external environment. Neither is it to be interpreted as predicting the behaviour of the agent and the system. Instead, it is seen to refer to the modeller's experience and expectation. As long as the responsibility for particular state changes is projected, every observable/family of observables is immediately viewed as an agent.

For example, the handle on my door is an observable. It becomes an agent as soon as I turn it down to open my door since it is now responsible for changing the door's state. It does not matter that the handle of the door which I turn down is incapable of

[14] Without specification, the term 'agent' is used in the thesis in this sense. In addition, state changes are meant to refer to a change in the values and/or definitions of observables.

bringing about this attributed state change with my assistance; nor does it matter what the door's state is, such as whether it is locked or unlocked. Only if this responsibility for state change disappears, such as if I stop turning, will the handle of my door revert to an observable.

At the same time, no certain outcome is ensured for the attributed agency, even though this agent is responsible for the attributed state change. An agent engages with its responsibility according to its internal status, as defined, for example, by its abilities and intentionality, and its external situation, as defined, for example, by other agents' status. Uncertainty within both makes the result to a certain extent unpredictable.

Agent and agency defined in this way are strongly associated with the context of the external observer and the observed real world. The attribution of responsibility for particular state changes is not persistent nor are its execution and consequence guaranteed. In fact, all of these conditions can be altered at any moment, that is, they are situated and unpredictable. The 'protocol' part in the LSD account is used to convey this situated concept. In addition, an agent can be viewed as privileged to instigate change to some observables and dependencies. The modeller in EM is called a 'superagent', since no other modeller is privileged to change *all* observables and dependencies.

Different principles are used in shaping agency in EM and DEM. The shaping of agency in EM is accomplished by the interaction of the modeller with the computer model and ideally the referent. As described in section 2.2, the modeller, as the superagent, is able to change all observables and dependencies on behalf of other agents in a computer model. Each agent reliably performs the actions specified by the superagent (either explicitly or by automatic procedures) irrespective of its autonomy[15]. By means of repeated observation and experimentation, the agency of agents can be gradually shaped

[15] The supporting tool tkeden does not take into account each agent's context, including its own status and its environment situation. This account of reliable agents is acceptable in a stand-alone modelling environment because the context of each agent in this case is prescribed only by the super-agent.

in an open-ended fashion. This approach to shaping agency leads to a model that reflects the modeller's subjective, private and provisional perspective, and is an effective way to construct a computer-based model in EM that captures the modeller's construal [ABCY94]. It is also suited to the development of a computer model whose complex interactions are based on an aggregation of competing primitive stimulus-response mechanisms, resembling M. Minsky's model of the human mind as "a society of ever-smaller agents that are themselves mindless" [Min88].

Few agent models in AI take this situatedness of agents into account. Most conceive an agent as entity with innate properties, such as autonomy, reactivity, and social ability that are typically fixed and persistent. This association of properties with an entity takes no explicit account of the situated interactions between the agent and its external environment. Hence, the agency of an agent is determined by a set of essentially invariant properties of the agent that does not take account of its situatedness.

An exception to this norm is the agent model proposed by M. Luck and M. d'Inverno in [LdI95]. Within their model, an agent is instantiated from, and in turn reverts to, an object at some points in time. In contrast to the tradition of viewing agents as specialised objects, they argue that agency is transient for serving given goals. Without serving these goals, an object cannot be an agent. To some extent, this concept is similar to the situatedness of agents proposed in this thesis, but its focus lies mainly on formalising the appropriate agency of an agent in order to act for the intended goals. In this way, agency is defined in advance according to the intended goals instead of being shaped by repeated observations and experimentations, as in EM. In addition, the situatedness of agents described here should not be confused with the 'situated agents' concept proposed by Rosenschein and Kaelbling in [RK95]. The latter means that agents interact with their environment in a situated way, that is, to respond appropriately to diverse situations, as identified by prescribed rules.

The concepts of agent and agency in EM are adopted with a slight extension in order to apply to a distributed framework of DEM. An agent within DEM is not simply a mindless component of the computer model, which is reliably responsible for the state change given by the external observer to reflect his/her understanding of the agent's agency. Instead, by means of the enaction of the A-modellers, each agent can interact with other agents as if it is in its customary context. Corresponding to its current situation, each agent is engaged in situated actions accountable to others. The actions in turn reflexively create the reality of the interaction between agents. In other words, the agency of each agent, that is, the responsibility for state changes, is shaped in the process of creating the reality of agents' interaction rather than only in the individual cognitive process of the external observer in EM.

The shaping of agency has to be achieved by performing a cognitive process and a social process in parallel. After building up the individual computer models to correspond to the referent in the real world, as in EM, A-modellers as external observers shape the agency of individual agents in a cognitive process of experiential learning through repeated observations and experiments. In this way, the agency of an agent is simply localised in the mind of an A-modeller who is modelling this agent. Since an agent is only a part of a system consisting of many interlocking, interacting and mutually dependent components, its agency should be associated with those of other agents. In other words, the agency must be made accountable to other agents for the purpose of integrating with other parts.

In order to reach this point, a social process to do with construing the interaction between agents and constructing a working understanding among them is invoked. Each A-modeller acts as an agent by doing what the agent does in its practices in order to give sense to other A-modellers acting as other agents. Also, it is expected by the A-modeller that other A-modellers can do whatever else is accountable in their enacted agents' context. In this being-participant-observer way (cf. [AA87]), A-modellers adapting to the observed practice and to the reaction of others are led to shape their agency in a reflexively constitutive fashion. At the same time, the S-modeller acting as the rapidly changing environment also becomes involved in the social process to intervene with the interaction between A-modellers from the global view. As described in the last section, all modellers can collaboratively interact with each other in order to contribute to the embodiment and enrichment of the shaped agency of agents. The collaborative interaction for agency can be achieved through the incorporation of the following three modes:

1. without any interaction with others, through an individual cognitive activity of the A-modeller self, in which the interaction with the local computer model is invoked in order to maintain the virtual correspondence between the model and the referent (see Section 2.2);
2. through interaction with other A-modellers via the connection of all computer models, in which A-modellers act as agents in their (agents') ordinary context in order to reflexively constitute the agency of agents (see Section 4.2);
3. through the intervention of the S-modeller via a network connection with the computer models of agents, in which the new context triggers adaptive responses in the interaction between A-modellers (see Section 4.2).

As a result, the agency of agents is not only in the mind of the S-modeller, but is also socially distributed across A-modellers, as described above.

In practice, the agency of agents is reflexively constituted by a large number of local interactions and adjustments in practice. Many of these adjustments appear to reflect the common insight distributed across A-modellers emerging from two processes, that is, the social process and the cognitive process. To the extent that the exploration of agency adapting to a changing environment counts as learning, it may be said that shaping agency in DEM is an instance of collaborative learning that incorporates the experiential learning highlighted in EM.

4.4 Design and Evolution for SSD

There is an important and interesting difference between the processes of shaping agency in AI, EM and DEM discussed in the previous section. The first two attempt to shape agency through supervisory intention and intervention granted to the external observer, but DEM shapes agency through the process of evolutionary interaction between modellers for adaptation. This difference strongly resembles the difference between design and evolution for software system development (SSD). This merits further discussion here.

Design[16] for software development is an intelligence-intensive effort to build a software system and make it work. Its purpose is to create a software system whose behaviour is consistent with what the developer and ideally the user want to achieve. A key concern in this design activity is to predict the behaviour of the system to be developed under various scenarios of use. In this connection, the *predictive strategy* – in the sense introduced by D. Dennett in [Den87] – that is adopted by the developer is centrally important. In traditional SSD, system behaviour is typically predicted via an *intentional strategy* which "consists of treating the object whose behaviour you want to predict as a rational agent with beliefs and desires and other mental states exhibiting what Brentano and others call *intentionality*" [Den87, p. 15, original emphasis][17]. The concept of intentionality applies to all kinds of mental acts, for example believing, imagining,

[16] Here the general meaning of this term 'design' is used for the synthesis of conceptual design, physical design (or implementation) and testing.

[17]This predictive strategy is only made explicit in certain agent-oriented programming development approaches [Rao94, Sho90, WJ95], but is sufficiently general to account for most approaches to SSD. In particular, the intentional strategy can take account of both reactive components and user activity (cf. [Den87, p.17, p.20 & p.22]).

wanting, and so on [Hau97]. In the light of this discussion, traditional SSD can be viewed as an 'intentional strategy' for design, and the developed system as an 'intentional system' (cf. [Den87, p22-23]).

For more accurate and detailed prediction of the intentional system, the method of decomposition is exploited. By this method, other intentional strategies at a number of different levels of abstraction can be used in a descending series to decompose each higher-level strategy into a number of lower-level ones. After continuous decomposition, an intentional system can be developed by a collection of hierarchical intentional stances whereby the intentionality of the developer is then exhibited in the behaviour of the intentional system. In this way, the behaviour of an intentional system can be predicted, though the results may be flawed.

From the developer's perspective, the intentional strategy as a means for drawing up design specifications relies upon being able to conceive abstract entities capable of performing required functions. This is the appropriate way to organise the design activities before undertaking the detailed implementation in terms of formal abstracted notions, such as search trees, data structures, and evaluation algorithms. For example, in the object-oriented framework, the design specifications will be couched in terms of the 'objects', 'properties' and 'procedures', irrespective of whether or not these have been clarified in detail in the mind. The entities identified by the intentional strategy will be established by top-down analysis rather than by a bottom-up approach from the underlying mechanisms. This strategy presupposes an 'explanatory cascade' [Clar90], leading from top-level computational theory down to low-level elementary implementation, that needs to be taken on faith.

In the light of the above discussion, most design approaches proposed for software development, such as object-oriented design [Boo94, Jac92, CY90] and agent-oriented design [Sho93, Rao94], can be viewed as instances of implementing the intentional strategy. Within these approaches, the intentional strategy adopted by the developer is created from the clues that emerge from simplifying the object of study and postulating a set of rules for describing the behaviour of a system. Given this description in mathematical or logical forms, the developer can then prescribe the behaviour of the intentional system and refer its consequences to reality. If the consequences accord with reality, then the used intentional strategy is indeed feasible; if not, another intentional strategy should be implemented. Hence, the approach to design becomes a process of search optimised intentional strategy whereby the behaviour of a system can be predicted.

However, because of the growing size and complexity of today's information systems, descriptions specifying the system's behaviour (or intentionality) have become 'moving targets'. They are difficult to capture and pin down into a specification cabinet [BCDS93]. The need for incremental descriptions of such systems makes any preconception of the design system unreachable and inappropriate [Blu94a, Blu94b, Fis91, Bro87]. That is to say, it is in practice hard to describe clearly the top-level intentionality. As a consequence, it becomes problematic to adopt the intentional strategy for the development of complicated computer models having open or ill-defined requirements [Hen96].

Additional concerns are raised by design activities, such as verification and validation, that are used for the purpose of characterising an intentional system with the required predictability. These activities demand a rigid correspondence between the specifications and the implementation of the system's behaviour. The fixed relationship indicates the correctness of the developed system, but it also constrains the developed system from adapting to a rapidly changing environment. As a result, some inevitable problems emerge, when the developed system is applied to an open real world [Bro87, Bro95].

Today, it is often no longer acceptable if a developed system is correct but solves only the problems for which it was originally designed. Ideally, it should be able to grow and change in order to solve further relevant new user problems that emerge over time. In other words, 'building the thing right' and 'building the right thing' is not enough: today's systems have to be adaptable to the changing real world and cope with the diverse situated needs of users in solving their problems [FC96, Flo87, Leh98, LR98]. This fact highlights the inadequacy of an SSD based on the intentional strategy with step-by-step refinement in phase-based process models.

In fact, software design can be viewed as a problem-solving activity, and as such it is very much a matter of trial and error [Vli93]. I. Sommerville has provided the following helpful description:

> "[Software] design is a creative process which requires experience and some flair on the part of the designer. Design must be practised and learnt by experience and study of existing systems. It cannot be learned from a book. Good design is the key to effective engineering but it is not possible to formalise the design process in any engineering discipline."
> [Som92, p. 172]

In other words, the creative learning process of software design has to be unfolded by means of successive events of trial and error. Certainly, phase-based models provide only limited help. Neither the prescribed specifications for the designing product nor rigid engineering norms postulating the developer's activities can make the process progress in an effective and efficient fashion. For these models, trial and error is only possible in the form of retrospective feedback, that is, by learning and creating after the product has been produced. As a result, this afterthought often involves a very large additional and unanticipated cost. This consequence is evident in the fact that the maintenance cost of software systems is often very high [Sch90, Som95].

To incorporate trial and error in the intentional strategy, some models invoke the *experimental strategy*, in which one can treat elements of a system as rational agents and explore their behaviour through experiments. With the integration of the two strategies, these models, such as prototyping [Rei92, Mar91, And94] and scenario-based analysis [DF98, RSB98, SDV96, WPJH98], seek to explore and clarify the behaviour of the developing system through continuous experiments. The experimental exploration can provide the developer with experiential information on how the system will behave.

The two strategies are complementary for shaping the behaviour of the developed system. The intentional strategy is engaged in a top-down approach. It decomposes a higher-level strategy into a number of more accurate and detailed strategies by providing a less abstract level of descriptions for their design. On the other hand, the experimental strategy can be treated as a bottom-up approach in which certain parts of the system are clarified individually and incrementally integrated together. By means of both strategies, the developer can search for the optimal solution to prescribe the behaviour of the system.

In this respect, the behaviour of such a developed system (or an intentional system, in Dennett's term [Den87]) exhibits the developer's individual intentionality but also reflects the connection between his/her experimental experience and the system. Nevertheless, this also indicates that the development process and its products are characterised by the individual mental states of this external observer, including knowledge, understanding, judgement, and so on. The developed system is in effect closely related to the developer's individual cognition of the user's needs. As a result, it is very difficult for those models that involve both the intentional strategy and the experimental strategy (and these models that involve only the intentional strategy) for SSD to avoid the emergence of a gap between the developer's system and the user's system. As described in Section 1.1, this gap, that mainly arises from the cognitive difference between the developer and the user, has led to practical difficulties in SSD and in the use of the developed system in the user's environment. In order to narrow the gap, a strategy for evolution has been to some extent embedded in most models.

Evolution is a classic concern of SSD. Its feature of incremental refinement is found in many models for SSD. This feature can be embedded in the whole development process [Boe88, Yeh90], or just parts of it, for example, the implementation phase [Boo94]. B. Boehm proposed a spiral model to highlight the iterative framework of software development in which software systems are incrementally refined and enhanced. R. Yeh recommended a software evolution paradigm in which – in order to keep pace with the changing environment – validation and evolution are embedded in the whole development process, rather than performed as an afterthought upon completion of the development [Yeh90]. Instead of concentrating on the evolution of a software system in its development process, M. Lehman placed the emphasis on the evolution in its operational domain. He argued that the organisation of developing and maintaining a large software system should itself be regarded as a self-stabilising feedback system [Leh94b, Leh97, LR98]. The feedback from the operational domain (of users) is the main resource for the development of a software system, whereby the understanding of the developer and the structure of the software system are evolved.

Evolution in SSD can be unfolded in two stages. One is at the stage of system construction where developers are developing the system in order to satisfy the user's requirements. The other is the stage of system operation where the system is used in the user's actual practice. During the stage of system construction, evolution is meant to develop a software system incrementally in order to satisfy the emerging requirements of the user's intended system [Boe88, Boo94]. Evolution in the system operation stage is to refine the developed system in order to be consistent with the user's actual system [Leh94b]. Though they have different subjects, both types of evolution are processes where the developer's system, which is developed or still under development, is changed in order to coincide with the user's system, whether it is in actual use or is just being conceived for use. In short, evolution within the above models is viewed as a process in which the developer's system is evolved in order to narrow the gap between the developer's system and the user's system.

Hence, design activities for refining or enhancing the developer's system are applied to the process in order to bring the developer's system and the user's system as close as possible. However, this has not proved very satisfactory and successful. This is partly because the developer's system usually changes too slowly to keep up with the change of the user's system. A more important reason is that the evolution through the refinement and enhancement of the developer's system cannot rule out the emergence of cognitive differences between the developer and the user. As described in Section 1.1, being an 'outsider' of the user's system that resides in a rapidly changing environment, the developer can hardly reach a full understanding of the system. The developer's understanding of the user's system is usually limited and fragmented. It is not surprising that evolution invoked by the developer with his/her incomplete understanding still exposes the gap between both systems. For this reason, it is possible to regard the developer-centred evolution within these models as a form of design associated with the intentional and/or experimental strategies mentioned above.

Where shaping the agency of agents is concerned, it is plausible to regard EM as involving both the intentional strategy and the experimental strategy in an open-ended situated manner. The intentional strategy is embedded in the agency of certain agents whose responsibility for state change has been confidently described. And the experimental strategy, which can provide the modeller with practical experience in a particular situation, is invoked by means of 'what if' experiments in order to explore the agency of agents. With situated modelling (see Section 3.2), the implementation of the two strategies for SSD in EM becomes more effective and powerful for shaping the agency of agents. The intentionality of the modeller for shaping the agency is thus exhibited in the process of experimental exploration, and at the same time his/her empirical experience of how the system will behave in a particular situation is gained. In this respect, like those models that involve the two strategies for SSD, EM also indicates a process of individual cognition that is associated with the modeller's intentionality and empirical experience. This egocentric perspective of the modeller (that is, 'agency is in the mind of the external observer' [BeyMsc]) accords with Y. Shoham's assertion that 'agenthood is in the mind of the programmer' [Sho93].

A weaker notion of intentionality is taken into account in EM. The behaviour of the system developed by EM is not guaranteed to be permanent in the real world, though it still is expressed in the form of intentionality of the developer. Until confidence in the elements' behaviour has been established, theoretically the modeller can continue to perform 'what if' experiments to search for the optimal description of the agents' agency. This search process through both intentional and experimental strategies is consonant with D. Sonnenwald's concern for the knowledge exploration process for artefact design in which diverse knowledge from multiple domains, disciplines and contexts among specialists can be explored and integrated [Son96].

More significantly, DEM, that is proposed for clarifying and enhancing the distributed perspective on EM, involves a continuous process in which the agency of agents is evolved through their actors' (that is, A-modellers') interaction with each other in their (agents') ordinary context. This process is very similar to the evolution in an ecosystem for adapting to a changing environment by means of its natural emergent, situated activity [Hen96, AL89, SS95]. The evolution in an ecosystem, whose activities invoked for change (and their results) cannot be preconceived, is conducted by elements of the system itself rather than by an 'outsider'. It is not an exercise in retrospective patching by means of feedback with the aim of refining or enhancing the previous system.

Instead, it emerges from continuous local self-adjustments of the system in response to diverse situations encountered in a changing environment. Each local self-adjustment, by carrying out an incremental change to the system, may cause adaptive responses in other parts of the system. As a result, more local self-adjustments are invoked until each element of the system is stable in relation to the current environment. In other words, evolution within an ecosystem itself is carried out through the system's elements in interaction, and the details of its exploration process significantly affect the outcome. This evolutionary interaction between elements of an ecosystem and its changing environment enables the system to survive in the rapidly changing environment. Hence, evolutionary superiority results from just letting the system run by itself. In this case, there is no gap between the developing system and the actual running system because both are the same system.

In this respect, DEM involves the *evolutionary* strategy in which a software system is treated as an evolving system whose behaviour is adapted to its rapidly changing environment in a situated manner. Unlike the other two strategies mentioned above, the evolutionary strategy in DEM emphasises the role of interaction for adaptation. The interaction that is embedded in the evolutionary strategy leads the system to evolve so as to fit well into the current situation. It is unpredictable but powerful, because no prescribed algorithm is provided to constrain the evolution. The behaviour of the system is no longer persistent, since it could be adapted to a new situation at any moment. Like an ecosystem, the system evolves in response to its surrounding situation, and its evolution never ceases until it is thrown away. The evolutionary strategy proposed here for adapting a software system to a rapidly changing real world in an interactive manner is in line with P. Agre's view of an intelligent system: the structure of the system's behaviour is located in the interaction between the system and its environment [Agr95]. Note that evolution within DEM is accomplished by combining the evolutionary strategy with the intentional strategy and the experimental strategy. It is distinguished from design involving only the latter two strategies.

Certainly, it is very difficult for most traditional design models to support the evolutionary interaction between elements in a situated manner. Design within these models can be regarded as an exploratory search rather than an evolutionary process. This is because the behaviour of a system is designed by what J. Goguen has called 'introspection' where the elements and the causal links, or mechanisms, of the system are classified and identified through the imagination of the developer [Gog97]. Even given the experimental strategy mentioned above, the system behaviour is still bounded by certain prescribed descriptions. In other words, design is a searching process where the developer optimises the imagined description by continually changing the structure of mechanisms and the hierarchy of elements. This optimisation, making the system's behaviour quite describable and predictable, can presumably be achieved by providing enough consecutive events of trial and error. In this sense, these models are capable of supporting search but not evolution [AL89].

Within the framework of DEM, however, the evolutionary strategy can be employed in a natural fashion. It can be exploited in the two stages of evolution in SSD: system construction and system operation. During system construction, DEM empowers A-modellers to interact with their own computer models at any moment in order to adapt to a new situation happening in their observed worlds and/or their computer models. This new situation may immediately give rise to further adaptive responses from other A-modellers if this new situation on their computer models does not correspond closely to their observed worlds. Consequently, a continuous process of interactive adaptation emerges and at the same time the system evolves incrementally.

With the concepts of ethnomethodology, A-modellers in DEM interact with one another, their changing environment, and their computer models by pretending to be agents. In pretend play, it is as if A-modellers are affiliated to a society of agents and make their interaction with each other accountable. In this sense, it is plausible to view each A-modeller as an 'insider' of the system, who is capable of enacting the interaction between agents. Each interaction leads to a local self-adjustment on the A-modeller's computer model, and also brings the whole system and some computer models to a new situation due to the propagation of dependency.

In other words, A-modellers on behalf of agents invoke the evolutionary interaction for adapting the developing system to the current situation. This evolutionary interaction eventually leads to the evolution of the system. The concept of evolutionary interaction has been applied to the case study of a railway accident illustrated in Chapter 6 and also to develop a framework for modelling the situated process of requirements engineering discussed in Chapter 7.

The evolutionary interaction for A-modellers is situated. No result can be predicted because of the nature of situatedness and the uncertainty of the interactions that will be triggered [LR98]. As happens in an ecosystem, this interactive process should enable the system to converge to a stable state if the invoked interaction corresponding to the situated change is accountable to others and is adapted to the current environment. If there is a failure to reach convergence, for example, through conflict or inconsistency in the definition of an observable between A-modellers acting as agents, the developing system is unlikely to succeed. In other words, the system is evolved through the interaction between the 'insiders' of the system for adaptation in a situated manner. This evolution is very different from the developer-centred evolution mentioned above. The latter is carried on by the 'outsiders' of the system through design activities.

In a similar manner, the evolutionary strategy has also extensive application to the stage of system operation. Traditional design approaches aim at questing for a perfect software product without the gap mentioned above, but it is still like the Tower of Babel in the real world. This is because cognitive difference is inevitable while the developer, an 'outsider' of the developing system, develops the software system. From the perspective of users, the system they need is one that is able to solve their practical problems in time. In this respect, an adaptable system that can cope with emerging change may be more suitable for users in their real world than a perfect system. It is because invoking developer-centred evolution as suggested in traditional models is too late and expensive, due to the problem of tacit knowledge [Gou94]. Moreover, it is true that no one can do the adaptation better and faster than users, *if* they are qualified enough, since only they know what is needed at this moment for solving their current problem. In fact, the concept of enabling users to adapt their systems to a rapidly changing environment has been well implemented in a spreadsheet system. Users of a spreadsheet system initially develop a spreadsheet, and then adapt it by themselves to their changing practices. In other words, spreadsheet use is evolution of a spreadsheet by the users themselves. In this sense, the use of spreadsheets blurs the distinction between users and developers.

Focusing on users, rather than relying on the afterthought activities of an outsider for refining or enhancing the developed system, is an alternative way of bridging the gap between the developed system and the actual system. This means letting users adapt the system in use by themselves. To achieve this, it will be necessary to enhance the user's ability to perform 'end-user programming'. The developed system must also be characterised by considerable adaptability, so as to enable the user to evolve the developed system in a timely fashion. The first step towards enabling evolutionary interaction to occur at the operation stage is to create a system that is readily adaptable to its rapidly changing environment.

However, most design approaches end up with a final product that provides little support for adaptation in a situated manner. Within this final product, all knowledge associated with the development of the system, such as its architecture, data structure and functionality, must be determined and unchangeable before the product is operated in the user's environment. This is because the knowledge embedded in the final product is fixed and determined by optimisation aimed at satisfying a prescribed goal. The frozen boundary of the final product reduces the risk of its being an incorrect or incomplete system but also prevents the developed system from dynamic modification for adapting to the user's practice.

In addition, in many cases, users are not allowed to modify the final product, even if they are sufficiently qualified and the modification is urgently required. This is because, from the developer's viewpoint, any change, no matter whether it is for correction, perfection or adaptation [Fai96], should be finished by taking the whole system, rather than local parts, into account. Any local change can affect other parts of the system and then lead the system into a mess if this affect is not taken into global account. It is recognised that local changes normally introduce more errors and lead to higher costs [STM95, Som95]. Indeed, this demand for a whole-system perspective is understandable and inevitably necessary in conventional top-down approaches for SSD.

Therefore, it is clear that the crucial issue of invoking evolutionary interaction during system operation is the difficulty of building a system that is adaptable through local change. In the light of the definitive programming described earlier (Section 2.3), the need of propagating local adjustment to global change is met through dependency maintenance. Any change in a local part can be propagated to the whole system and affect the other elements if the change is authorised. In other words, the user can make a local change for satisfying his/her individual need and/or for adapting to the current situation without taking global propagation into account.

With the benefit of definitive programming, evolutionary interaction can be accomplished through local changes from the users of the system. That is, the system can be evolved through its qualified users in interaction during the stage of system operation. Clearly, the capability of users (such as programming skill) is another critical factor for the success of the evolutionary interaction in the stage of system operation. Hopefully, performing local change to the evolutionary interaction is usually less difficult and demands less sophisticated skill in programming. Also, advances in end-user computing have increasingly reduced the problems of incapability [DL91]. In other words, in the near future, the capability of users should not be a big problem for invoking evolutionary interaction. The trend towards enabling the user to adapt the system to diverse situations in use accords with the demand for *design-in-use* in which the design activities are extended to the operational domain of users in order to meet their changing needs [Fis93, SH96].

By supporting the evolutionary strategy proposed here, DEM can bridge the gap between the developed system and the user's actual system in a significant way. In order to make the developing system as close as possible to the intended system, A-modellers can act as agents as if inside the intended system during the system construction stage. Recognising the fact that the user's need is never complete, the goal of the evolutionary strategy is to build an adaptable system that can evolve in a situated manner. Given such a system, the user, on the basis of his/her individual profession and practical experience, can adapt the system to his/her rapidly changing environment. Accordingly, the adapted system can always keep up with the user's actual system. In fact, the system-under-development converges to the system-in-use.

Chapter 5

Implementation to Support Distributed Empirical Modelling

DEM aims to enable all modellers in a distributed environment to interact collaboratively with each other in order to establish a working or shared understanding. Such understanding is crucial for improving interpersonal interaction in general and for shaping the agency of agents whose roles are associated by A-modellers in particular. In order to provide a distributed environment for collaboration, a tool supporting DEM (discussed in the previous chapter) is essential. This tool should be able to support the individual modelling task of each modeller, the interaction between modellers, and the interference of the S-modeller in this kind of interaction. It is clear that the tool tkeden[1], created to support individual modelling in EM, is unable to cope with all these needs. This motivates the creation of a new tool to satisfy the further requirements.

5.0 Overview

This chapter discusses the implementation issues involved in creating a tool to provide modellers with a collaborative distributed environment for supporting DEM. The tool

[1] Since the scripts of Donald and Scout are interpreted by Eden in tkeden and dtkeden, the term Eden will be used to represent the integration of Eden, Donald and Scout in this thesis, except where indicated.

tkeden, described earlier (Section 2.2), supports only the individual modelling task of the superagent within the framework of EM. Obviously, this functionality is not sufficient for a distributed environment supporting collaborative interaction between multiple modellers. On the other hand, the core part of tkeden, that is, its dependency maintainer, is still very useful for providing the environment with the advantages of definitive programming. In view of the limited time available for developing a new tool, the author decided to reuse this core part and extend it to a distributed environment.

A new tool called dtkeden has thus been developed by the author on the basis of the dependency maintainer implemented in tkeden. With this tool, modellers are empowered to interact with their own computer model, which corresponds to their own observed world, in an open-ended, situated manner. This kind of human-computer interaction not only facilitates individual understanding through modelling but also promotes other modellers' understanding through network communication. In this way, modellers can interact with each other via computer-mediated communication in order to support their collaborative interaction in a distributed environment.

In section 5.1, the main technical issues arising from the construction of the networking environment for dtkeden are discussed. First of all, a star-type logical network configuration for dtkeden is proposed. Then, the popular concept of client-server communication is introduced as the fundamental mechanism of network communication within this configuration. By integrating this mechanism with the socket program-to-program protocol and the TCP/IP communication protocol, a two-way network communication is devised for dtkeden. The networking feature can be used to shape the agency of the agents within an application by integrating a social process and a cognitive process, as described in the previous chapter. In addition, the problem of synchronisation arising from network communication between computer models is identified, and a solution (designed by the author) that involves using a request-wait-reply mechanism with the aid of queued service numbers, is proposed to resolve this problem.

Section 5.2 focuses on the implementation of support for diverse interaction modes between modellers. The interaction between modellers is one of the main themes for DEM. Different kinds of interaction should be supported by different interaction modes for serving particular purposes. For example, an open, shared topic may suit a many-to-many interaction style, but negotiation-seeking in order to solve conflicts between group members is often invoked in the context of one-to-one interaction. Within the network configuration proposed in Section 5.1, four interaction modes have been implemented in dtkeden: broadcast, private, interference, and normal.

In the *broadcast* model, an open, shared environment for the interaction between modellers is provided. Any interaction triggered by a modeller is propagated to other modellers. In contrast to the broadcast model, the *private* model provides each A-modeller with an individual channel for one-to-one communication. The *interference* model then enables the S-modeller to interfere with the interaction between A-modellers in order to create a new context for modellers. The mode that is most characteristic of DEM is the *normal* model, which enables each A-modeller, acting as an agent, to interact with others by accessing authorised observables. This interaction mode exhibits the key principles of a framework for DEM in their full generality, and highlights the concepts of ethnomethodology and distributed cognition in particular. Within dtkeden, these four modes are used alternatively and can be changed at any moment in response to the context of modellers.

The main topic discussed in Section 5.3 is the reuse of software components. Over the past decade, the concept of software reuse has become widely promoted in the software community as a way of reducing development and maintenance costs and increasing productivity [Pau97, Pre97, Pri93, Som95]. The traditional approach to software reuse usually seeks to develop software components that can be reused in their entirety without adaptation (so called 'black-box reuse'). In practice, the application of such black-box reuse in SSD remains a major challenge for software developers [PF87].

Section 5.3 offers an alternative to black-box reuse: adaptable reuse, which involves the reuse of software components by adaptation[2]. This mirrors the way in which a human being reuses experience by adapting it to similar situations in everyday life. This adaptable reuse hopefully loosens the boundary of reusable components and makes reuse more widely applicable. A particular kind of adaptable reuse can be achieved in EM through the concept of virtual agent implemented in dtkeden. First, the concept of virtual agent is proposed, and then the method of using it to facilitate adaptable reuse is discussed. A new kind of observable – the generic observable (GO) – is proposed to represent a set of definitions (that is, a definitive script) that can be reused through adaptation in this sense. The remainder of this section discusses the difference between using Abstract Data Types (ADTs) and GOs for developing reusable software components.

5.1 Network Communication in dtkeden

As explained in Section 4.2, network communication has an enabling role in supporting DEM. Within the framework for DEM, the S-modeller, as the external observer, and the A-modellers, as the internal observers, are empowered to perform empirical modelling by means of interaction with each other as well as interaction with their own computer models. In order to provide such a distributed, computerised interactive environment,

[2] The concept of adaptable reuse should not be confused with so called "white-box" reuse. See Section 5.3 for more details.

network communication[3], connecting together all the computer models of modellers, thus has the highest priority in the implementation of dtkeden.

This section illustrates the technical issues and problems encountered in implementing the distributed architecture of dtkeden. In the first subsection, this architecture is discussed on the basis of a star-type network configuration and client/server communication. The problems raised by asynchronous communication are identified and solved by the new synchronous communication mechanism implemented in dtkeden. The details are given in the second subsection.

5.1.1 A Distributed Architecture with Client/Server Communication

Since network communication is now ubiquitous in everyday life, the investigation of distributed systems has become an increasingly important trend in the software community. A distributed system normally consists of a set of software components located on different machines[4] and a network allowing these software components to communicate with each other to produce an integrated computing facility [CDK94, Hug97]. One of the fundamental issues in developing a distributed system concerns the architecture of network communication. From the perspective of software development, the architecture involves at least two parts: a logical network configuration and the techniques of network communication. The former determines the distribution of software components of the system, and the latter enables these components to communicate with each other via a physical network.

[3] In this thesis, "network communication" and "data communication" are interchangeable terms to denote an information technique whereby electronic data can be transferred from one hardware device to another through physical network facilities.

[4] In principle, it is not really necessary for all software components to run on different machines, but their communication with each other definitely must go through a communication network.

Referring to the framework for DEM shown in Figure 4-6, a star-type logical network configuration – probably the most common of all configurations [CCH80] – is exploited for network communication in dtkeden (see Figure 5-1). This star-type network configuration represents a logical interconnection between software components and is independent of the network communication topology, which illustrates the physical interconnection between the hardware components in which the software components reside [Hug97]. From the perspective of developing distributed systems, the logical configuration of software components is much more important than their physical configuration [CCH80].

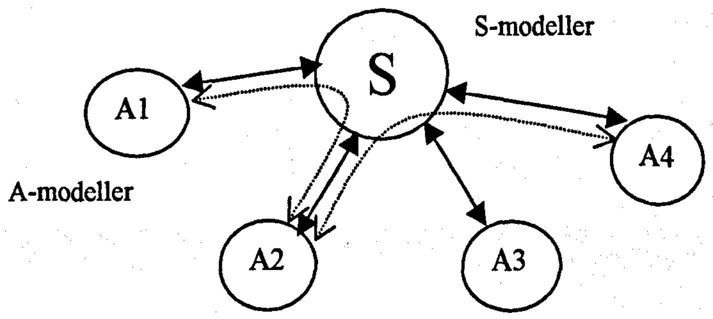


Figure 5-1. A star-type logical configuration for the network communication in dtkeden

There are two kinds of nodes in this star-type configuration. At the points of the star, A-nodes are created to represent A-modellers as shown in Figure 4-6. Each A-node can communicate with the S-node, at the centre of the star in Figure 5-1, via the network. Where the communication between two A-nodes is concerned, dtkeden does not provide a direct dialogue between them. Instead, this kind of communication is achieved through the involvement of the S-node, that is, by using the S-node as a message transferring centre to transmit the message[5] from one A-node to another. This kind of A-node-to-A-

[5] It should be noted that a *message* in dtkeden typically takes the form of a definitive script, possibly together with auxiliary functions and actions. Contrast the common use of 'message' in object-oriented programming to refer to a method invocation [Boo94].

node-via-the-S-node communication is provided through a built-in procedure in dtkeden. The modeller at the A-node can simply call this procedure to interact with the specified modeller at another A-node.

The S-node representing the S-modeller is responsible for the transmission of all messages between two A-nodes. This superior responsibility highlights the importance of the S-modeller as the only modeller (representing 'God's view') who is supposed to have direct access to all contexts. It also enables the S-modeller to intervene in the interaction between A-modellers. In this way, adopting a star-type network configuration has the advantage of enhancing the control of security and access privilege to observables, as explained in Section 5.2.

After determining the network configuration of dtkeden, the technique used to support network communication is taken into account. Following state-of-the-art techniques of network communication, the technique of the client-server (or request-response) model[6] is chosen to support the communication between computer-based models in the framework of DEM. The client/server model is currently the best known and most widely adopted system model for distributed systems [BG96, CDK94, KJ98, Som95]. It provides an effective general-purpose approach to the sharing of information and resources through network communication.

A typical client-server model, as shown in Figure 5-2, is oriented towards service provision. Within this model, each invoked client/server network communication consists of the following steps:

1. transmission of a request from a client to a server through the network;
2. execution of the request by the server;

[6] Although this term can also express a hardware-oriented view [BG96], it is used here to refer to a software-oriented technology.

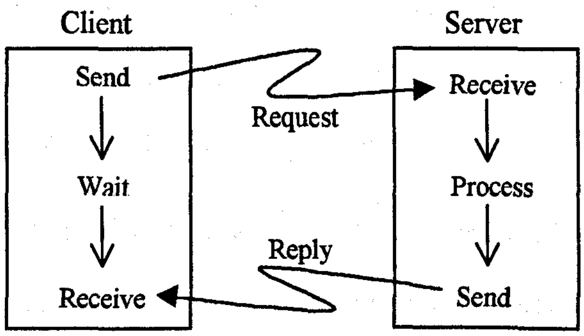


Figure 5-2. A typical client/server communication model

3. transmission of a reply to the client through the network.

This pattern of communication involves the transmission of two messages and a specific form of synchronisation[7] of the client and the server. As soon as a request is sent in step 1 to invoke a communication, the client is blocked until it receives a reply from the server in step 3. At the same time, the server must become aware of the request message sent in step 1 as soon as it arrives. After processing the received request, the server then sends its reply to the client in order to unblock the client and allow it to continue.

This elementary pattern of client/server communication can be implemented in terms of the message-passing operations *send* and *receive* as outlined above. Before sending a request, a client must know the names of the available servers and the services that they provide. Each request contains a communication identifier that is used to transmit the reply to the client. However, a server need not know either the identification of clients or how many clients there are. Typically, when a server starts up, it registers itself with a naming service, stating its network address and a global name for the service that it provides. A client obtains its network address by interrogating the naming service and is thus able to communicate with the server by using the same global service name.

[7] In this case, the synchronisation mechanism is called blocked communication. In other architectures, asynchronous communication may be involved: here the client is not waiting in suspense for the reply to be sent by the server.

Although client/server communication can be installed in various configurations - either centralised or highly distributed, as described in [BG96] - the communication of their components often follows this elementary pattern.

The implementation of dtkeden exploits the concept of protocols for program-to-program communication (or interprocess communication [Bac93]) that is widely used for implementing distributed systems [CDK94]. It enables direct dialogue between applications: for example, a C program calls a library function *send* to access the network communication layer. The library also offers functions for the initialisation of links, the connection establishment and breakdown, and the control of the data transmission itself, that relieve the programmers of most aspects of network communication. For example, the Socket protocol, used widely in distributed systems, provides such functions for a common communication interface. In dtkeden, the socket protocol provided by the Tcl/Tk[8] software package is used. Also, the TCP/IP protocol family, which is popularly applied to a local SUN workstation environment and available at Warwick, is used to provide the lower-layer network communication in dtkeden. An in-depth discussion of all these details is beyond the scope of this thesis.

In summary, the client/server communication in dtkeden is achieved by using the TCP network protocol and the Socket abstraction in the Tcl/Tk level. Figure 5-3 shows the layout of communication between the S-node and A-nodes, as implemented in dtkeden.

In some client/server architectural software systems, each client/server component is specified to be either a client or a server, but not both. Each component is only responsible for either requesting a service (that is, being a client) or providing a service

[8] Tcl is a scripting language and an interpreter for that language that is designed to be easily embedded into an application, and Tk is a graphical user interface toolkit for Tcl [Ous98, Wel97]. The tool tkeden uses the Tcl/Tk for the purposes of graphical interface control and visualisation, and dtkeden uses it also for dealing with network communication.

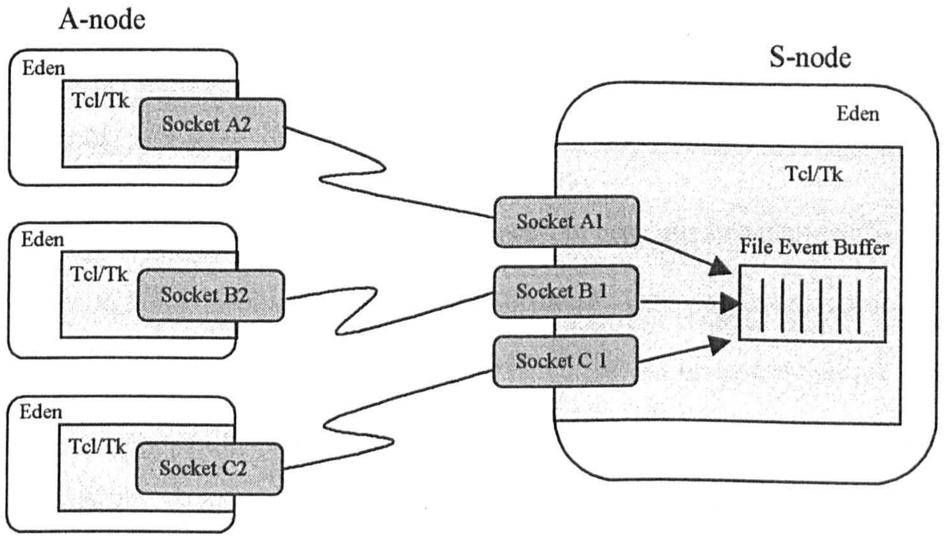


Figure 5-3 The communication between the S-node and the A-nodes

(that is, being a server). In such a case, there is only one-way, and not two-way, communication between components. For example, in the database world, the data repository and its associated functionality are termed the server; the client is the application (which could be on the same hardware or software, or not) [Aye95]. This exclusive attribution of client or server status indicates the one-way communication between the client and the server. In a similar fashion, for example, the Internet or WWW world has the web browser as the client, and the application providing the requested information as the server.

However, dtkeden refers to a client/server component in a different way. In dtkeden, a client means the requester of a service, and the server is the provider of a service. Any client/server component can be both a client and a server, depending on what it is doing at the time. In this case, 'client/server', as it applies to the implementation to support DEM, denotes a client/server communication technique rather than a client/server architectural configuration as described in [BG96]. Therefore, each node in Figure 5-1 is not exclusively specified to provide or request services. Instead, each node is devised to perform both functions in order to provide its user, that is, the modeller, with two-way rather than one-way interactive communication. That is to say, each modeller in any node of either kind can communicate with other modellers by sending a request or providing a service.

An interesting comparison can be made between the client/server architecture of dtkeden and the information sharing architecture that is used in Empirical Worlds, recently developed by R. Cartwright for EM in [Car98]. In Empirical Worlds, information sharing is achieved by centralising all definitive scripts to the server machine so that clients can access the scripts via a 'definitions-database' system. The authority of each client accessing these definitions can be specified through a system of security control similar to that of the UNIX file system. In that case, clients can connect to the server in order to perform a modelling task. They do not need to communicate with each other, since each of them remains essentially independent. Any task for modelling, such as giving (re)definitions, is executed only in the server and displayed in the client. This centralised modelling environment allows each modeller to undertake exploration and experiments independently but to share commonly used definitive scripts in a collaborative manner. Obviously, such architecture, though having the advantages of sharing information and reusing definitions, provides limited support for DEM.

5.1.2 Synchronous Communication for dtkeden

One of the major challenges involved in the implementation of dtkeden arises from the need to tackle its synchronisation problem of client/server communication between nodes via a communication network. Before discussing this issue further, it will be helpful to re-examine the way in which intercommunication between Eden and Tcl/Tk is integrated with the execution of Eden.

As already explained, tkeden is a hybrid tool combining Eden and Tcl/Tk. Eden is an interpreter developed in C program language, and the Tcl/Tk is a popular tool for event-driven programming. In tkeden, the tool Tcl/Tk is applied to deal with issues of external presentation such as the user interface, graphical display and window interaction, whilst Eden then focuses on handling the internal representation, for example, dependency maintenance and data manipulation. Since the Tcl/Tk is an event-driven tool, most of the time it is in the so called *EventLoop* loop, where the Tcl/Tk interpreter is devised to keep track of the condition of each event handler already issued in order to trigger its further actions when the condition is satisfied. For example, when Tcl/Tk finds that a button is pressed, the event handler *ButtonPressed* will be invoked to undertake specified actions. In the case of tkeden, the event handler *DoWhenIdle* has been specified to call a function of Eden, when Tcl/Tk is in an idle state. Hence, the intercommunication between Tcl/Tk and Eden is achieved by programming each to call functions of the other, so that program control passes forward and backward between them.

Two queues have been devised to hold received messages in Eden. One, called the executing queue (EQ), holds those messages to be executed. The other, called the waiting queue (WQ), stores the received message waiting to be moved into EQ. There is a function to process messages in EQ and then move the contents of WQ into EQ. It is this function that is triggered by the event handler *DoWhenIdle* when Tcl/Tk is in an idle state.

When tkeden is started, Eden builds up its initial state, then passes control to Tcl/Tk and awaits further actions invoked by the received messages. After taking program control from Eden, Tcl/Tk, as explained above, enters the EventLoop in which Tcl/Tk keeps track of the conditions of all issued event handlers in order to trigger further actions specified by these event handlers. Now, if Tcl/Tk receives a message, for example from the input window where the user can enter scripts, Tcl/Tk calls a function in Eden, and passes it the program control and the received message. This function then appends the received message to WQ and returns the control to Tcl/Tk. When Tcl/Tk enters an idle state, the event handler *DoWhenIdle* calls a function in Eden and passes the program control to Eden. As mentioned above, this called function then executes each message holding in EQ and after the execution moves the messages in WQ into EQ. Finally, Eden returns control to Tcl/Tk and again Tcl/Tk enters the EventLoop. Figure 5-4 illustrates this intercommunication mechanism.

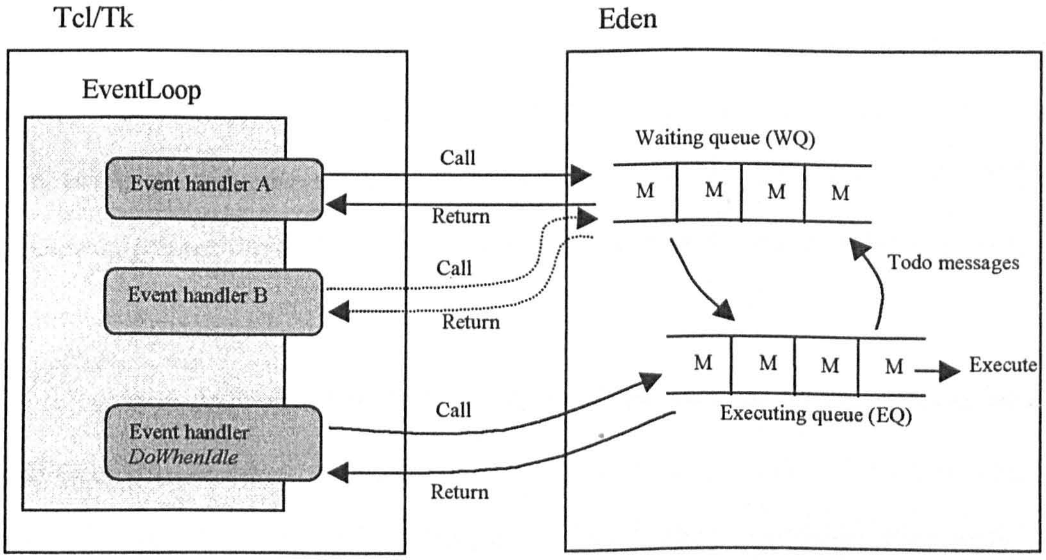


Figure 5-4. The intercommunication mechanism between Eden and Tcl/Tk

The main concern here is that Eden does not execute messages appended to its WQ until its EQ is exhausted. This feature allows Eden to deal with messages in a state-based fashion, that is, all messages arriving in the same conceptual state are executed at the same time. In particular, this mechanism underlies the development of a virtual machine for dependency maintenance, which is the most important and fundamental principle for designing a tool to support the framework of EM. Therefore, all auxiliary actions arising from dependency maintenance (e.g. as a result of triggering actions) are appended to EQ rather than WQ. In this way, it is as if they and all their preceding actions are executed in the same state (or conceptually at the same time). In practice, it is clear that genuine concurrent execution is not possible for Eden, but by this mechanism concurrency can be achieved conceptually by separating messages arriving in one state from those that arrive in the next state.

In effect, as soon as Eden moves messages in WQ into EQ, it enters a new state. Until all these messages, including those moved in from WQ and those incurred from dependency maintenance, have been processed, Eden is viewed as being in the same state. This is the reason why two queues are needed. In this mechanism, it is evident that each time Tcl/Tk passes a message to Eden, it is appended to WQ rather than executed. In other words, the reply sent back to Tcl/Tk is not a confirmation that a message has been processed, but only an acknowledgement that a message has been received. This acknowledgement does not guarantee that processing of the received message will be immediately carried out.

In stand-alone use of tkeden, this mode of acknowledgement does not cause too many problems in terms of synchronisation. This is because Tcl/Tk still needs to wait for the execution of the acknowledged message after receiving the reply of acknowledgement. In tkeden, the only data stream from which input messages can be obtained is the interface between the modeller and Tcl/Tk, e.g. the input window devised for the modeller to input scripts. The messages received from this data stream can only be passed to, and processed by, Eden in sequential order. To be precise, these messages are handled by Tcl/Tk and Eden one at a time since tkeden is not a real concurrent system. This one-at-a-time mechanism forces Tcl/Tk to wait for the execution of the last message in Eden before it is available to receive another message. As a result, Tcl/Tk sometimes has to be suspended when Eden is dealing with a substantial computational task (such as a complex evaluation). Despite this, the synchronisation of Tcl/Tk and Eden in the same tkeden can still be achieved. However, for dtkeden, the situation is different. The mechanism of receiving and processing messages between Tcl/Tk and Eden does not work in the same way as in tkeden, because more than one machine with similar interfaces between Tcl/Tk and Eden are involved.

First, in addition to the original window-based interface, another data stream is provided by dtkeden for collecting messages. This stems from the sockets implemented for client/server communication. In dtkeden, the socket technique for program-to-program protocol is used to establish communication channels in the Tcl/Tk level to receive/send messages from/to other machines via the communication network. Hence, within a machine running dtkeden, the collected messages can come from other machines. In other words, messages requesting services can come from both kinds of data streams, and can be mixed up together and executed by Eden.

An event handler *fileevent* is implemented to deal with the tasks of reading/writing a message from/to a socket following this event handling strategy mentioned above. For example, when a message requesting a service arrives at a socket in the server, a signal is sent to Tcl/Tk to start the event handler *fileevent* of this socket. This event handler then buffers the received message and passes it to Eden. As soon as the message is received, Eden appends it to the WQ and passes program control to Tcl/Tk with an acknowledgement, as described above. This acknowledgement then leads *fileevent* to reply to the client with a confirmation.

This message-passing mechanism for remote communication (as illustrated in Figure 5-5) is essentially asynchronous, though it still takes the form of a kind of synchronous message passing. It does not guarantee that Eden in the server has accomplished the requested service when the client receives the confirmation reply. In effect, it just acknowledges that the request has been appended to WQ in Eden. This mechanism is not the same as typical synchronous client/server communication in which the client waits for the server's reply in order to confirm that the requested service has been executed.

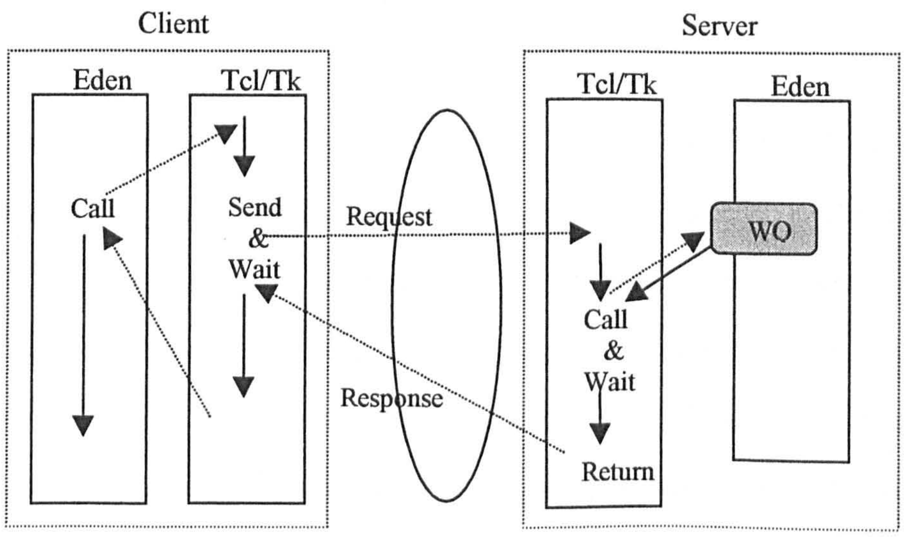


Figure 5-5. The asynchronous communication in dtkeden

An even worse situation can arise in a distributed environment, when two or more clients are sending various requests to the same server. Figure 5-3 shows a situation in which many senders are connected to one receiver. In this case, there are as many sockets in the server as there are senders in order to distinguish individual connections between each client and the server. This disambiguation of messages does not resolve all communication problems, however. Time delay is unavoidable and the sequence of executing the received requests is unpredictable. After all, the Tcl/Tk interface to the server can only handle one event at a time. It should be noted that an event handler *fileevent* is created for each socket in the server and in the client.

Therefore, problems of synchronisation between the sender and receiver arise, especially when modelling distributed, real-time systems in which timestamps in connected machines have to be consistent. For example, in a case study of modelling a railway accident (discussed in the next chapter), the same train's position has to be displayed at the same place on the screens in different machines running dtkeden. The asynchronous communication between these distributed dtkeden environments potentially leads to some unacceptable situations: for example, the trains may be displayed at different positions; some of them may stop and then jump to a new position instead of moving smoothly, and so on.

In fact, such problems of synchronisation could be avoided if the client/server communication through sockets were to be implemented in the Eden level rather than in the Tcl/Tk level. In that case, the client and the server will be totally suspended until the requested service is provided, since there is only one thread in the server to deal with the requests of clients. That is to say, no more requests can be accepted before the current request has been served. This limitation on processing of requests leads to inconsistency in the suspended components' states and other components' states. To resolve this problem of inconsistency, multi-thread or event-handling techniques are needed for dtkeden. However, implementing these new techniques using the C language could involve more unpredictable challenges, such as problems of concurrent control. Due to the limitations of research time, the decision to implement client/server communication in the Tcl/Tk level was made. This makes the problems of asynchronous communication identified here inevitable, and a solution is therefore needed.

To overcome the problems posed by asynchronous communication in dtkeden, a request-wait-reply (RWR) mechanism was devised to improve the existing message passing mechanism provided by Tcl/Tk and Eden. This new synchronous mechanism for remote communication in dtkeden, illustrated in Figure 5-6, will not allow the server to reply immediately to the client with an acknowledge message when it receives a request that needs to be served synchronously. Instead, the event handler *fileevent* in the server will enter a state to wait for the accomplishment of the requested service, even if it is queued in WQ in Eden. This waiting situation in the server will consequently lead the event handler *fileevent* in the client to enter a waiting state as well, since the communication between these two fileevent handlers via sockets is essentially synchronous.

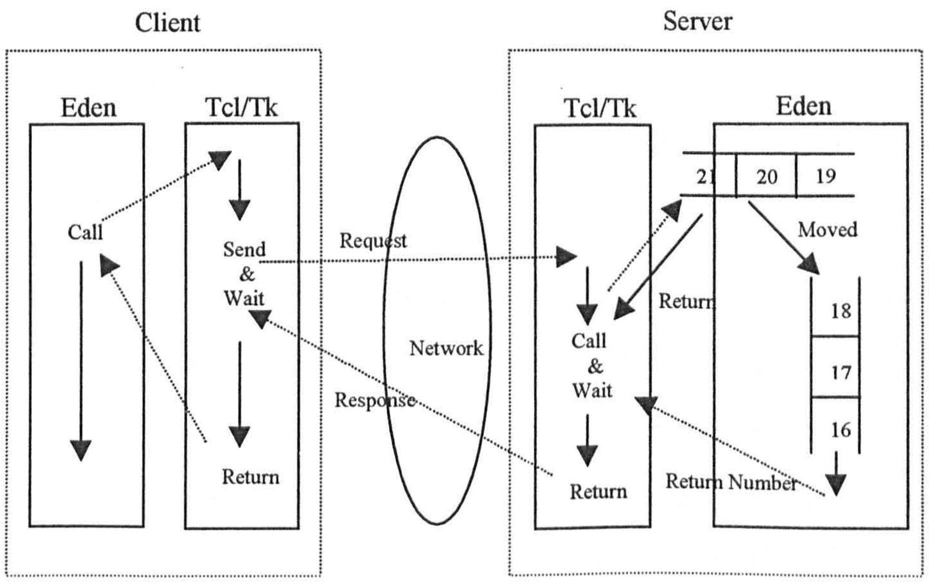


Figure 5-6. A synchronous model for remote communication in dtkeden

It should be noted that when they are in a waiting situation, neither the server nor the client is completely suspended. Only partial components, that is, the pair of sockets connecting the communication between the server and the client, are suspended: for example, the pair of sockets socketA1 and socketA2 shown in Figure 5-3. The partial suspension enables the client and the server to continue to deal with messages from their other input streams, such as the graphical interfaces and sockets in the Tcl/Tk level. Only when the requested service is provided does the server send the client a confirmed reply to release both suspensions. The need to minimise the suspension of the client and the server in their synchronous communication is the main reason why the communication sockets are implemented in the Tcl/Tk level rather than in the Eden level.

One of the crucial issues to be addressed in implementing this RWR mechanism is how the event handler *fileevent* knows that its request has been served after entering a waiting state. Within a single client/server communication, as illustrated in Figure 5-2, the server can only serve one request at a time. When it is busy serving, other requests must wait in a queue for their turn to be executed. Once a request is executed, the server continues to process the request without delay and interruption until the request is completed. Hence, when a request is accepted, the client sending this request clearly knows that the server is processing the request, and the server also knows to whom the reply should be sent.

However, on the account of dependency maintainer in tkeden, the processing of messages in dtkeden is state-based rather than sequential. As explained earlier, each request arriving at the server must be queued in WQ rather than be executed immediately. Because of this accepted-first-served-later feature in Eden, the event handler *fileevent* loses contact with its request. If no account is taken of this fact, RWR mechanism devised above can keep the *fileevent* waiting for a reply, but the *fileevent* cannot recognise whether or not its request has been served, since the linkage between the *fileevent* and its request has disappeared.

The method of using a sequence of numbers to arrange the order of service (a popular method in everyday life), is applied to deal with this problem. Each time *fileevent* event handler passes a request to Eden, Tcl/Tk issues a service number to the event handler and suspends it for further reply. At the same time, Tcl/Tk passes the request plus the service number to Eden. When Eden processes a numbered request, it replies to Tcl/Tk with the service number. Once the replied number is equal to the issued number of an event handler, Tcl/Tk releases the suspended *fileevent* immediately. The fileevent in the server is then available to reply to the client with a confirmation. After receiving this confirmation, the fileevent in the client is also released. In other words, a synchronous communication between the client and the server is achieved. The number on each message square in WQ and EQ (shown in Figure 5-6) serves to illustrate how this concept of queued service number operates.

5.2 Interaction Modes in dtkeden

As explained in earlier chapters, the human factor is arguably a critical issue within software system development (SSD) [LR98, Som95, Flo87]. Interpersonal interaction amongst participants is one of the main resources for guiding and shaping the process of SSD and is therefore a key factor in determining the success or failure of a SSD project. It is widely recognised that inefficient and ineffectual interaction between participants is one of the major sources of confusion and error in SSD [STM95].

In dtkeden, interaction between modellers is achieved through networked computer models whose architecture and mechanism for network communication have been discussed in the previous section. Within such a computer-based distributed modelling environment, the interaction between modellers is computer-mediated and may involve no face-to-face interaction. Within the computer-mediated interaction, modellers cannot necessarily look at each other or use verbal or body language for interaction. They may not know each other. Instead, their networked computer models become the communication medium for the interaction between modellers. The visualisations of these models represent the construals of the modeller. In such interaction, each modeller 'speaks' through changing their computer model. Such a change is passed through the communication network and affects the computer models of others (the 'listeners'), so as to 'tell' them what the speaker is thinking.

Computer-mediated interaction is not subject to the same temporal and spatial constraints imposed by face-to-face interaction, but is recognised to be less effective and less socially rewarding [GK94]. On the other hand, compared with paper-based interaction, computer-mediated interaction provides modellers with active assistance in searching, understanding and creating knowledge in the course of co-operative problem solving processes [Fis91, DS97]. Indeed, it is hard to say which mode of interpersonal interaction is best suited for modellers, since real-world situations vary considerably.

Although in technical respects the architecture of dtkeden is based on the framework of DEM, both E-modelling (which is concerned with the interaction between modellers acting as external observers) and I-modelling (which is concerned with the interaction between agents enacted by A-modellers as internal observers) are supported by the current version of dtkeden. Both kinds of modelling require modellers to interact with each other for shaping agency and exploring the mutual understanding between modellers in a distributed environment. As pointed out by D. Sonnenwald [Son93, Son96], diverse interactions between group members in the design process are required in order to develop a comprehensive understanding of design and facilitate multiple exploration of knowledge. Sonnenwald identified a variety of roles and interaction networks for intergroup and intragroup members in each phase during the design process in order to highlight the diversity of interaction styles between all modellers. In the context of a large system, the architecture of the interaction among all modellers from multiple disciplines, domains and individuals can become exceedingly intricate [Son96]. One possible way to support such interaction is to decompose it into a number of small group interactions. Each small group interaction could possibly be supported by a distributed computational environment such as dtkeden. Figure 5-7 illustrates how such decomposition could be based on the framework proposed for DEM[9]. It should be noted that the intergroup communication between intergroup stars (in D. Sonnenwald's terms in [Son93]) is not yet explicitly supported in dtkeden.

[9] This illustration is based on D. Sonnenwald's work on intergroup communication in the planning phase among intergroup stars in the user, designer and developer group [Son93, p. 63].

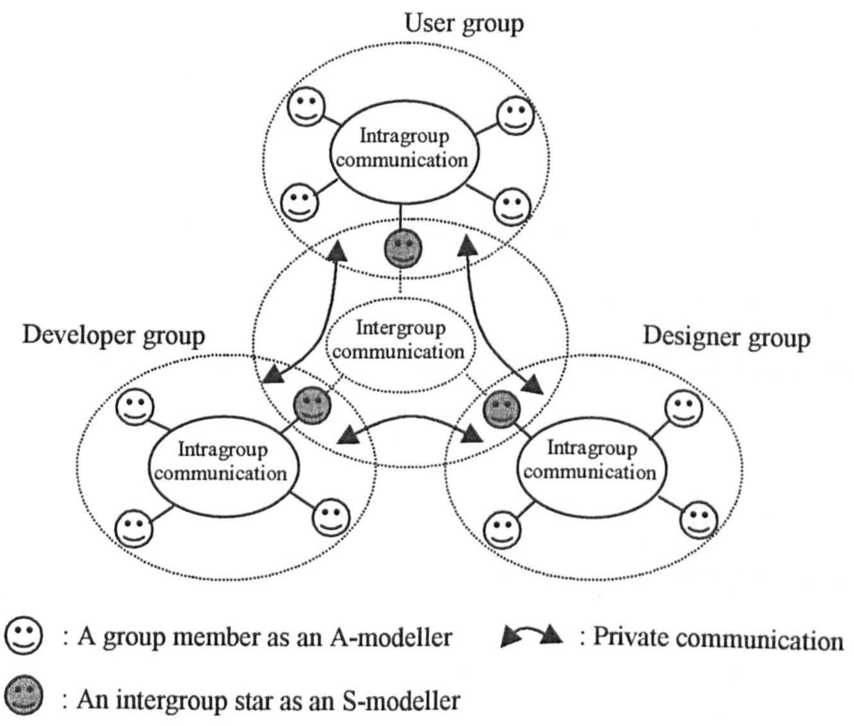


Figure 5-7. Decomposing large group communication into small group communication on the basis of the DEM framework

To support the interaction between modellers in forms of I-modelling and E-modelling in a distributed environment, four interaction modes for the S-node[10] have been implemented in dtkeden: broadcast, private, interference and normal.

The *broadcast* mode is the most primitive style of interaction between modellers. Its broadcasting mechanism is established by means of the S-node performing the role of a message-transferring centre. According to the star-type logical network configuration that is implemented in dtkeden, each message sent from an A-node must first come to the S-node. If the S-node at that moment is in the broadcast mode, the arriving message will be automatically broadcast to all other A-nodes, and this will consequently change the visualisations of the computer models at these A-nodes. The modellers at these A-

[10] In the current version of dtkeden, the interaction mode between A-modellers is established only in the S-node since it is a message transferring centre (see Section 5.1). Each A-node only sends and/or receives messages, but does not transmit any message. In addition, an A-modeller can change the interaction mode, if the necessary privilege is given.

nodes then construe this change and take this into account in their own computer models when invoking further interaction.

In as much as modellers co-operating in the broadcast mode are explicitly informed of changes, they can be regarded as being in an electronic group meeting without video and audio support. They share all messages sent out from any modeller by means of propagation, and typically interact with each other in an iterative manner in order to reach consensus. Interaction of this nature is common in the interaction between group members, such as in most inter- and intragroup interactions during the design process proposed in [Son93]. This mode can be used for developing a system, such as a multi-user game, that requires a shared environment for supporting the interaction between its users. (Examples are given in Section 6.3.)

The *private* mode provided by dtkeden supports a one-to-one interaction between the S-node and an A-node. As D. Sonnenwald observed from empirical studies, in many cases, one-to-one interaction plays an important role for managing different perspectives in a group [Son93]. To support this one-to-one interaction, the private mode provides each modeller at A-nodes with a private interaction channel to the S-modeller at the S-node. In the private interaction mode, in contrast to the broadcast mode, no message will be propagated to the other A-nodes. Since it is possible for more than one such private channel to exist in parallel, this private mode is also suitable for many-to-one modelling environments, such as is required in a system monitoring student learning (see Section 6.3). It should be noted that the interaction between two A-nodes is achieved through a built-in procedure providing A-node-to-A-node-via-the-S-node communication, as discussed in the previous section (Section 5.1).

The *interference* mode is a very useful mode for modelling a situation through 'what if' experiments. It allows the modeller at the S-node, acting as the superagent, to interfere directly in the interaction between modellers. Before being serviced, each message arriving at the S-node is displayed on the input window of the modeller at the S-node. The superagent can exercise discretion over how the suspended message is processed by changing its content. The 'what if' experiment enables the superagent to explore and experience an unfamiliar context for the interaction between modellers. This often leads to unexpected situations that can provide modellers with surprising and enriching discoveries.

Interference mode is of particular interest for modelling the phenomena of the real world, where many-to-many interactions are the norm and singular conditions require the intervention of a God-like superagent (cf. [Bey98]). It is also applicable to the case in which the interaction between modellers necessitates multi-faceted exploration in order to improve comprehension among modellers at the S-node and A-nodes. For instance, the need for such mutual exploration has been identified by Sonnenwald in connection with the creation of innovative artefacts [Son96]. In addition, high-level managerial interference in the interaction between lower-level personnel in an organisation is also a suitable application for the interference mode.

The default interaction mode in dtkeden is called *normal* mode. In this mode, the interaction between modellers is mediated by the computer with reference to specified privileges of modellers to access observables. For E-modelling, the access privileges of modellers typically are given so as to reflect management control and social relationships between modellers. For example, in a design process, users are often not allowed to change those observables associated with implementation, and designers from different groups can be restricted to access different observables (cf. [ABCY94]).

Normal mode is particularly significant for I-modelling, where modellers are required to interact with each other in the roles of agents. As explained in Section 4.2, their interaction, which is invoked in the form of pretend play, should reflect the agency of the individual agents whose roles they are playing, as defined by their privileges to change observables.

A privilege to act is specified by a guarded action such as is used in the *protocol* part of an LSD account. The generic form for such a guarded action is as follows:

guard → action

This guarded-action can be interpreted as asserting that the agent has context-dependent privilege to undertake the state-changing *action* if the *guard* is true. A privilege to act implies that certain privileges to access those observables that are involved in the action are also needed. That is to say, in order to exercise its agency, the agent must be able to gain access to the observables involved in the *guard* part for observation and to those in the *action* part for modification. The agency of an agent is suspended if the agent does not have the appropriate access privileges.

Access privileges are significant for the interaction between agents. They provide a richer model of context-dependent agency than guards alone supply. As explained and illustrated in Section 4.3, agency can appear and disappear as the context surrounding an agent changes. Taking the context of agency into account helps to make the computer model resemble the referent in the real world more closely, as is needed in order to clarify the modellers' understanding.

In previous work on EM, the access privileges to observables by an agent have been described in the *oracle* and the *handle* parts of an LSD account respectively. The design of tkeden does not take this description into account. This is partly because the core part of tkeden is essentially observable-based. The agent concept is only partially represented by triggered action in tkeden.

A more important reason is that in such a stand-alone modelling environment the unique modeller of EM, being a superagent, is empowered to access all observables.

However, when tkeden is exploited in a distributed environment, the agents described in an LSD account are distributed on different workstations. Any observable could be accessed by the superagent S-modeller but also by A-modellers acting as agents via a communication network. In this case, the access privilege to an observable by an agent should be considered carefully. After all, in the real world, observables are often not open to all agents. For example, in a two-player draughts game developed in dtkeden, it is not appropriate for a player to be able to change the positions of his/her opponent's pieces except in a situation in which the player has captured the opponent's piece.

To deal with access privileges to observables by agents, a system agent called *LSDagent* has been implemented in dtkeden by the author. The current version of dtkeden can take two kinds of access privilege to observables into consideration: *oracle*, to specify which observables an agent can access for observation; and *handle*, to specify which observables an agent can conditionally change. When an agent is attempting to access an observable for observation or modification, the LSDagent checks whether or not this agent has the necessary permission to access this observable. Without valid permission, an agent is not allowed to observe and/or modify an observable.

The author has extended dtkeden to enable the S-modeller to specify agent's access privileges to observables that are described in the oracle and the handle parts of the LSD account [Bey86]. This extension involves introducing an LSD-based notation into dtkeden to provide for the declaration and cancellation of access privileges. The details of using the notation are given in Appendix 5-A. With this LSD-based notation, the S-modeller is empowered to declare or remove the privileges of agents to access observables with reference to the LSD account. Accordingly, agents' access privilege to observables can be established in the computer model of the S-modeller. With the specified access privilege, the LSDagent in the computer model of S-modeller can verify the permission of an agent to access observables, and can propagate the change of an observable that is changed by an agent to other agents who have access privilege to the observable for observation.

In the real world, an agent's access privilege to an observable is not always persistent but can be mobile and volatile. Considering the draught game example above, when a player's piece jumps over the opponent's piece, the player can remove this piece from the board. In this case, the player gains temporary control over the position of his/her opponent's piece. Similarly, after this piece has been removed from the broad, there is no significant sense in which either player can change its position again. In other words, an agent's access privilege to observables changes dynamically. Functions for the purpose of changing the access privilege on-the-fly have been implemented in dtkeden. Their details are also given in Appendix 5-A.

It should be noted that the implementation for supporting normal mode is different from the implementation for the other three interaction modes in dtkeden. The normal mode is implemented in the Eden level, since agents' access privileges to observables are associated with the internal representations of each observable and each agent, and their interrelationships. The other modes all concern the mechanism of message passing (that is, the protocol for the transmission of a definitive script) and can be implemented in the Tcl/Tk level.

With the star-type architecture that is implemented in dtkeden, an interesting issue emerges when multiple modellers are involved in a distributed environment: which modeller should be at the S-node? As explained earlier (Section 5.1), the role of the modeller at the S-node is to be the superagent transferring messages that occur in the interaction between modellers at A-nodes and potentially interfering with these messages. Thus, it is appropriate for the modeller at the S-node to play a more powerful role, such as that of the manager, the intergroup star, or intragroup star, in order to highlight the modeller's characteristics.

In addition, the interaction between modellers is not always in one of the above, specific interaction modes. It can instead be switched dynamically between these modes by programming or choosing different menu items in the input window at the S-node. In practice, the interaction between modellers can be more subtle than these four interaction modes can provide, even in combination. Further improvement is suggested in Section 8.3 for future work.

5.3 Adaptable Reuse in `dtkeden`

Over the past decade, the reusability of software components has become widely publicised in SSD, since it has been proved to be helpful in reducing development and maintenance costs and increasing productivity [Som95, Pre97, Pau97]. The traditional approach to software reuse usually seeks to develop software components for complete reuse without modification. This demands that the reused component be completely fitted into a new solution domain. In practice, it is very hard to find two solution domains that are exactly the same [PF87]. This difficulty leads to a major challenge of putting reuse into practice and integrating it into software development processes.

As described earlier (Section 2.2), each definition of an observable in the computer-based model is the modeller's construal of the observed world. In the context of the A-modeller, the definition is unique. However, the uniqueness may become problematic when it is sent to the computer-based model of the S-modeller. This is because within this model there may exist more than one definition for the same observable. These definitions indicate that the same observable can be construed in different ways by different A-modellers in a distributed environment. For example, an A-modeller X may define an observable M as 'M is A+B', but another A-modeller Y may define the same observable M as 'M is A+C'. Obviously, a problem of inconsistency between two definitions of the observable M arises, when both are sent to the S-modeller.

In a conventional computational framework – and indeed in the stand-alone environment of tkeden – such inconsistency is not allowed, since a variable cannot have two different definitions (or internal representations) at the same time. A mechanism by which a later definition overwrites an earlier one may be invoked. In some context, further interaction between X and Y or an appeal to arbitration will be invoked to eliminate the inconsistency.

However, in the real world, the elimination of inconsistency is not absolutely necessary. The coexistence of different definitions is sometimes needed, for example, for the purpose of distinguishing individual differences in perception. Taking the same example above, to clarify the difference between the two definitions given by A-modellers X and Y, the S-modeller may want to display the observable M's content by evaluating the two definitions simultaneously in various situations. In this case, the need to keep both local definitions co-existing in the model of the S-modeller becomes clear. In practice, even in a local model, it is sometimes necessary to separate others' definitions from the modeller's own for the same observable.

Moreover, from the perspective of distribution, it is better to prevent the contexts of all modellers from becoming mixed up together in order to keep track of the individual context. To achieve this, all definitions given by modellers must independently co-exist, otherwise the overwriting mechanism will be invoked to eliminate the inconsistency between different definitions of the same observable. However, the internal representation of a variable in a program can only be described by a definition at one particular moment, so that it is not possible to give an observable multiple definitions that co-exist at the same time. The notion of *virtual agent*[11], which provides a means to

[11] For convenience, the virtual agent is usually given the name of the A-modeller providing these definitions, though it is not necessary to do so.

associate a family of definitions with an agent, is introduced into dtkeden in order to cater for the demand for co-existent definitions.

Conceptually, virtual agency provides a way of attaching a definitive script to a particular observer, typically so as to represent the personal perceptions of that observer. Ideally, the association between a script and its observer should be defined and manipulated as a form of dependency. For instance, it would sometimes be convenient for one agent to hand over a script to another. In practice, there are serious technical difficulties in implementing such a feature in tkeden. This means that virtual agency is managed in dtkeden in a procedural fashion.

Definitions in a dtkeden script are associated with a virtual agent according to the context in which they are introduced. If no virtual agent context has been declared, definitions are in the *root context*. As soon as a virtual agent is declared, dtkeden shifts its current context to the context of this virtual agent and then localises each subsequent definition until another new context is required. The *localisation* of a given definition means that each observable used in the definition will be renamed by appending the virtual agent's name to its original name. The renamed definitions then can be distinguished from those given by others. For example, with this virtual agent notion, if virtual agents are given the names of their respective A-modellers, the two above definitions for the observable M can be localised as 'X_M is X_A + X_B' and 'Y_M is Y_A + Y_C'. Both these definitions are present in the computer model of the S-modeller.

The virtual agent concept is helpful for creating a distributed model. For the S-modeller, the different definitions of the same observable associated with different contexts can co-exist in the same model, and can be easily accessed by declaring the context of a virtual agent. A definition made by the A-modeller X in his/her local context is transmitted to the S-node and, through localisation, is interpreted as if it were introduced at the S-node in the context of a virtual agent X. This mechanism allows each

A-modeller to use observables in his/her computer model without needing to take special steps to guard against ambiguity. An A-modeller's local context is independent of the localised context in the S-modeller's model, because the localisation mechanism is invoked after the message is passed to the S-modeller and before it is internally represented in the S-modeller's model. Appendix 5-B illustrates the syntax of using a virtual agent to shift context in dtkeden.

In effect, given a virtual agent and a set of definitions (that is, a definitive script), the localisation mechanism can be regarded as a mechanism for generating a new definitive script associated with the context of a virtual agent. The generated definitive script with renamed observables is different from the original one so that dtkeden will store all these new definitions and maintain their dependency automatically. It should be noted that there is no defined dependency between both definitive scripts after localisation.

Hence, with the virtual agent notion, a definitive script can be used as a pattern to generate different definitive scripts associated with different contexts. This conclusion is very valuable for dtkeden when the reusability of a definitive script is taken into account.

In reusing a definitive script, an ontological problem concerning observables in EM and DEM emerges. Following [Bey98], it appears that each observable in a definitive script must correspond to a characteristic of the modeller's external environment [Bey98, SB98]. Taken at face value, this means that each observable must be conceptually subject to an object in the environment, including the observed world and the computer model; otherwise the modeller cannot observe it[12]. According to this rule, for example, the corner of this table and the status of my bank account, are observable, but the corner of *a* table

[12] In fact, [Bey98] involves an extended discussion of how the notion of an observable in EM embraces entities that are quite different in character from the physical observables of commonsense. To observe an observable in EM and DEM need not mean to physically "see" this observable. Instead, it can be used in a broader sense to refer to entities that are typically construed, for example, as imaginary.

and the status of *a* bank account are not observables because no particular object is identified for the modeller's observation.

By this interpretation, an observable in EM can only be described in the details of the modeller's observation. It is quite inappropriate to reuse an observable of this kind because it has a specific referent. This is the reason why most systems developed by EM contain a huge numbers of definitions. For example, Figure 5-8(c) illustrates several observables, called '1st door', '2nd door', ..., and 'nth door'. Each consists of a family of observables with similar characteristics. The similarities between them provide no help for reuse due to their specific context. In the same manner, in fact, no description of observables corresponding to the observed world can be reused as a pattern.

For the purpose of reuse, 'observables' that are not subject to a particular context corresponding to the modeller's environment are needed (cf. [GYCBC96]). A possible solution to this crucial problem of reuse in dtkeden is to use abstraction, which disassociates the significant characteristics of an object from any specific instance [Ber94]. Although abstraction can separate observables from their detailed context to serve the purpose of reuse, this separation has other implications that impose inevitable limitations on the scope and nature of reuse. Further details will be given later when adaptable reuse in dtkeden is compared with complete reuse based on an abstract data type.

A more appropriate solution, devised and implemented by the author in dtkeden, is to create a new kind of observable, called a *generic observable* (GO), for the purpose of reuse (cf. footnote 11). Unlike those observables that correspond to real world objects in the modeller's external environment (as described in Section 2.2), GOs are created to correspond to the modeller's experience, which is inside the modeller's mind and emerges from repeated description of certain observables with the same characteristics. For example, after repeatedly describing a number of doors with the same characteristics,

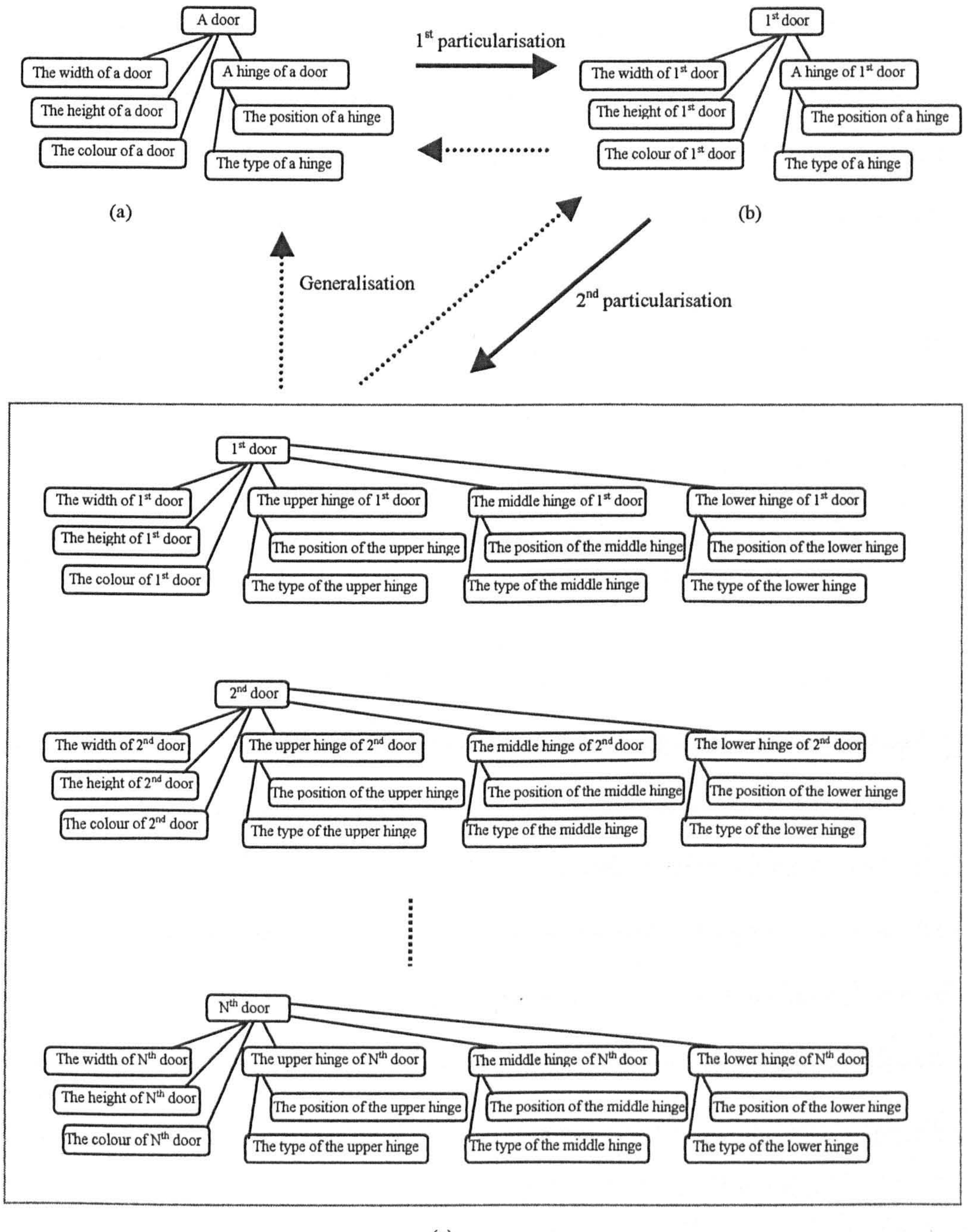

A door
The width of a door
The height of a door
The colour of a door
A hinge of a door
The position of a hinge
The type of a hinge
(a)
1st particularisation
1st door
The width of 1st door
The height of 1st door
The colour of 1st door
A hinge of 1st door
The position of a hinge
The type of a hinge
(b)
Generalisation
2nd particularisation
1st door
The width of 1st door
The height of 1st door
The colour of 1st door
The upper hinge of 1st door
The position of the upper hinge
The type of the upper hinge
The middle hinge of 1st door
The position of the middle hinge
The type of the middle hinge
The lower hinge of 1st door
The position of the lower hinge
The type of the lower hinge
2nd door
The width of 2nd door
The height of 2nd door
The colour of 2nd door
The upper hinge of 2nd door
The position of the upper hinge
The type of the upper hinge
The middle hinge of 2nd door
The position of the middle hinge
The type of the middle hinge
The lower hinge of 2nd door
The position of the lower hinge
The type of the lower hinge
Nth door
The width of Nth door
The height of Nth door
The colour of Nth door
The upper hinge of Nth door
The position of the upper hinge
The type of the upper hinge
The middle hinge of Nth door
The position of the middle hinge
The type of the middle hinge
The lower hinge of Nth door
The position of the lower hinge
The type of the lower hinge
(c)

Figure 5-8 An example of particularisation and generalisation

the modeller can generalise a GO and give its description as illustrated in Figure 5-8(a) or (b).

The *generalisation* process, as shown in Figure 5-8, is a process in which similar descriptions are repeatedly given for certain observables with the same characteristics, so that the modeller is able to create GOs and give their descriptions in response to the emerging experience. In this sense, the created GOs correspond to the modeller's practical experience rather than to particular objects in the modeller's external environment. Since the experience is obtained from the repeated description in practice, it is inappropriate to view this generalisation process as a form of abstraction, which is completely independent of practice. The key point is that the experience emerging from the generalisation process is shaped by the character of the modeller, as determined by his/her knowledge, intelligence and past experience. This is in contrast with the concept of abstraction in which personal characteristics are not taken into explicit account.

Since GOs are derived from the modeller's practical experience in a generalisation process, no guarantee can be given that they will emerge and be of adequate quality. Instead, they are contingent, volatile and unpredictable. The modeller, observables themselves and the generalisation process are all factors eligible to affect the creation of GOs. For example, a novice modeller may take a long time to form practical experience underlying the generalisation of GOs, but an experienced modeller may not. Also, the qualities and the content of the GOs' descriptions as generalised by both kinds of modellers could be very different. Figures 5-8(a) and (b) shows that different generalisation processes, from (c) to (a) or from (c) to (a) via (b), are produced, and this leads to the creation of different GOs, such as 'a door' and 'a hinge of my door'.

In this sense, the term 'reuse' in dtkeden in effect refers to the reuse of practical experience. The reuse of experience is one of the fundamental ways in which human beings cope with problems in everyday life. There are many examples: the driver finding his way round a city he has not visited before, the teacher teaching a new class, and so on. In such cases experience provides human beings with a great deal of help in solving situated problems.

Therefore, it is clear that the experience-oriented reuse of a definitive script is informed by the essential characteristics of empirical modelling, such as subjectivity and situatedness. This means that on the one hand, reuse in dtkeden can become uncertain, risky and not persistent. On the other hand, these characteristics do offer benefits, such as usability and adaptability, as suggested by P. Ness in [Nes97].

After creating a GO, its description, that is, a definitive script, can be used as a reusable pattern by combining it with a virtual agent. A new definitive script can be generated by localising the pattern. In order to use this definitive pattern over and over again, it is convenient to store it in a file. Each time the modeller wants to reuse this pattern, he/she just needs to include this file in his/her computer model and specify a virtual agent for localisation.

So far, only the mechanism by which a virtual agent can be explicitly specified from the input window in dtkeden has been described (see Appendix 5-B for details). Although the way in which a virtual agent is specified does not influence the semantics of a definitive script in reuse, a way to specify a virtual agent without the explicit involvement of the modeller is also helpful. This is because the context of the virtual agent may have to be determined and established in an automatic fashion. For this purpose, the name of a virtual agent can be specified implicitly by a string variable. With this feature, the modeller can store a definitive pattern in a file with an anonymous virtual agent as its header. In this way, the context of a GO can be dynamically determined.

With these mechanisms in place, the reuse of a definitive pattern in dtkeden becomes more flexible and applicable. As soon as an observable similar to the characteristics of a created GO is observed, the definitive pattern of the GO can be exploited for reuse. To reuse the definitive pattern, the floating context of the GO has to be particularised to the context of the observable. This *particularisation* can be achieved by three steps: specifying the undetermined virtual agent (identification), retrieving the content of a GO (instantiation), and finally localising the retrieved content to the new context of this observable (localisation).

The *identification* step can be accomplished by the modeller by declaring a virtual agent interactively via the dtkeden input window, or by invoking a suitable procedure or function to change the virtual agent context. The *instantiation* step creates an instance of a GO by introducing a definitive pattern into the context established in dtkeden in this way. Following this instantiation, *localisation* of the definitive pattern is the process that creates an appropriate syntactic variant of the definitive script in the computer model. In this way, the definitive pattern of the GO is reused.

Experience is adaptable. Certainly, experience is not only reused for the same situation, but also for similar situations. For a driver, no driving situation is exactly the same as any previous driving experience. Too many factors, such as different roads, the presence of different cars and drivers, different traffic system, etc. add new dimensions to the driving situation. However, a driver still can drive in such different situations and does not need to relearn how to drive. This is because human beings can adapt past experience to a new situation, and indeed all past experience was originally derived from previous different but similar situations. Reusing experience is apparently more significant in everyday life than reusing well-defined but unchangeable program codes or even abstract components, such as specification and design [Som95].

Traditionally, there are two ways to reuse software components: black-box reuse and white-box reuse. The former refers to the reuse of well-defined components without modification, but the latter demands further change to the reused components. In practice, black-box reuse is of limited use, because it is very expensive to develop reusable components that are suited to a variety of situations without change [PF87, Pri93]. So far this kind of reuse is only applied in developing relatively low-level components, such as procedures for supporting the GUI (graphical user interface) and database framework.

Regarding white-box reuse, the difficulties mainly stem from the extra efforts for the necessary modification. This is because changing components can be formally thought of as engaging in parts of, or even the whole of, a software development process. The effort needed for this engagement can to a large extent offset the benefit of reuse [Pau97]. In addition, the trend in white-box reuse is towards parameterisation and built-in adaptability [Pri93]. This obviously makes white-box reuse more flexible and applicable, but it requires further formalisation of the intended components prior to reuse. The prior formalisation usually must be invoked before run time and accomplished in a context-free manner. As a result, problems similar to those of using the concept of abstraction for reuse emerge. For example, the extent to which inheritance, devised for white-box reuse in object-oriented programming, supports software reuse is subject to controversy [GHJV95, Som95].

By contrast, the reuse of definition patterns by adaptation is encouraged in dtkeden. In comparison with white-box reuse, there is less difficulty in changing the reused components (that is, the definitive patterns) in adaptable reuse, as proposed here. In the light of definitive programming, as described in Section 2.4.2, adapting the particularised definitive scripts is far simpler than the software development process in conventional programming. If any part of the reused definitive pattern is not well suited to the new context, the modeller can simply give new definitions to replace the unsuitable definitions. As described above, experience is adapted to a new situation in everyday life in much the same manner. For example, in reusing the definitive pattern of a GO called 'an action button' to produce a new action button, the caption of the latter has to differ from that of the original one. After localisation, by means of definitive programming, the modeller can easily change the localised definitions of observables associated with the caption simply by introducing new definitions.

In principle, stored definitive patterns can be reused over and over again to produce different definitive scripts. They can also be reused for other systems when the modeller faces a similar situation. This is one of the main reasons why experienced modellers usually spend less time accomplishing the modelling task than novice modellers. In addition, definitive patterns stored by a modeller also can be reused by other modellers. The method of reusing others' patterns resembles the way in which one person reuses another's experience to solve similar problems. In fact, this method of reusing past experience is very common among programmers. Very rarely does software system development begin from scratch [Pot93]. Many programmers have experience of reusing parts of previously developed program codes by pasting them into a new context with or without modification. It should be stressed that reusing program codes should not be regarded as simply reusing certain functionalities embedded in the codes; rather it involves the reuse of the programmer's past experience.

By means of the virtual agent concept and the generalisation-particularisation process, the modeller can create GOs and reuse their definitive patterns to reduce the size and complexity of a system. This benefit has been found in several practical case studies. For example, in an electronic circuit laboratory project for education, a GO called 'a painted button' is created and its definitive pattern is reused to create a further 150 similar buttons for storing diverse circuit graphs [Dor98, She98]. Another example is a classroom simulation project in which its source codes are rewritten by means of adaptable reuse proposed here (detailed in the next chapter). Each project reduces the total size of the model by more than 60%. Similarly, the case study of the simulation of a railway accident, discussed in the next chapter, also shows how a GO named 'a train' can be created and its definitive pattern reused to create a number of trains in the animation (see Appendix 6-B for more details).

Like the GO notion, one of the most important notions for reuse in software development is that of abstract data types (ADTs). The ADT is a very fundamental notion for modern, for example, object-oriented programming [Boo94]. In contrast to the standard data types provided by a programming language, an ADT is a programmer-defined data type whose logical behaviour is defined by a set of values and a set of operations on those values [Azm88, Cle86, Ber94, DW96, Wei99].

The most important notion behind ADTs is that of 'abstraction'. This term refers to a process that discards many details and emphasises only the 'main features' of interest at a particular 'level of concern'. It is used in many areas of computer science to reduce the complexity of tasks to a manageable level. C. Hoare suggested that "abstraction arises from a recognition of similarities between certain objects, situations or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" [DDH72, p. 83, quoted from [Boo94, p.41]].

The essential idea behind ADTs is to separate the specification of an ADT from its implementation; hence, an ADT is a mathematical model [Azm88]. Nowhere in the definition of an ADT is there any description of how the set of values is represented and the set of operations is implemented. Encapsulation of this nature allows a separation of concerns: the user of ADTs can use the data type, but does not need to know how the data type is implemented. The specification of an ADT becomes the sole interface for both the people writing applications and the people who implement the abstract data type in a computer program.

Although both are based on the '*recognition of similarities*', ADTs and GOs are very different, as can be seen from the summary in Table 5-1. The most obvious difference is that an ADT is abstraction-based, but a GO is experience-oriented. As explained above, abstraction leads to the separation of specification from implementation.

On the one hand, this separation has the advantage of facilitating reuse, since the specification of an ADT can be used over and over again, and its implementation details can be considered only once. Moreover, once any value or operation in the specification of an ADT is declared, its associated implementation can also be reused without worrying about its details. On the other hand, such a strong separation of concerns means that, once a specification is given, only *prescribed* values and operations are available for reuse. Any attempt to adapt prescribed elements, including values and operations, or to access unspecified elements, becomes problematic [BE94, McG92, OS93]. Although this strong typing[13] feature makes programs manageable, easy to debug and more effective, it also to some extent discourages reuse [Ous98]. In these contexts, the reuse of ADTs can be viewed as creating instances from standard components whose descriptions are well-defined and unchangeable[14]. In other words, reuse can only reach the level of instantiation.

	GOs	**ADTs**
Typing	Typeless	Strong/weak
Form	Experience-oriented	Abstraction-based
Purpose	Reuse a definitive pattern for new observables; adaptable reuse	Separate specification from implementation; instantiate prescribed components
Character	Context-dependent	Context-free
Defined time	Prior/Run time	Prior
Usage	Situated, adaptable	Inflexible

Table 5-1. A comparison of GOs and ADTs

[13]The term "typing" is used to "refer to the degree to which the meaning of information is specified in advance of its use" [Ous98].

[14]In object-oriented programming, an alternative to reusing well-defined components is to reuse through inheritance, meaning that it is possible to overwrite the inherited description [GHJV95]. Though the latter is increasingly common in object-oriented programming, its suitability is controversial [GHJV95, Ous98, Som95]. Note that even when overwriting is allowed in an Object-oriented context, it still has to be done before run time.

In dtkeden, a GO is created to reflect the practical experience obtained in describing certain observables with the same characteristics. Its definitive pattern is not derived by discarding details, which – according to the concept of ADT – should be ignored prior to the process of abstraction: instead, it emerges from recognised similarities. For example, after describing a number of buttons, common features, such as their appearance and the action triggered by clicking the mouse left button, could be recognised and grouped together as the definitive pattern of a GO called 'an action button'. In this sense, the definitive pattern is shaped and generated through the practical, situated experience of the modeller in recognising these similarities between buttons.

No separation of specification and implementation concerns is possible in the context of such reuse in EM. No so-called specification can be given before the emergence of this practical experience, and no implementation details can be ignored during the generalisation process that leads to a GO, that is, by the work practice of 'recognising similarities'. This is because a GO is created in a situated fashion and on the basis of practical experience. This perspective on reuse is consistent with the concepts of 'experience in action' [Bur91, Hen96, LW91] and of Piaget's 'knowing-by-doing' [diS88, Puf88].

In summary, the mechanism of particularisation in dtkeden is to a large extent a form of reuse of the experience implicitly embedded in the description of a GO[15]. After generalising recognised similarities to the definitive pattern of a GO, the modeller is able to reuse the definitive pattern in the same way as experience is reused in everyday life when similar situations appear. Particularising the context of a GO to the context of an observable – specifying the virtual agent and including the definitive pattern into dtkeden – can be viewed as connecting past experience with the current situation.

[15]This is not to claim that definitive patterns exactly represent the experience obtained from the generalisation process. After all, experience cannot be totally represented in the form of language [Haml78, Jam96].

Localising the definitive script, in effect, can be regarded as a way of embodying reused experience into the computer model. The generalisation-particularisation process proposed here for adaptable reuse is, to a great extent, in line with the experiential learning process [Bur91] and the situated learning process [LW91]. The latter emphasises learning from practical experience and then applying it to future actions. Both concepts have been widely used in applied science, for example, in nursing and in training apprentices in work practices.

Appendix 5-A: The use of LSD notation

There is support for the LSD notation at the S-node in dtkeden when it is operating in 'normal mode'. Scripts in the LSD notation define agents' access privileges to observables, and a set of special dtkeden procedures is available to perform the same functions. Only some parts of the LSD notation are implemented in dtkeden - *oracle*, *handle* and *state*. Scripts in the LSD notation should start with '%lsd' and end with the name of another notation, such as '%eden'. The syntax is as follows:

```
agent agentName
oracle observableName[, observableName]
handle observableName[, observableName]
state observableName[, observableName]
remove LSDType observableName[, observableName]
```

The agent statement associates all the following statements with agent *agentName*. The oracle, handle and state observable lists contain observable names associated with the privileges to access the observables by *agentName*. The following example illustrates how to use the LSD notation.

```
%lsd             <----- declare the use of LSD notation

agent  xxx       <----- declare the agent name
oracle  a, b, c  <----- add to agent xxx the oracle agency for observables a, b, c.
handle a, m      <----- add to agent xxx the handle agency on observables a, m.
remove oracle b  <----- remove from agent xxx the oracle agency for observable b.

agent yyy        <----- declare a new agent name
oracle m, c      <----- add to agent yyy the oracle agency for observables m, c.
handle b, m      <----- add to agent yyy the handle agency for observables b, m.

%eden            <----- back to EDEN
```

After receiving the above description, the LSDagent creates the links between agents and observables. The LSD description can then be checked from the 'View LSD Description' subitem in the pop down menu of the 'View' item of the server's input window menu.

There is another way to configure the link between an agent and an observable. The commands 'addAgency', 'removeAgency' and 'checkAgency' can be used dynamically to manage the access privileges of an agent. Their syntax as follows:

```
addAgency("agentName", "LSDType", "observableName");
example: addAgency("yyy", "oracle", "b");

removeAgency("agentName", "LSDType", "observableName");
example: removeAgency("yyy", "handle", "m");

checkAgency("agentName", "LSDType", "observableName");
example: checkAgency("xxx", "handle", "w");
```

In these syntactic forms, the parameter *LSDType* is one of three basic keywords in the LSD Notation: oracle, handle and state. The first two commands are used as procedures in Eden, and the last command is used as a function returning a value of TRUE or FALSE. If TRUE (actually an integer of value 1) is returned by the command checkAgency, it means that the specified agent agentName has the identified privilege LSDType for the specified observable observableName. If the agent does not have this privilege, a FALSE (0) value is returned.

The current state of the S-node's interaction modes can be examined and changed via commands in the dtkeden input window. For the management of agents' privileges in 'normal mode', the LSD Agent in dtkeden creates and maintains three lists for each observable. For example, if the list of *oracle* privileges for an observable mmm is [xxx, yyy], then both agents xxx and yyy have the *oracle* privilege for the observable mmm. These three lists can be checked by using the function symboldetail("*mmm*"), which returns the privilege details associated with observable *mmm*. The format of the EDEN list returned is:

```
[mmm,type,defn,l1,l2,[oracle_agents],[handle_agents],[state_agents]]
example:
    writeln(symboldetail("a"));
    [a,formula,b+c;,[b,c],[],[EVERYONE],[sun,carters],[EVERYONE]]
```

For any observable in any state, an *oracle_agents* list, *handle_agents* list or *state_agents* list is either:

- empty - [] - no agent has the associated privilege;
- contains special agent 'EVERYONE' - [EVERYONE] - every agent has the associated privilege;
- contains a list of agent names (not including EVERYONE) - [sun,carters] - with the associated privilege.

When creating a new observable, dtkeden refers to a system variable called 'EveryOneAllowed' in order to give the default access privilege for each new definition of an observable. The two settings are:

```
EveryOneAllowed = TRUE;
```

In this case, a default agent name called 'EVERYONE' will be set for every new definition, and the LSD Agent will automatically grant every agent open access privileges to redefine and observe all newly created observables.

```
EveryOneAllowed = FALSE;
```

In this case, no agent has any access privileges to observe or redefine the observable.

Appendix 5-B: Virtual Agents

The concept of a virtual agent is motivated by the treatment of observables in the private interaction mode in dtkeden (see Section 5.3). In this interaction mode, a definition is introduced at an A-node is transmitted to the S-node in a form that is syntactical modified to its originating agent. For example, if agent X introduces a definition 'a is b + c', a new definition 'X_a is X_b + X_c' is generated at the S-node.

The virtual agent mechanism allows the superagent at the dtkeden server to introduce definitions in a context *as if* they were being generated by an agent in a similar fashion. The mechanism can be used in any interaction mode in dtkeden. If no virtual agent context for definition has been declared, all definitions are in the *root context*. In dtkeden, a virtual agent is declared by a symbol containing two characters. This symbol is followed by an agent's name that can be specified explicitly by a string constant (an *explicit* declaration) or implicitly by a string variable (an *implicit* declaration). In the former case, dtkeden will use the string constant as the current virtual agent's name to establish a context associated with this name, but in the later case the content of the given string variable will be used. If a symbol is not followed by an agent name, it indicates a reset of context to the root context. According to the given symbol, localisation is performed by appending the virtual agent's name to each variable identified in a postfix or prefix form. So far, five ways to declare a virtual agent have been devised in dtkeden. Table 5-B shows the use of these various methods to localise the definition 'a is b+c'.

The last symbol shown in Table 5-B does not cause dtkeden to shift the current context to the specified agent's name, so its localised definition is the same as the original definition. Normally, it is used for agency checking by the LSDagent embedded in dtkeden, to examine whether the named agent has appropriate access privileges to modify these variables in the root context. In this example, the LSDagent will check if the declared agent has the access privilege *handle* on the observable *a*.

In addition, the symbol '~' is available to reference the root context from another context, that is, it can be used to declare a global observable. For example, the definition '~a is ~b+~c' will be localised as 'a is b+c', whichever context is used. Furthermore, in the current version of dtkeden, the virtual agent can be applied to the Eden, DoNaLD and Scout notations.

Symbol	Declaring a virtual agent	Type of declaration of the virtual agent	Localised definition
>>	>>X	Explicit (in prefix form)	X_a is X_b+X_c
><	X = "x" ><X	Implicit (in prefix form)	x_a is x_b+x_c
<>	<>X	Explicit (in postfix form)	a_X is b_X+c_X
<<	X = "x" <<X	Implicit (in postfix form)	a_x is b_x+c_x
>~	>~X	The context that is associated with the virtual agent 'X' is declared, but localisation is not invoked	a is b+c

Table 5-B. Different ways to declare a virtual agent

Chapter 6
Case Studies

In accordance with the main principle of EM – knowing-by-doing – the best way to understand the issues discussed in the last two chapters is to undertake practical case studies. In fact, some points of view underlying DEM – such as A-modellers acting as agents to shape agency – and some of the requirements promoting the features of dtkeden – such as synchronous communication, the virtual agent, generic observable, and situated agency – are motivated by practice. In this respect, case studies not only make it possible to experience and explore DEM and dtkeden, but also serve to enrich and expand both in surprising ways.

6.0 Overview

This chapter seeks to exemplify DEM and some features of dtkeden through practical research. Three case studies are provided.

The first case study (6.1) involves modelling a nineteenth-century railway accident. The modelling aims to demonstrate DEM discussed in Chapter 4 (4.2). Several computer models are constructed as artefacts in a distributed modelling environment (that is, dtkeden) in order to shape the agency of agents associated with the accident by means of the successive interactions between A-modellers, who act as these agents, and their models. The concept of pretend play proposed in the last chapter is illustrated by the example of shaping agency that allows two signalmen to change the telegraph needle's position for communication. Also, in the animation of this historic accident, the role of the S-modeller in setting diverse contexts for A-modellers and authorising the accessibility of each agent to observables is illustrated by examples.

Another case study (6.2) illustrates the use of the virtual agent concept described in the last Chapter (5.3). In Section 6.2.1, an ADM translator developed by S. Yung [Yun92] for translating an ADM model to an Eden model is reviewed. A particular problem concerning the Eden models generated by the translator is their readability, because many additional string handling symbols are introduced into the models. A new ADM translator for reengineering this translation has been developed by the present author. The new translator generates Eden models in the virtual agent form. Without the additional symbols, the generated models are much easier to read and maintain.

In addition, Section 6.2.2 provides two examples of adaptable reuse, an important application of the virtual agent concept as discussed in Section 5.3. Within both examples, reusable definitive patterns with a virtual agent are established through the generalisation process (explained in 5.3). These patterns can be particularised to create the required definitive scripts in accordance with the context of the specified virtual agent in run time. Both examples highlight the benefits of using the virtual agent concept to reduce the size of the developed systems and to create definitive patterns for adaptable reuse dynamically, whilst the model is executing.

Section 6.3 illustrates the use of the interaction modes implemented in dtkeden (see Section 5.2). Unlike the two case studies in the previous sections, which make use of the normal interaction mode, the third case study introduced in this section shows the application of the other three interaction modes commonly used for concurrent modelling.

Two systems that have been created by extending pre-existing stand-alone versions in tkeden are given as examples.

6.1 A Railway Accident in the Clayton Tunnel

Table 6-1 describes a railway accident that occurred in the Clayton Tunnel near Brighton in 1861 [Rolt82]. The accident has been studied by using DEM and dtkeden in order to

The Clayton Tunnel Disaster **August 25th 1861**

Three heavy trains leave Brighton for London Victoria on a fine Sunday morning.

They are all scheduled to pass through the Clayton Tunnel---the first railway tunnel to be protected by a telegraph protocol designed to prevent two trains being in the tunnel at once. Elsewhere, safe operation is to be guaranteed by a time interval system, whereby consecutive trains run at least 5 minutes apart. On this occasion, the time intervals between the three trains on their departure from Brighton are 3 and 4 minutes.

There is a signal box at each end of the tunnel. The North Box is operated by **B**rown and the South by **K**illick. K has been working for 24 hours continuously. In his cabin, he has a clock, an alarm bell, a single needle telegraph and a handwheel with which to operate a signal 350 yards down the line. He also has red (stop) and white (go) flags for use in emergency. The telegraph has a dial with three indications: NEUTRAL, OCCUPIED and CLEAR.

When K sends a train into the tunnel, he sends an OCCUPIED signal to B. Before he sends another train, he sends an IS LINE CLEAR? request to B, to which B can respond CLEAR when the next train has emerged from the North end of the tunnel. The dial at one end of the telegraph only displays OCCUPIED or CLEAR when the appropriate key is being pressed at the other---it otherwise displays NEUTRAL.

The distant signal is to be interpreted by a train driver either as *all clear* or as *proceed with caution*. The signal is designed to return to *proceed with caution* as a train passes it, but if this automatic mechanism fails, it rings the alarm in K's cabin.

The accident

When train 1 passed K and entered the tunnel the automatic signal failed to work. The alarm rang in K's cabin. K first sent an OCCUPIED message to B, but then found that train 2 had passed the defective signal before he managed to reset it. K picked up the red flag and displayed it to Scott, the driver of train 2, just as his engine was entering the tunnel. He again sent an OCCUPIED signal to B.

K did not know whether train 1 was still in the tunnel. Nor did he know whether S had seen his red flag. He sent an IS LINE CLEAR? signal to B. At that moment, B saw train 1 emerge from the tunnel, and responded CLEAR. Train 3 was now proceeding with caution towards the tunnel, and K signalled all clear to the driver with his white flag.

But S had seen the red flag. He stopped in the tunnel and cautiously reversed his train to find out what was wrong from K.

Train 3 ran into the rear of Train 2 after travelling 250 yards into the tunnel, propelling Train 2 forwards for 50 yards. The chimney of the engine of Train 3 hit the roof of the tunnel 24 feet above. In all 23 passengers were killed and 176 were seriously injured.

Table 6-1. An account of the Clayton Tunnel railway accident (from [Bey98])

illustrate, and in fact also enrich and expand, DEM as well as the features of dtkeden. This section will focus on illustrating DEM by modelling the accident. The features of dtkeden are then illustrated by other examples shown in the next two sections (Section 6.2 and 6.3).

Modelling the railway accident has involved constructing computer-based artefacts to represent the perspectives of five human agents involved in the accident. These artefacts are also co-ordinated from the point of reference of the S-modeller, that is, an external observer with exceptional state-changing privileges. One significant motivation for building such a model is to gain insight into the individual understandings of the signalmen and drivers concerning their work practices at the time, and to explore how they may have contributed to the accident. The main insight gained concerns the interaction between agents. These agents could include the telegraphs, the alarm, the signal, the signalmen and the drivers. An LSD account for these agents is shown in Appendix 6-A, and Figure 6-1 depicts the distributed modelling environment of modellers. In terms of how the individual understandings of the participants contributed to the accident, this model then focuses on the animation of the accident, and the involvement and exploration of the S-modeller.

In order to capture the individual understanding of each agent, A-modellers can ask questions such as: How did the signalmen communicate with each other via the telegraphs? How big was the red flag? From what distance could the driver see the signal? On the basis of DEM, answers to such questions can be shaped through the interaction between A-modellers by means of the concept of pretend play referred to in section 4.2.

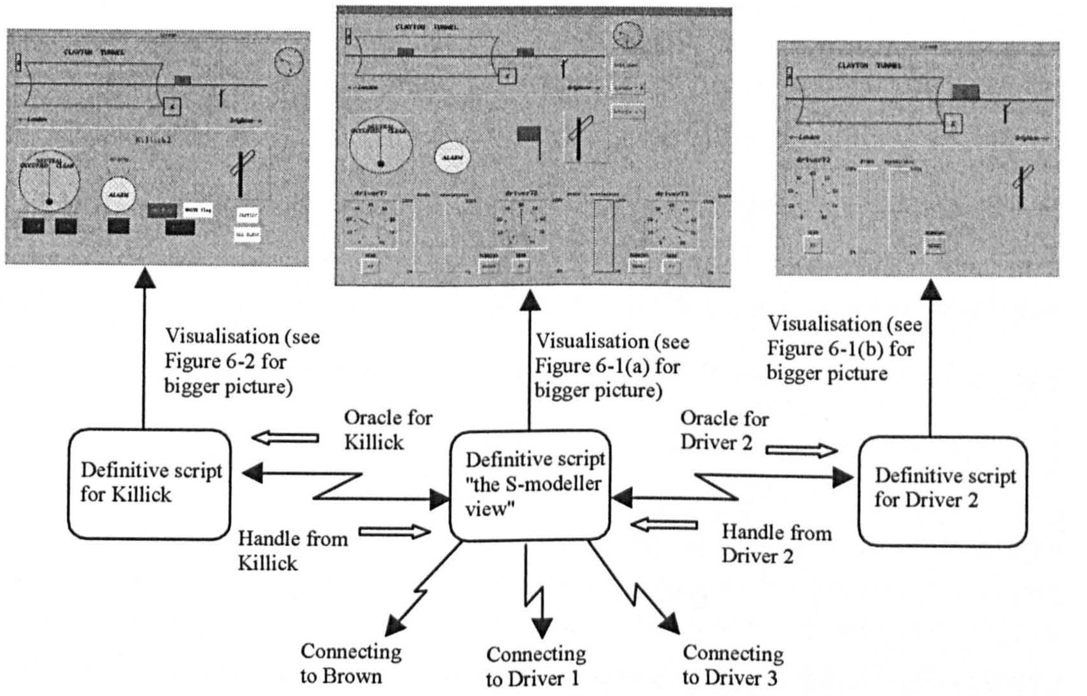

Visualisation (see Figure 6-2 for bigger picture)
Visualisation (see Figure 6-1(a) for bigger picture)
Visualisation (see Figure 6-1(b) for bigger picture
Oracle for Killick
Oracle for Driver 2
Definitive script for Killick
Definitive script "the S-modeller view"
Definitive script for Driver 2
Handle from Killick
Handle from Driver 2
Connecting to Brown
Connecting to Driver 1
Connecting to Driver 3

Figure 6-1. The modelling environment for the railway accident

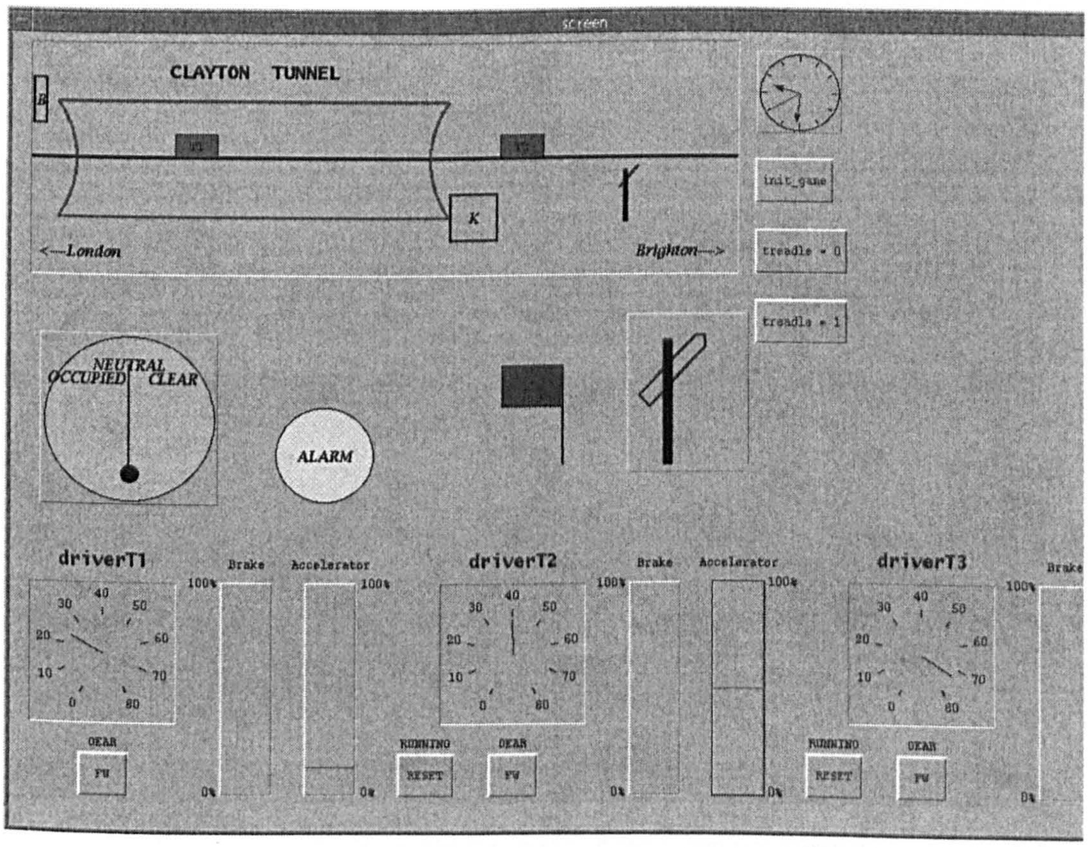


Figure 6-1 (a) A global view of the Clayton Tunnel

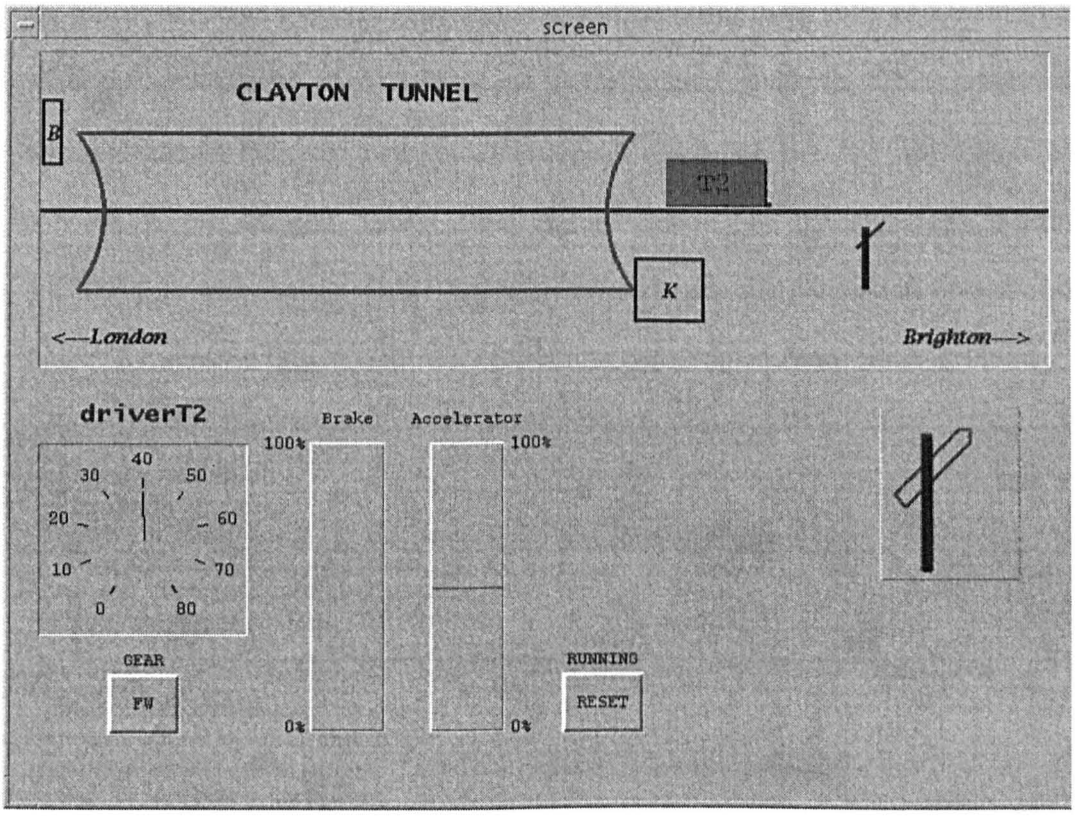


Figure 6-1 (b) The second driver's view of the Clayton Tunnel

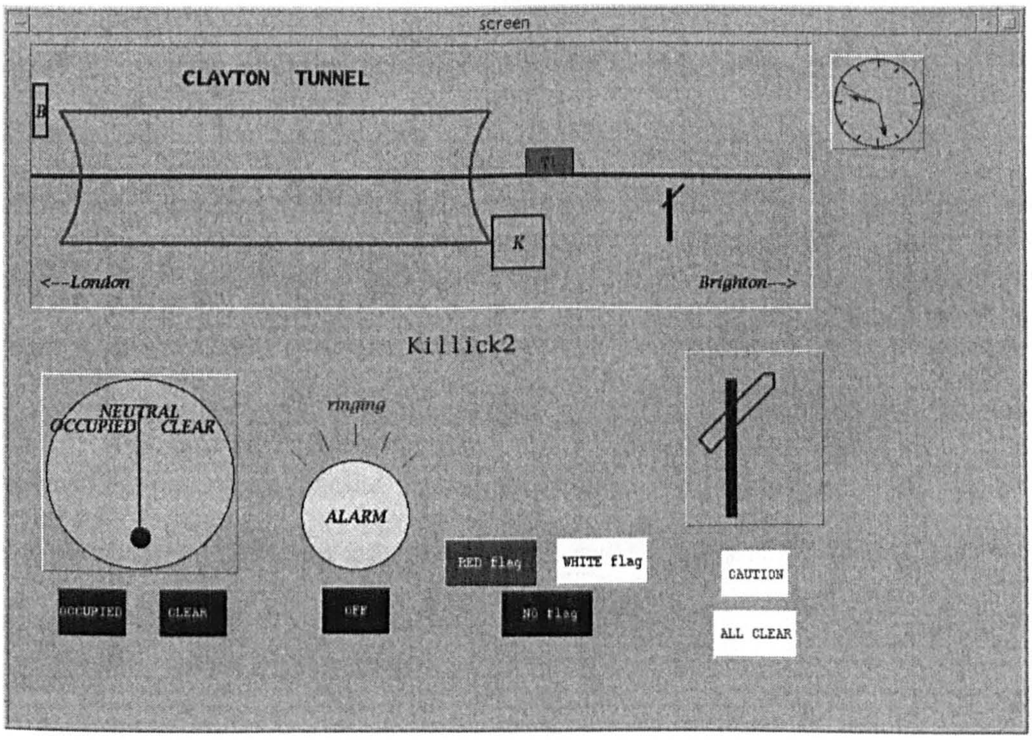


Figure 6-2. A signalman's view of the Clayton Tunnel

By way of illustration, consider how two A-modellers acting in the roles of signalmen can shape the agency involved in telegraph use. In reality, two signalmen, Killick and Brown, whose location is indicated by the boxes labelled K and B in Figure 6-2 (which represents a view of the Clayton Tunnel from the perspective of the signalman Killick), communicate with each other via the telegraph shown in the bottom-left corner of the figure. To represent this agency, two A-modellers define their private views as follows:

```
Signalman Killick:
  NeedlePos is clicked;
  clicked is (click_clear)?(1):((click_neutral)?(0):((click_occupied)?(-1)));
  clear_clicked is Killick_click_clear;
  neutral_clicked is Killick_click_neutral;
  occupied_clicked is Killick_click_occupied;
```

```
Signalman Brown:
NeedlePosition is which_clicked;
which_clicked is (clear_clicked)?(1):((neutral_clicked)?(0):((occupied_clicked)?(-1)));
click_clear is Brown_click_clear;
click_neutral is Brown_click_neutral;
click_occupied is Brown_click_occupied;
```

Definitions of this nature specify computational artefacts to represent the personal agent perspectives. Different naming conventions for observables are used to reflect the independence of the agent's observation. (It should be noted that, for simplicity, the definitions concerned with visualisation are omitted.) The order in which the various definitions are introduced is not as important as in traditional programming because dtkeden automatically maintains dependencies amongst observables. For example, in the case of Brown, the value of the observable 'needlePos' will be changed whenever the value of the observable 'clicked' is changed. Further description is needed to express the way in which the signalmen inform each other about the state of the tunnel. This could be as follows:

```
Signalman Killick:
Func send1: Killick_click_clear{
  if (click_clear)
    sendAgent("Brown", "Killick_click_clear=TRUE;");
  else
    sendAgent("Brown", "Killick_click_clear=FALSE;");
}
Func send2: Killick_click_neutral{
  if (click_neutral)
    sendAgent("Brown", "Killick_click_neutral=TRUE;");
  else
    sendAgent("Brown", "Killick_click_neutral=FALSE;");
}
Func send3: Killick_click_occupied{
  If (click_occupied)
    sendAgent("Brown", "Killick_click_occupied=TRUE;");
  else
    sendAgent("Brown", "Killick_click_occupied=FALSE;");
}
```

```
Signalman Brown:
Func sendA: Brown_click_clear{
  if (click_clear)
    sendAgent("Killick", "Brown_click_clear=TRUE;");
  else
    sendAgent("Killick", "Brown_click_clear=FALSE;");
}
Func sendB: Brown_click_neutral{
  if (click_neutral)
    sendAgent("Killick", "Brown_click_neutral=TRUE;");
  else
    sendAgent("Killick", "Brown_click_neutral=FALSE;");
}
Func sendC: Brown_click_occupied{
  if (click_occupied)
    sendAgent("Killick", "Brown_click_occupied=TRUE;");
  else
    sendAgent("Killick", "Brown_click_occupied=FALSE;");
}
```

Although the above mechanisms send definitions from one signalman to the other, this does not of itself establish useful communication. To this end, there must also be a dependency between the definitions received by a signalman acted by an A-modeller and the private definitions within its own computational artefact. When redefinitions such as the following are added to their computer-based models, the agency of changing the telegraph's states for communication is established:

```
Signalman Killick:
  clear_clicked is Killick_click_clear or Brown_click_clear;
  neutral_clicked is Killick_click_neutral or Brown_click_neutral;
  occupied_clicked is Killick_click_occupied or Brown_click_occupied;

Signalman Brown:
  click_clear is Brown_click_clear or Killick_click_clear;
  click_neutral is Brown_click_neutral or Killick_click_neutral;
  click_occupied is Brown_click_occupied or Killick_click_occupied;
```

This example illustrates how A-modellers on the basis of DEM act as agents to shape the agency of agents through the interaction with each other (that is, in the form of pretend play proposed in Section 4.2). In the early stage of modelling this railway accident, the agency that is illustrated above took the following form: a signalman presses a key on his telegraph to set both needles' positions to indicate the state of the tunnel, such as 'occupied' and 'clear', and the other signalman resets the positions. After experiments with the computer model, this agency was revised so as to take the following form[1]: a signalman holds the key down to keep the needles at a position for a while and then releases the key to return the needles to the 'neutral' position. This evolution of agency in this model highlights the significance of shaping agency through the interaction between agents acted by A-modellers. For A-modellers, the concept of pretend play can help not only to shape the agency of agents but also to improve their understanding of these agencies through enabling them to gain experience of the state change caused by their interaction from their computer-based models.

For the purpose of animating the accident, the S-modeller, who can exercise super agency in providing A-modellers with a particular context, is involved. The S-modeller, for example, can make the signal definitely fail to work by refining the observable

[1] The model has been constructed in the absence of explicit information about how signalmen at the Clayton Tunnel communicated with each other by using two telegraphs in 1861. However, the revised scenario for communication reflects our knowledge of the technology of the time (for example, the fact that electric current had to be generated by a manually operated dynamo) and allows us to interpret the interactions between the signalmen Killick and Brown, as recorded by the accident enquiry (in particular, the information that Killick sent an IS LINE CLEAR? signal to Brown and Brown replied CLEAR).

treadle_reliability equal to 0 (that is, treadle_reliability = 0)[2] in order to explore the consequent interaction between A-modellers. Exploration through 'what if' experiments provides the S-modeller with contextual resources to gain his/her insight into the interaction between agents. The S-modeller can undertake many 'what if' experiments to explore the reasons for the accident: for example, slowing down the speed of train 2, increasing the time interval between trains, enlarging the distance between Killick and the signal, and so on. It is to be expected that the S-modeller can broaden his/her insight into the accident as a result of this arbitrary exploration. This open-ended, situated modelling, that provides the S-modeller with more contextual resources, is hard to achieve by traditional modelling methods, where the allowed exploration has to be anticipated.

The potential usefulness of the computer model to an accident investigator can be illustrated by a number of 'what if' experiments. In this way, the author was able to identify scenarios to show the responsibility for the accident cannot be pinned on any one agent. For example, let us consider the case in which driver Scott sees the red flag and just stops his train in the tunnel rather than reversing it. Since Killick has seen the 'CLEAR' message from Brown, he decides to wave his white flag to inform train 3 to enter the tunnel. When train 3 enters the tunnel, the driver does not expect another train to be there (because there is no accessibility to observe train 2's position) and there is no time to stop his train before crashing into train 2. In that case, the accident occurs even though Scott follows his protocol to the letter. As a second example, consider the scenario in which Killick questions the meaning of the 'CLEAR' message and decides to stop train 3 from entering the tunnel. Since the driver of train 2 (Scott) has seen the red flag, he decides to reverse his train in order to discover what is wrong. When he is reversing the train, he is not aware that another train has stopped just in front of the entrance of the tunnel. Though Killick can see the positions of both trains, he cannot stop them, since he

[2] For reasons of convenience in animating the accident, the buttons used to set the signal as definitely working or not working are implemented in the S-modeller's model as shown in Figure 6-1(a)

has no direct control over their movements. This helps to clarify Killick's role in the accident. Analyses of this sort can show that no single agent should be blamed for the accident. It is more accurate to conclude that the accident is due to the collaborative interaction between all the agents.

Another important task performed by the S-modellers is to determine the accessibility of observables by each agent (see Section 5.2 for a more detailed explanation). For example, the following LSD account is given by the S-modeller to describe how certain observables can be accessed by the two signalman agents.

```
%lsd
agent Killick
oracle _FLAG_showing, showingFlagColour, _FLAG_flagpole_pos_x, _FLAG_goLeft
oracle _ALARM_ringing , _ALARM_flash, _TELEGRAPH_needle_pos, signalSign
oracle _CLOCK_hour, _CLOCK_min, _CLOCK_sec

handle _FLAG_showing, showingFlagColour, _FLAG_flagpole_pos_x, _FLAG_goLeft
handle _TELEGRAPH_needle_pos, signalSign, _ALARM_ringing

agent Brown
oracle _TELEGRAPH_needle_pos
handle _TELEGRAPH_needle_pos
```

Not all observables can be described in advance. For example, the observable T2*_TRAIN_train_pos_x* representing the position of train T2 occurs only if train T2 starts to move. Also, the accessibility of each agent to this observable is not persistent. In fact, signalman Killick has access to this observable when the train is in a position where it can be seen by him. In particular, Killick cannot access this observable when the train enters the tunnel. An Eden action to implement such a privilege is given as follows (only a part of the action is shown here, due to limitations of space):

```
proc changeTrainAgency : _TRAIN_train_pos_x, driverSeeFlagPos, KillickSeeTrainPos {
   auto trainPos;
   trainPos="_"//str(eval(~trainDriver))//"_TRAIN_train_pos_x";
   if (_TRAIN_train_pos_x - ~_SITE_Killick_pos <= driverSeeFlagPos &&
      _TRAIN_train_pos_x >= ~_SITE_tunnel_r_pos && !driver_see_flag) {
      addAgency("Killick", "oracle", trainPos);
      Killick_see_train = TRUE;
   }
   if ((_TRAIN_train_pos_x >= ~_SITE_Killick_pos + KillickSeeTrainPos ||
      _TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_tunnel_r_pos) &&
      Killick_see_train) {
      removeAgency("Killick", "oracle", trainPos);
      sendClient("Killick", trainPos // " = 99999;\n");
      Killick_see_train = FALSE;
   }
}
```

In this Eden action, changes to the train's position and to the distances at which objects become visible serve as triggers to redefine Killick's privilege to observe the train. The model assumes that Killick's flag will be visible to the train driver precisely when the train is visible to Killick.

The railway accident case study[3] illustrates very well not only the points of view underlying DEM but also the key features of dtkeden for supporting distributed modelling: for example, synchronous communication (among Killick, Brown and each driver), and reusable definitive patterns (see the way in which trains are specified in Appendix 6-B for example). The next two sections illustrate these features in more detail.

[3] More details of the accident animation are available on the website http://dcs.warwick.ac.uk//modelling/railway developed by S. Maad.

6.2 The Application of the Virtual Agent Concept

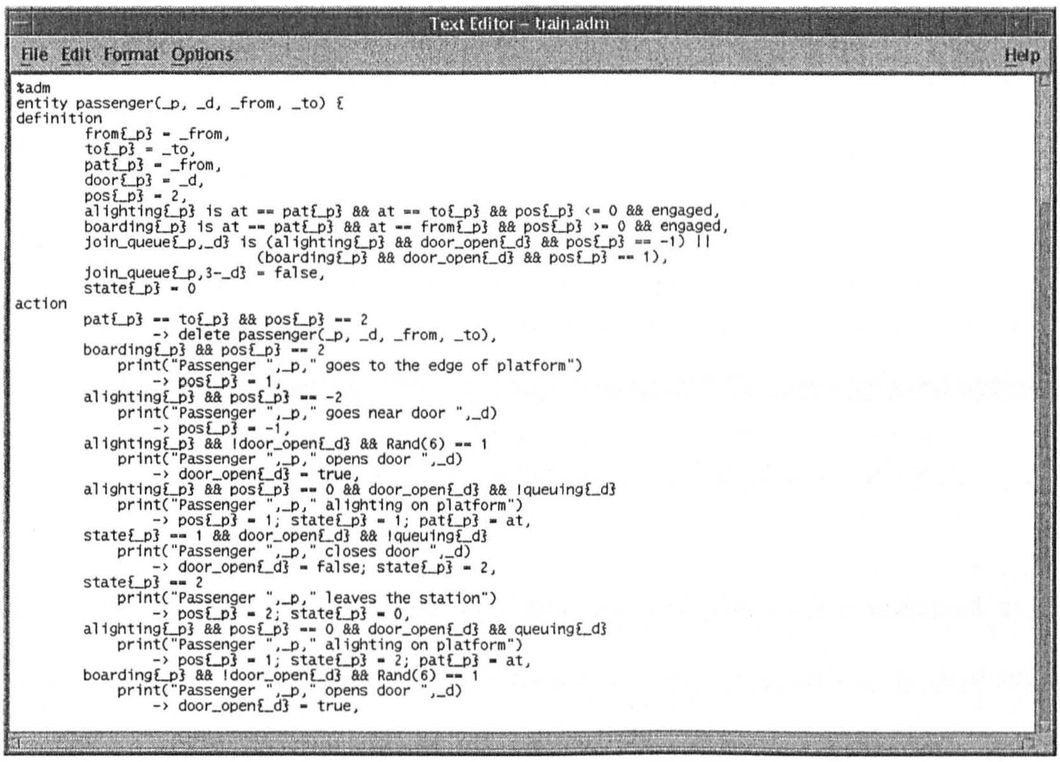
One of the most important features of dtkeden is the virtual agent concept. As described in section 5.3, the use of the virtual agent concept enables a modeller not only to localise a definitive script in accordance with a specific context but also to generate scripts for reuse. This section gives examples of using the virtual agent concept.

6.2.1 Reengineering ADM

As described earlier (2.3.1), the ADM is an abstract machine which has been developed to give operational meaning to the characters of parallel state-change and openness in an LSD account [Sla90]. An LSD account intended for describing systems at a higher-level abstraction is non-executable. In order to create a computer-based model to animate the description in an LSD account, the development of an abstract machine such as the ADM is required. With the ADM, the modeller can animate the behaviour described by an LSD account and intervene in this animation by (re)definition through definitive programming.

The task of translating an LSD account into an ADM model is so far performed manually. The account of each LSD agent is presented in the form of an entity, which includes a definition section and an action section. A general rule for this translation is that variables owned by the agent are put into the definition section and the description of *protocol* is put into the other section in the form of action (For precise details of this translation, see [Sla90]).

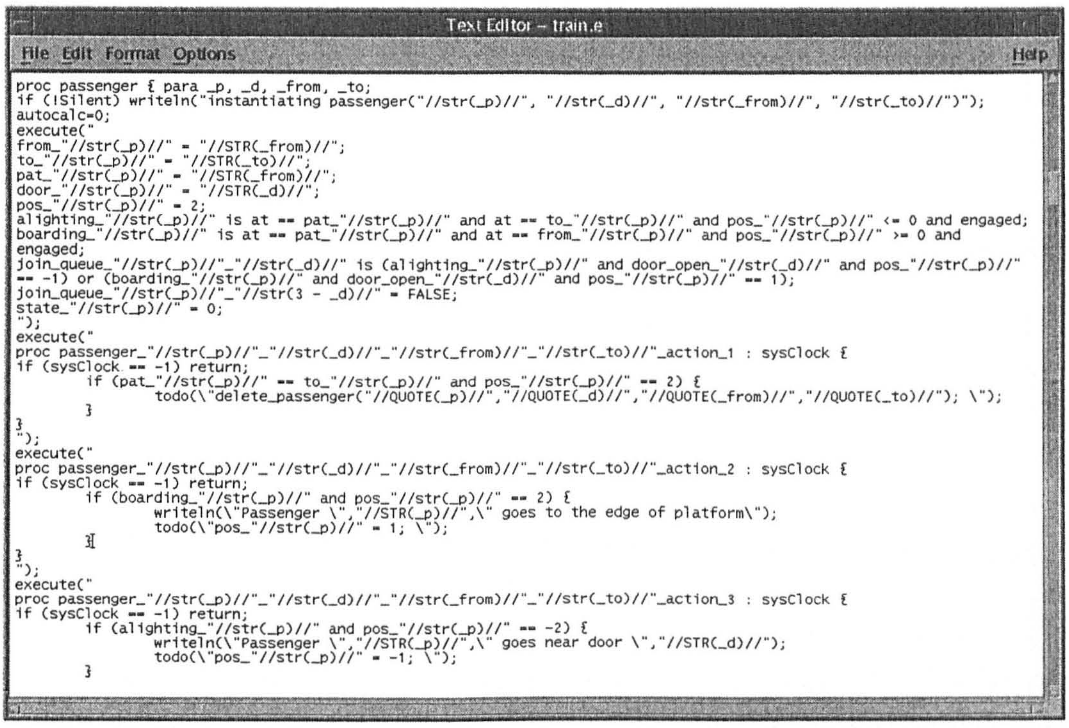
Though Slade's implementation of the ADM is a successful proof-of-concept tool to demonstrate how an animation can be created from an LSD account, its applications are limited. This is because, in this implementation, the ADM only accepts the integer data type, and has no support for visualisation. Hence, for most applications, ADM models are translated into Eden models rather than interactively interpreted. The translation can be divided into two parts: system and application. The system part embedded in each translated Eden model deals with the animation of parallel state-change in the ADM, and the application part is generated by translating entities in the ADM into Eden notation (such as definitions and actions).



```
%adm
entity passenger(_p, _d, _from, _to) {
definition
        from{_p} = _from,
        to{_p} = _to,
        pat{_p} = _from,
        door{_p} = _d,
        pos{_p} = 2,
        alighting{_p} is at == pat{_p} && at == to{_p} && pos{_p} <= 0 && engaged,
        boarding{_p} is at == pat{_p} && at == from{_p} && pos{_p} >= 0 && engaged,
        join_queue{_p,_d} is (alighting{_p} && door_open{_d} && pos{_p} == -1) ||
                             (boarding{_p} && door_open{_d} && pos{_p} == 1),
        join_queue{_p,3-_d} = false,
        state{_p} = 0
action
        pat{_p} == to{_p} && pos{_p} == 2
                -> delete passenger(_p, _d, _from, _to),
        boarding{_p} && pos{_p} == 2
            print("Passenger ",_p," goes to the edge of platform")
                -> pos{_p} = 1,
        alighting{_p} && pos{_p} == -2
            print("Passenger ",_p," goes near door ",_d)
                -> pos{_p} = -1,
        alighting{_p} && !door_open{_d} && Rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open{_d} = true,
        alighting{_p} && pos{_p} == 0 && door_open{_d} && !queuing{_d}
            print("Passenger ",_p," alighting on platform")
                -> pos{_p} = 1; state{_p} = 1; pat{_p} = at,
        state{_p} == 1 && door_open{_d} && !queuing{_d}
            print("Passenger ",_p," closes door ",_d)
                -> door_open{_d} = false; state{_p} = 2,
        state{_p} == 2
            print("Passenger ",_p," leaves the station")
                -> pos{_p} = 2; state{_p} = 0,
        alighting{_p} && pos{_p} == 0 && door_open{_d} && queuing{_d}
            print("Passenger ",_p," alighting on platform")
                -> pos{_p} = 1; state{_p} = 2; pat{_p} = at,
        boarding{_p} && !door_open{_d} && Rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open{_d} = true,
```

Figure 6-3. A snapshot of the entity *passenger* in ADM

A translator named adm has been developed to support the translation [Yun92]. With adm, in addition to the system part, each entity description is translated into a procedure (with the same name as the entity) in which the textual forms of the definition and the action sections of the entity are treated as parameters to an Eden built-in function called *execute*. Once the procedure is called, Eden definitions and actions to represent the original ADM entity are generated. This same procedure can be used when the model is running (with specified parameters, if appropriate) in order to instantiate Eden definitions and actions to reflect the (specified) context of the entity. Figures 6-3 and 6-4 illustrate the translation by adm in which an entity called *passenger* is translated into a procedure

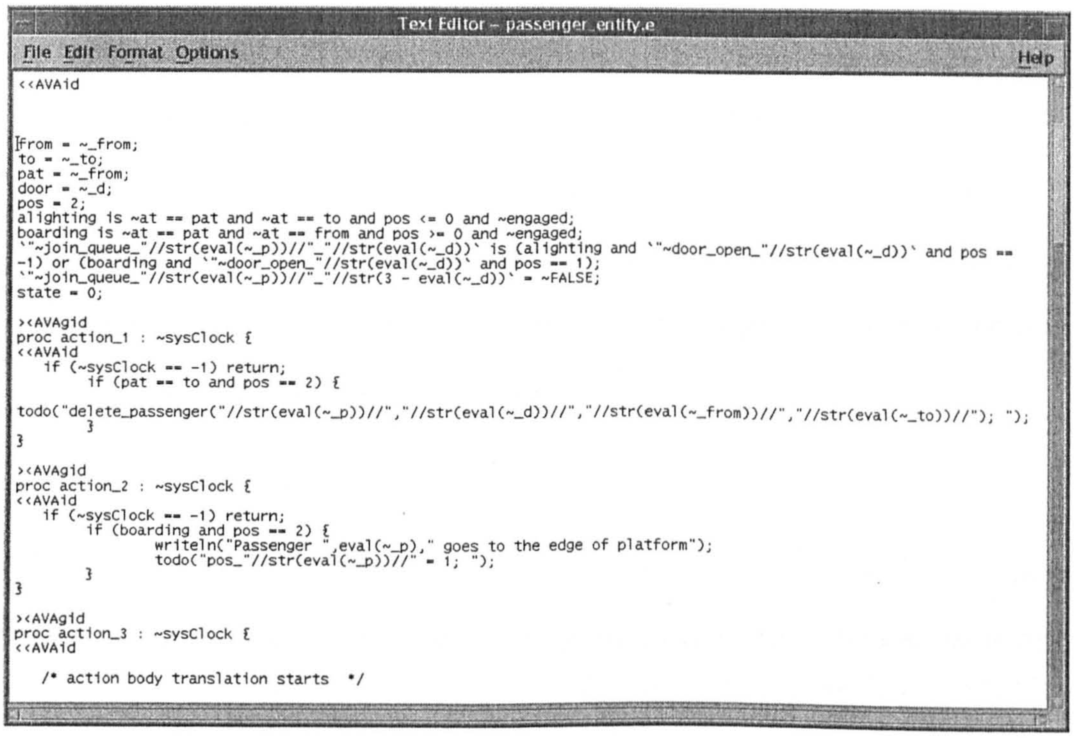


```
proc passenger { para _p, _d, _from, _to;
if (!Silent) writeln("instantiating passenger("//str(_p)//", "//str(_d)//", "//str(_from)//", "//str(_to)//")");
autocalc=0;
execute("
from_"//str(_p)//" = "//STR(_from)//";
to_"//str(_p)//" = "//STR(_to)//";
pat_"//str(_p)//" = "//STR(_from)//";
door_"//str(_p)//" = "//STR(_d)//";
pos_"//str(_p)//" = 2;
alighting_"//str(_p)//" is at == pat_"//str(_p)//" and at == to_"//str(_p)//" and pos_"//str(_p)//" <= 0 and engaged;
boarding_"//str(_p)//" is at == pat_"//str(_p)//" and at == from_"//str(_p)//" and pos_"//str(_p)//" >= 0 and
engaged;
join_queue_"//str(_p)//"_"//str(_d)//" is (alighting_"//str(_p)//" and door_open_"//str(_d)//" and pos_"//str(_p)//"
== -1) or (boarding_"//str(_p)//" and door_open_"//str(_d)//" and pos_"//str(_p)//" == 1);
join_queue_"//str(_p)//"_"//str(3 - _d)//" = FALSE;
state_"//str(_p)//" = 0;
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_1 : sysClock {
if (sysClock == -1) return;
        if (pat_"//str(_p)//" == to_"//str(_p)//" and pos_"//str(_p)//" == 2) {
                todo(\"delete_passenger("//QUOTE(_p)//","//QUOTE(_d)//","//QUOTE(_from)//","//QUOTE(_to)//"); \");
        }
}
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_2 : sysClock {
if (sysClock == -1) return;
        if (boarding_"//str(_p)//" and pos_"//str(_p)//" == 2) {
                writeln(\"Passenger \","//STR(_p)//",\" goes to the edge of platform\");
                todo(\"pos_"//str(_p)//" = 1; \");
        }
}
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_3 : sysClock {
if (sysClock == -1) return;
        if (alighting_"//str(_p)//" and pos_"//str(_p)//" == -2) {
                writeln(\"Passenger \","//STR(_p)//",\" goes near door \","//STR(_d)//");
                todo(\"pos_"//str(_p)//" = -1; \");
        }
```

Figure 6-4. A snapshot of the Eden scripts generated for the entity *passenger* by the original version of the ADM translator

(also called *passenger*). As shown in Figure 6-4, the Eden scripts generated by the translation are difficult to read. This is because additional string handling symbols for the parameters of the function *execute* are inserted. This problem can be almost entirely eliminated by re-engineering this translation.

The use of the virtual agent concept, which can repeatedly generate similar definitive scripts from a definitive pattern (explained in Section 5.3), is able to serve the purpose of instantiation in adm. This has motivated the author to develop a new translator named adm3. In adm3, each entity is translated into a definitive pattern with an unspecified virtual agent, as discussed in section 5.3. The definitive patterns stored in individual files can then be reused to generate similar scripts according to the contexts of their specified virtual agents. The function *execute* is no longer used, so that additional string handling symbols are almost unnecessary. In order to make it possible to reuse each definitive pattern describing an entity, adm3, like adm, generates a procedure with the



```
<<AVAid


from = ~_from;
to = ~_to;
pat = ~_from;
door = ~_d;
pos = 2;
alighting is ~at == pat and ~at == to and pos <= 0 and ~engaged;
boarding is ~at == pat and ~at == from and pos >= 0 and ~engaged;
`"~join_queue_"//str(eval(~_p))//"_"//str(eval(~_d))` is (alighting and `"~door_open_"//str(eval(~_d))` and pos ==
-1) or (boarding and `"~door_open_"//str(eval(~_d))` and pos == 1);
`"~join_queue_"//str(eval(~_p))//"_"//str(3 - eval(~_d))` = ~FALSE;
state = 0;

><AVAgid
proc action_1 : ~sysClock {
<<AVAid
   if (~sysClock == -1) return;
        if (pat == to and pos == 2) {

todo("delete_passenger("//str(eval(~_p))//","//str(eval(~_d))//","//str(eval(~_from))//","//str(eval(~_to))//"); ");
        }
}

><AVAgid
proc action_2 : ~sysClock {
<<AVAid
   if (~sysClock == -1) return;
        if (boarding and pos == 2) {
                writeln("Passenger ",eval(~_p)," goes to the edge of platform");
                todo("pos_"//str(eval(~_p))//" = 1; ");
        }
}

><AVAgid
proc action_3 : ~sysClock {
<<AVAid

   /* action body translation starts  */
```

Figure 6-5. The Eden scripts generated for the entity *passenger* by the author's revised version of the ADM translator

name of this entity. This procedure is much simpler than that generated by adm, since most of its content is extracted and stored into a file as a reusable definitive pattern. In effect, this procedure serves to perform the process of particularising a reusable definitive pattern as discussed in Section 5.3. It specifies the context of the virtual agent and the parameters, if any, and then includes the file of this definitive pattern in order to reuse the definitive pattern in the current context. Figure 6-5 shows the definitive pattern generated by adm3 for the entity *passenger* in Figure 6-3. Compared with the Figure 6-4, it is clear that the generated Eden model, in which there are no additional string handling symbols, is easier to read and maintain. In respect of the translation of the system part, adm3 is exactly the same as adm.

6.2.2 Other Examples

In addition to the application of reengineering the ADM translator, the use of the virtual agent concept for adaptable reuse has been illustrated in several projects. By reusing adaptable definitive patterns, these projects not only succeed in simplifying the programmer's task but also benefit from reducing model size and generating structured definitive scripts. Two examples are given here:

- The classroom project[4]

This project seeks to develop an animation system to model the interactive behaviours of pupils and the teacher in a classroom. Since the variables associated with a pupil, such as those for showing the face of a pupil, must be defined for each specific pupil in response to the observed world of the modeller, the size of the developed system (for 6 pupils) is large (about 165K bytes in total). This leads to time-consuming inconvenience and difficulty in maintaining the system: for example, the need to change some observables' descriptions and to add more pupil icons into the system on-the-fly. In fact, examining the system carefully, it is found that many chunks of scripts have a high degree of similarity: for example, the description of each pupil's behaviour, the Scout windows for showing pupil icons and measuring the personal characteristics of pupils, and the description of drawing each pupil's face in DoNaLD. These similarities point to the use of the virtual agent concept for reducing size and maintenance load. Corresponding to the modeller's practical experience, GOs (that is, generic observables) and their definitive patterns, for example, *girlface*, *boyface*, *pupil-icon*, can be generalised as reusable patterns. These patterns can then be particularised to create the needed definitive scripts on-the-fly. The generated scripts are adaptable in accordance with their

[4] The project was originally developed by a third-year student in the author's department. To date, its modification by using the virtual agent concept has been developed collaboratively by the present author and another PhD student S. Rasmequan.

specific contexts. For example, the description of the position of each pupil's icon can be modified in response to its real position on the displayed screen.

By exploiting the virtual agent concept, the size of the system has been reduced by 40% (to about 100K bytes). More significantly, these definitive patterns become reusable components and can be conveniently reused with optimal modification on-the-fly. Neither white-box nor black-box reuse provides this feature of adaptable reuse when the model is running, since both kinds of reusable component are well-defined in advance and cannot be modified on-the-fly. Figure 6-6 shows that the system has been extended to 12 pupils just by reusing definitive patterns without additional description.

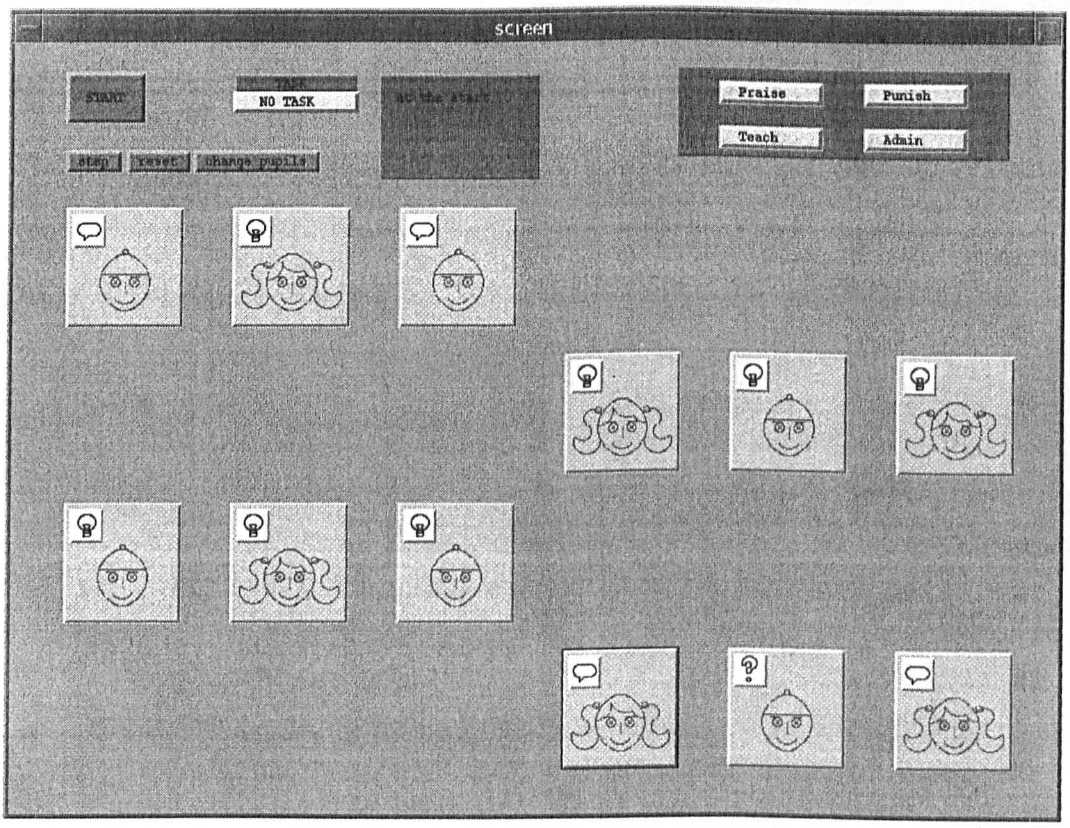
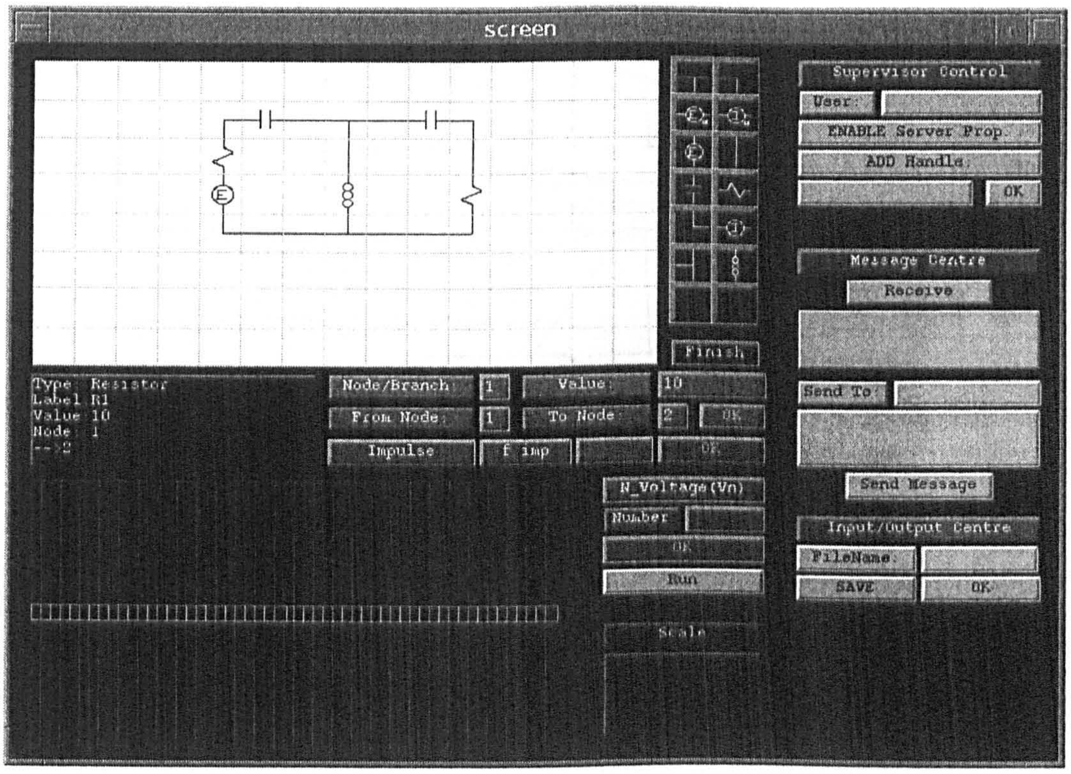


Figure 6-6. The application of reusable definitive patterns in the classroom simulation system

- The virtual electrical laboratory project[5]

This project uses EM to develop a distributed electrical laboratory system for educational purposes. Within the dtkeden environment, the scenarios of the system can be considered as follows: the teacher at the S-node draws up a circuit diagram and sends it to students at A-nodes, and each student interacts with the received diagram by changing its components and their values for learning purposes.



```
Type: Resistor
Label R1
Value 10
Node 1
-->2
```

Figure 6-7. A snapshot of virtual electronic laboratory

In order to support the frequent interaction between the teacher and students, and their computer models, a large number of icons are used. For example, a circuit diagram can be drawn by selecting a symbol from an icon bar and pasting the selected symbol into an icon of the workspace consisting of another 120 icons with empty contents. In fact, as shown in Figure 6-7, more than 200 icons are used for the interface to support the

[5] This MSc research project was jointly developed by H. P. D'ornellas [Dor98] and C. R. Sheth [She98].

interaction and display graphic data. Each icon is specified by defining a Scout window to include a DoNaLD picture. As can be imagined, a heavy load of modelling is inevitable for creating and maintaining these icons in tkeden; however, the load can be relieved by using the virtual agent concept. For example, a GO named 'agent.p4' provides a definitive pattern which can be reused for describing icons in the icon bar (more details are given in [She98]).

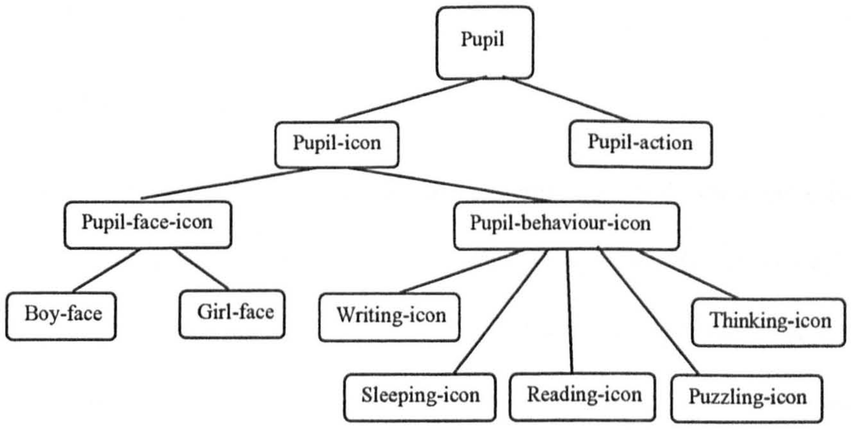


Figure 6-8. The partial hierarchical structure of the modified classroom simulation system

Both the above projects demonstrate the advantages of applying the virtual agent concept to software reuse, in particular for adaptable reuse when the model is running, as discussed in Section 5.3. More significantly, it is found that the creation of definitive patterns facilitates the construction of the hierarchical structure of the developed system. Where definitive programming is used, as in tkeden, it is typically not necessary for the modeller to address issues associated with the structure of models. However, a structure for dtkeden models can be explicitly introduced when reusable definitive patterns in dtkeden are subtly devised. For example, Figure 6-8 illustrates the partial structure of the classroom simulation system (the modified version). It should be noted that such an application of definitive patterns need not be regarded as imposing a fixed structure on the system being developed, due to the adaptability of these patterns.

6.3 Examples of Interaction Modes

There are four interaction modes in dtkeden. The examples in the previous sections are typically in the normal mode. This section includes another two examples developed by the author to illustrate different modes.

- A two-player OXO game

This system was originally developed to model a generic OXO game in the tkeden environment, that is, in a stand-alone environment (details can be found in [Nes97]). The only user of the system is the modeller, who, being the superagent, is empowered to access and change all variables of the computer-based model. When the system is extended to a distributed environment, there is no longer a single super agent.

In the extended OXO model, more modellers are involved: two for players X and O at A-nodes and one for the umpire (associated with 'the S-modeller view') at the S-node. Through a communication network, a player X can send a definitive script, for example a definition s1 = x (describing a cross is placed to position s1), to interact with (e.g. to play OXO with) another player O[6]. Due to the star-type logical network configuration described earlier (Section 5.1), the script is first automatically directed to the S-node. If the interaction mode of the S-node is broadcast mode, the script will be accepted so as to affect the computer model at the S-node, and will concurrently be sent to another player O, leading to the change of the visualisation of player O's computer

[6] Although a player can also input a definitive script without sending it to the umpire, in order to interact with his/her own computer model, the interaction will be regarded as a stand-alone modelling in EM for individual cognition. This is not considered in this section.

model. Figure 6-9 shows the interaction between the three models in this form of broadcasting.

If the interaction mode is declared as interference mode, scripts that are sent to the S-node will be displayed in the umpire's input window pending further action from the umpire (as shown in Figure 6-10). At this point, the umpire can interfere in the interaction between two players, for example by changing the definition s1 = x (received from player X) to s1 = o (signifying the placing of a nought in position s1) and sending it to the player O. As a result, a surprising inconsistency occurs in the players' computer models. The unexpected contexts that arise from interference at the S-node can enrich the understanding of all the modellers involved in the system.

- A monitoring system for an educational game

The jugs model in tkeden implements a simple educational game intended to teach pupils elementary number theory (see [Bey89+, BS98] for details). A dtkeden system has been developed by extending the stand-alone version of the jugs model in tkeden. This system allows a teacher (sited at the S-node) to monitor the progress of several pupils who are independently playing jugs. The interaction mode at the S-node for this system is set to the private mode so as to establish private channels to individual pupils for monitoring their playing context. In this way, each interaction performed by a particular pupil who is playing jugs on his/her computer model is propagated to the model at the S-node and only affects the part of the teacher's model which corresponds to that pupil's context. Figure 6-11 illustrates how different contexts, corresponding to the models of different pupils, can be shown in the same model at the S-node, though these models essentially have the same observables and dependency for each pupil. This system demonstrates the archetypal usage of the virtual agent concept, viz. to localise definitive scripts in accordance with their contexts. This concept of localisation gives rise to the concept of adaptable reuse.

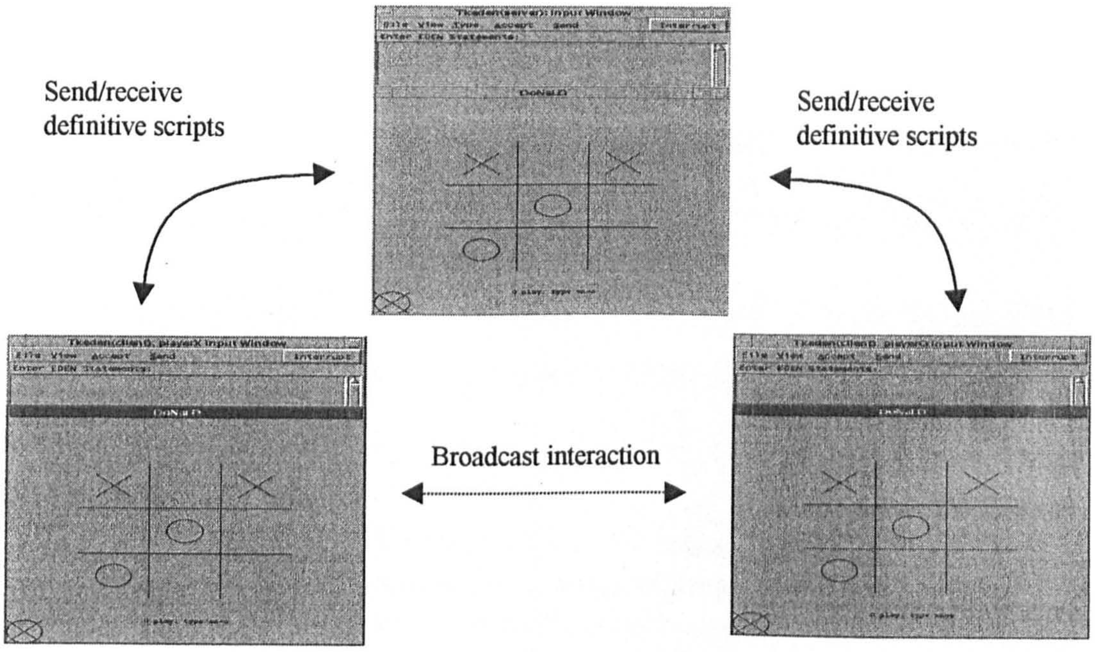

Send/receive definitive scripts
Send/receive definitive scripts
Broadcast interaction
DoNaLD
DoNaLD
DoNaLD

Figure 6-9. Interaction in the broadcast mode

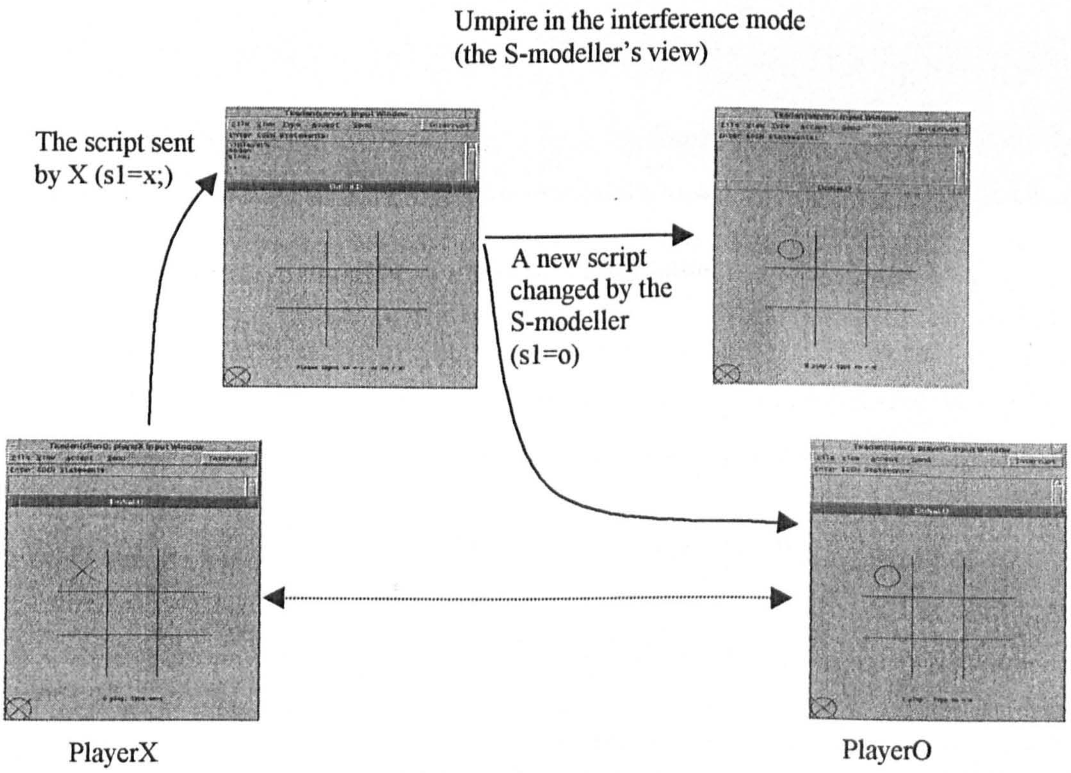
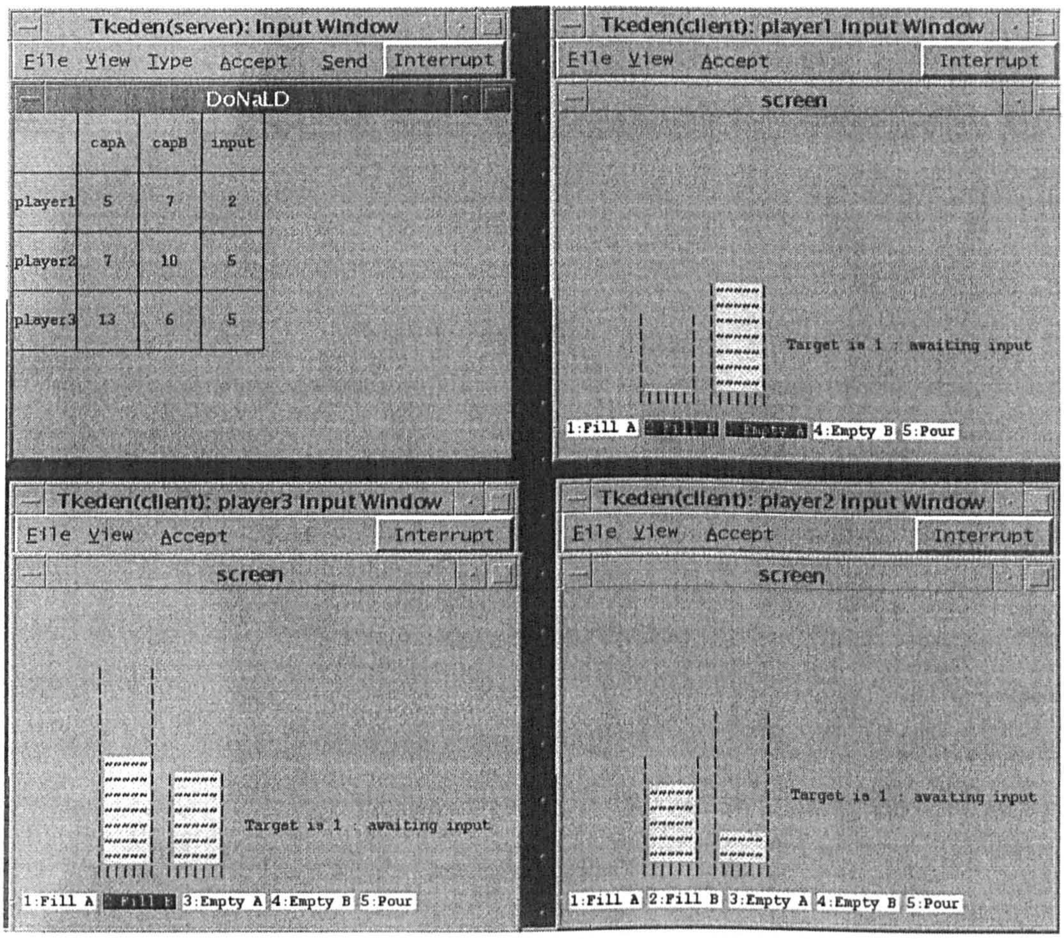

Umpire in the interference mode
(the S-modeller's view)
The script sent by X (s1=x;)
A new script changed by the S-modeller (s1=o)
PlayerX
PlayerO

Figure 6-10. Interaction in the interference mode


Tkeden(server): Input Window
File View Type Accept Send Interrupt
DoNaLD
Tkeden(client): player1 Input Window
File View Accept Interrupt
screen
Tkeden(client): player3 Input Window
File View Accept Interrupt
screen
Tkeden(client): player2 Input Window
File View Accept Interrupt
screen
1:Fill A
2:Fill B
3:Empty A
4:Empty B
5:Pour

	capA	capB	input
player1	5	7	2
player2	7	10	5
player3	13	6	5

```
Target is 1 : awaiting input
```

```
Target is 1 : awaiting input
```

```
Target is 1 : awaiting input
```

Figure 6-11. Different contexts of a jugs game in the private mode

Appendix 6-A: An LSD Account for the Railway Accident

This appendix presents an LSD account of the agents involved in the Clayton Tunnel railway accident. The account is subject to change as more knowledge emerges from the modelling. In addition, for the sake of convenience, each of the key positions of a train in this account is represented by a simple letter. These symbols and their meaning are explained after the LSD account.

```
agent Killick {
state
    set_alarm_off
    set_signal
    showing_flag
 oracle
    clock_time
    train_position(i)       /* get this agency when he can see the train i, but lost this agency when
                               the train i enters the tunnel. */
    telegraph_needle_position
    alarm_ringing
 handle
    set_needle_position     /* set telegraph_needle_position to OCCUPIED(1) */
    set_signal          /* set signal_sign to ALL_CLEAR(0) or CAUTION(1) */
    set_alarm        /* we assume alarm will keep ringing until Killick reset it */
    showingFlagColour
 protocol
    /* when telegraph needle position is in CLEAR, set it to NEUTRAL
       and reset signal to ALL_CLEAR */
   telegraph_needle_position == -1  -> set_needle_position = 0, set_signal = 0
    /* when train is entering the tunnel, Killick wants to set telegraph
```

```
        needle's position to OCCUPIED and no showing flag */
      train_position(i) >= e    -> set_needle_position = 1, showing_flag == FALSE
      /* when the alarm is ringing, Killick wants to set it off and show a flag to indicate situation */
      alarm_ringing == TRUE  -> set_alarm = OFF, showing_flag = TRUE
      /* when Killick wants to showing an indicating flag and a train is
        in the tunnel, he shows the red flag. */
      showing_flag == TRUE && telegraph_needle_position == 1 -> showingFlagColour="RED"
     /* when Killick wants to showing an indicating flag and no train is
        in the tunnel, he shows the white flag. */
      showing_flag = TRUE && telegraph_needle_position == 0  -> showingFlagColour="WHITE"
}
agent Brown {
oracle
      train_position(i) /* will get this agency when the train i is leaving the tunnel. */
      telegraph_needle_position
  handle
      set_needle_position /* set telegraph_needle_position to CLEAR(0) */
  protocol
      trans_position(i) + train_length(i) >=f    -> set_needle_position = -1
}
agent driver(i) {
  state
      pedalAccelerator /* TRUE or FALSE */
      pedalBrake       /* TRUE or FALSE */
      set_train_speed  /* UP, KEEP, DOWN and STOP */
      incrAccPos   /* pedal accelerator further(1), still(0), or less(-1) */
      set_gear
  oracle
      signal_sign  /* the driver gets this agency when he can see the signal */
  showing_flag /* the driver gets this agency when he can see the flag showing by Killick */
handle
  treadlePressed /* to see if the train has press the treadle  */
  signal_treadle  /* set signal_sign to CAUTION(1) or FAIL(-1) when the train passes it */
  brakePos      /* 0 ~ 1 */
  accPos        /* 0 ~ 1 */
privilege
  train_position > b  -> train_press_treadle = @
  train_position <= b  -> train_press_treadle = TRUE
  train_position + train_length <= b  -> train_press_treadle = FALSE
  /* when the train i passes over b point and havn't pressed the treadle,
    press the treadle and make it set the signal or alarm */
  train_press_treadle == TRUE && treadlePressed == FALSE
    -> treadlePressed = TRUE, signal_treadle = ON
  train_press_treadle == FALSE && treadlePressed == TRUE  -> treadlePressed = FALSE
  /* When the driver see ALL_CLEAR sign indicated by signal or a flag,
    he speeds up or keeps the train speed */
  signal_sign == 0 || showing_flag == "WHITE"
    -> set_train_speed = UP or set_train_speed = KEEP
  /* When the driver see CAUTION sign indicated by signal, he slowes
    down the train's speed by reducing his accelerator*/
  signal_sign = 1   -> set_train_speed = DOWN
  /* When the driver see OCCUPIED sign indicated by the red flag, he
    stop the train behind the point d by using his braker*/
  showing_flag = RED -> set_train_speed = STOP
  set_train_speed = UP  -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = 1
  set_train_speed = KEEP  -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = 0
  set_train_speed = DOWN -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = -1
  set_train_speed = STOP    -> pedalBrake = TRUE, pedalAccelerator = FALSE
```

```
}
agent signal {
  state
    signalSign        /* CAUTION(1) and ALL_CLEAR(0) */
    currentSignalSign  /* current signal sign        */
    signal_treadle    /* cause the signal to be set to CAUTION(1) */
    treadlePressed
  handle
    signalSign
    set_alarm
  protocol
    /* When the treadle is set to ON by a train and the signal doesn't fail,
      reset the treadle to OFF and set signal to CAUTION. */
    signal_treadle == ON && signalFailure = FALSE  -> signal_treadle = OFF, set_signal = 1
    /* When the treadle is set to ON by a train and the signal does fail,
      set the alarm to ringing and reset the treadle to OFF. */
    signal_treadle == ON && signalFailure = TRUE
      -> set_alarm = ON, signal_treadle = OFF, set_signal = -1
    set_signal == 1  -> signalSign = 1, currentSignalSign = 1
    set_signal == 0  -> signalSign = 0, currentSignalSign = 0
    set_signal == -1  -> signalSign = currentSignalSign
    /* When the treadle is set to ON by a train, it wants to set signal
      CAUTION by a given way (here I assumed it is random) */
    signal_treadle == ON  -> signalState = rand(100), signalFailure = @
    signalState <= signalReliability * 100  -> signalFailure = FALSE
    signalState > signalReliability * 100    -> signalFailure == TRUE
}
agent telegraph {
  state
```

```
    telegraph_needle_position /* OCCUPIED(-1) NEUTRAL(0) CLEAR(1) */
  protocol
    set_needle_position = -1 -> telegraph_needle_position = -1
    set_needle_position = 0  -> telegraph_needle_position = 0
    set_needle_position = 1  -> telegraph_needle_position = 1
}
agent alarm {
  state
    alarm_ringing
  protocol
    set_alarm = ON -> alarm_ringing = TRUE
    set_alarm = OFF -> alarm_ringing = FALSE
}
agent train (i) {
  state
    train_position(i)
    train_speed(i)
    train_length(i)
    gearFw
  derivate
    movedDistance = train_speed(i) * timePeriod + 0.5 * Acc * timePeriod ^2
    train_position(i) = |train_position(i)| + movedDistance * movedDirection
  protocol
    gearFw = TRUE  -> movedDirection = 1
    gearFw = FALSE -> movedDirection = -1
}
```

The following shows various positions of a train passing through the Clayton Tunnel (from the right end to the left end).

← London

```
X<-----X<---------X<-----X<---------X<-----------------X<-------X<-
g      f          e      d          c                  b        a
```

point a: start point - it is assumed that the driver can see the signal from this point.
point b: the position of the signal
point c: the position where Killick and the drivers can see each other, this is changeable
point d: the position of Killick
point e: the entrance of the tunnel
point f: the exit of the tunnel where Brown sees the train emerging
point g: the end point where the train disappears from Brown's view.

Appendix 6-B: An Example of a Generic Observable (GO) – *train*

```
%donald
><trainDriver
viewport ~site
openshape TRAIN
within TRAIN {
    int trainLength, trainHigh
    trainHigh = ~_SITE_size! div 10
    point train_pos
    int train_pos_x, train_pos_y
    train_pos_x = (~_SITE_size! div 50) + (~_SITE_railLen!)
    train_pos_y = ~_SITE_rail_pos_y!
    train_pos = {train_pos_x, train_pos_y}
    rectangle trainBody
    trainBody = rectangle(train_pos, train_pos + {trainLength, trainHigh})
    label trainLabel
    char trainNo
    trainLabel = label(trainNo, train_pos + {trainLength div 2, trainHigh div 2})
    ? `"A_" // str(eval(~trainDriver)) // "_TRAIN_trainBody"` = "fill=solid,color=purple";
}
%eden
_TRAIN_trainLength=100 * ~len_ratio;
_TRAIN_trainNo = str(~train_id);
trainStartPos = (float(~_SITE_size) / float(50)) + (~_SITE_railLen);
Killick_see_train = FALSE;
Brown_see_train = FALSE;
driver_see_signal = FALSE;
driver_see_flag = FALSE;
defineCrash = FALSE;
driverSeeSignalPos is eval(~driver_see_signal_pos) * ~len_ratio;
KillickSeeTrainPos is eval(~Killick_see_train_pos) * ~len_ratio;
driverSeeFlagPos is eval(~driver_see_flag_pos) * ~len_ratio;

proc changeTrainAgency : _TRAIN_train_pos_x {
   auto trainPos, temp, temp1, temp2, i;
   trainPos="_"//str(eval(~trainDriver))//"_TRAIN_train_pos_x";
   if (_TRAIN_train_pos_x <= ~_SITE_Killick_pos + KillickSeeTrainPos &&
      _TRAIN_train_pos_x + _TRAIN_trainLength >= ~_SITE_tunnel_r_pos &&
   !Killick_see_train) {
   addAgency("Killick", "oracle", trainPos);
   Killick_see_train = TRUE;
 }
 if ((_TRAIN_train_pos_x >= ~_SITE_Killick_pos + KillickSeeTrainPos ||
    _TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_tunnel_r_pos) &&
   Killick_see_train) {
   removeAgency("Killick", "oracle", trainPos);
  sendClient("Killick", trainPos // " = 99999;\n");
   Killick_see_train = FALSE;
 }
 if (_TRAIN_train_pos_x <= ~_SITE_tunnel_l_pos && !Brown_see_train) {
  addAgency("Brown", "oracle", trainPos);
  Brown_see_train = TRUE;
  sendClient("Brown","autoClearButton();\n");
 }
 if (_TRAIN_train_pos_x + _TRAIN_trainLength < 0) {
  removeAgency("Brown", "oracle", trainPos);
  Brown_see_train = FALSE;
 }
 if (_TRAIN_train_pos_x - ~_SITE_signal_pos <= driverSeeSignalPos &&
   _TRAIN_train_pos_x >= ~_SITE_signal_pos && !driver_see_signal) {
  addAgency(eval(~trainDriver), "oracle", "signalSign");
  addAgency(eval(~trainDriver), "handle", "signalSign");
  addAgency(eval(~trainDriver), "handle", "_ALARM_ringing");
  sendClient(eval(~trainDriver), "signalSign = "//str(~signalSign)//";\n");
  driver_see_signal = TRUE;
 }
 if (_TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_signal_pos &&
   driver_see_signal) {
  removeAgency(eval(~trainDriver), "oracle", "signalSign");
  removeAgency(eval(~trainDriver), "handle", "signalSign");
  removeAgency(eval(~trainDriver), "handle", "_ALARM_ringing");
  driver_see_signal = FALSE;
 }
 if (_TRAIN_train_pos_x - ~_SITE_Killick_pos <= driverSeeFlagPos &&
   _TRAIN_train_pos_x >= ~_SITE_tunnel_r_pos && !driver_see_flag) {
  addAgency(eval(~trainDriver), "oracle", "_FLAG_showing");
  addAgency(eval(~trainDriver), "oracle", "showingFlagColour");
```

```
    addAgency(eval(~trainDriver), "oracle", "_FLAG_flagpole_pos_x");
    addAgency(eval(~trainDriver), "oracle", "_FLAG_goLeft");
    sendClient(eval(~trainDriver), "_FLAG_showing = "//str(~_FLAG_showing)//";\n");
    sendClient(eval(~trainDriver), "showingFlagColour = \""//str(~showingFlagColour)//"\";\n");
    sendClient(eval(~trainDriver), "_FLAG_flagpole_pos_x =
"//str(~_FLAG_flappole_pos_x)//";\n");
    sendClient(eval(~trainDriver), "_FLAG_goLeft = "//str(~_FLAG_goLeft)//";\n");
    driver_see_flag = TRUE;
  }
  if (_TRAIN_train_pos_x <= ~_SITE_tunnel_r_pos && driver_see_flag) {
    removeAgency(eval(~trainDriver), "oracle", "_FLAG_showing");
    removeAgency(eval(~trainDriver), "oracle", "showingFlagColour");
    removeAgency(eval(~trainDriver), "oracle", "_FLAG_flagpole_pos_x");
    removeAgency(eval(~trainDriver), "oracle", "_FLAG_goLeft");
    driver_see_flag = FALSE;
  }
  if (!defineCrash && ~trainList# > 1) {
    for (i = 1; i <=~trainList#; i++) {
      if (str(~trainList[i]) != eval(~trainDriver))
        ~defineTrainCrash(str(~trainList[i]), eval(~trainDriver));
    }
    defineCrash = TRUE;
  }
}

>>
message = "train_id = \"" // str(train_id) // "\";\n";
message = message // "trainDriver = \"" // str(trainDriver) // "\";\n";
message = message // "include(\"/dcs/res/sun/dtkeden/railway/train.panel\");\n";
sendClient("Killick", message);
sendClient("Brown", message);
append trainList, trainDriver;
```

Chapter 7
Distributed Empirical Modelling for Requirements Engineering

It is increasingly recognised that one of the most intricate and important tasks in software system development (SSD) is to understand the needs of users. No matter how well designed they are, software systems that fail to satisfy the users' needs will disappoint users and bring grief to developers. Accordingly, Requirements Engineering (RE), which aims to solve this problem by developing requirements in a systematic fashion, has recently attracted widespread attention in the software community. Unfortunately, since RE is located at the intersection of formal and informal, objective and subjective, and technical and non-technical approaches, the RE process (REP) is very difficult to understand fully [Fin94, Bub95]. The development of requirements remains the most crucial, labour-intensive and expensive part of SSD.

7.0 Overview

This chapter seeks to apply the framework of DEM to effective requirements development. In Section 7.1, a brief overview of RE and the difficulties of enacting the REP are given. These difficulties, arising from the inability to provide adequate support for the interpersonal interaction and take sufficient account of context (i.e. the real-world environment in which requirements are developed and the system is used), have given rise to the wide gap between research and practice [BL98].

Section 7.2 attempts to bridge this gap by reengineering the REP. Firstly, to reflect the situatedness of requirements [Gog94], a new definition of requirements that draws more attention to the context in which the requirements are developed and used is given. To correspond to this definition, the need for requirements to be incrementally formulated through interpersonal interaction is identified. This section then reengineers the REP by regarding it as a problem-solving process in which human agents interact with each other in order to develop requirements within their context. In this way, requirements are cultivated in order to solve problems as they are identified in the real world. Hence, the REP must closely intertwine with SSD in a symbiotic manner rather than simply being bolted on as a front-end to SSD. Section 7.2 also discusses the influence of the interactive relationship between human agents on the interaction of the REP, and consequently proposes that a computer-based, collaborative interaction environment is useful for cultivating requirements.

In response to the basic characteristics of the REP, Section 7.3 proposes a human-centred framework, called SPORE, whereby requirements as 'solutions to identified problems' in the application domain are developed in an open-ended and situated manner. Within this framework, people participating in the REP are able to cultivate requirements through collaborative interaction with each other in order to solve the identified problems, rather than by searching for requirements in the 'jungle' of users' needs. The principles and concepts of DEM are applied to SPORE to support the collaborative interaction between participants and the situatedness of the enacted the REP. By enacting DEM using dtkeden, computer-based models can be created and used by participants to serve two functions:

- as artefacts to explore, expand and experience the solutions to the identified problems.

- as a powerful communication medium to support their collaborative interaction in order to 'grow' the solutions through incremental development in a distributed environment.

Two examples of the use of SPORE for requirements development are given in section 7.4. The first example is that of developing requirements for an interactive software system embedded in an automatic teller machine (ATM). A comparison between SPORE and a viewpoint-oriented model, called VORD [KS98], of requirements development is discussed. Another example is that of a warehouse inventory information system. A use-case driven approach to requirements development for this application is found in [JCJO92], and this is compared with SPORE.

7.1 Requirements Engineering

In his celebrated paper [Bro87], F. Brooks argued that "[t]he hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements ... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later". Indeed, a poor understanding of the needs of users has become the main cause of system failures. The detection and correction of errors arising from such poor understanding is often the most difficult and expensive task in SSD [STM95]. The worst thing is that such errors may remain undetected until system operation, thereby provoking failures that have serious consequences, especially in safety critical systems. Therefore, capturing the needs of users accurately is a vital component of successful software system.

7.1.1 An Overview of Requirements Engineering

Requirements Engineering (RE) typically refers to that part of the SSD) life-cycle in which application engineers investigate the needs of the user community and abstract from these needs to form descriptive specifications for the development of software systems. It involves intellectually challenging and creative activities, acknowledged to be the most costly and error-prone parts of SSD. A systematic process is needed for RE to derive the users' needs for the software system that is to be developed. In order to facilitate requirements acquisition, this process is conventionally divided into a set of well-defined activities characteristic of an engineering discipline. The term 'requirements' is defined in IEEE-Std.'610' as follows [IEEE90]:

1. A condition or capacity needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or processed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in 1 or 2.

In this thesis, people who are engaged with these activities for understanding and acquiring requirements are referred to as *participants*. Typically, they come from two camps: a usage camp, including end-users of the software system, managers and others affected by the system; and a development camp, including analysts and designers responsible for the system development, maintainers in charge of maintaining the developed system, and requirements engineers enacting RE. For reasons of convenience, the general terms 'users' and 'developers' will be used here to describe people from the two camps.

So far, there has been no agreement on a standard definition of RE. For example, P. Loucopoulos and V. Karakostas define RE as "the systematic process of developing requirements through an iterative co-operative process of analysing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained" [LK95]. This definition is concerned mainly with the *technical* issues of enacting RE for requirements acquisition. Non-technical issues, such as social contexts and cognitive concerns, are only peripherally implicated in this account through the references to co-operative interaction between participants and to understanding gained. By contrast, according to J. Bubenko, RE "can be said to be the area of knowledge concerned with communicating with organisational actors with respect to their visions, intentions, and activities regarding their need for computer support, and developing and maintaining an adequate requirements specification of an information system" [Bub95]. This definition suggests that RE should embrace not merely technical but also managerial, organisational, economic, social issues and problems.

Although definitions of RE vary, it is commonly agreed that RE plays a very critical role in the development of an appropriate software system with the required degree of quality assurance [Rol94, LK95]. RE is conventionally viewed as a phase in the early stage of the life cycle of SSD [LK95, KS98, SS97]. It is assumed that the gap between users and developers can be narrowed by obtaining well-structured, well-described specifications of the needs of users. Given the specifications, developers can confidently continue the remaining phases of the SSD life cycle and finally deliver the system, fully satisfying the users' needs. This linear dependency emphasises the importance of RE in paving the way for successful SSD.

In spite of its importance, the process by which requirements are apprehended and acquired is poorly understood [Fin94, Bub95]. In order to overcome this deficiency, a model of REP is needed. Many process models serving this purpose have been proposed. According to M. Dowson's classification, each of them falls into one or more of the following categories: *product-oriented*, *activity-oriented* and *decision-oriented* [Dow87]. These may be summarised as follows.

- Product-oriented

This kind of model focuses on the product of RE. It aims to help developers to construct correct descriptive documents [DBP93a, DBP93b, FHW94, KS98, LL94, LH94, Lou94]. Most such models decompose a root definition from the highest level into a number of less abstract modules in order to understand the structure and functionality of the whole software system which it is proposed to develop. The common assumption is that requirements pre-exist and are hidden, and can then be retrieved from their sources before being fixed in the form of a descriptive representation. When enacted by this kind of model, the REP can be regarded as a process of transforming informal, fuzzy individual statements of users' needs to a formal precise description of requirements that is understood by all participants. The final result of this transition takes the form of requirements specifications recording the users' needs in a well-defined and well-structured descriptive format.

Requirements specifications are typically a kind of paper-based documentation in the form of text or diagrams. Rigid formality of requirements specification is usually required in order to conform to certain characteristics such as completeness, correctness, unambiguity, understandability, modifiability and consistency, though many of these qualities are very difficult to achieve and test. In addition, from the product-oriented perspective, it is maintained that the description of requirements should not involve any design details, due to the conventional wisdom in RE that requirements are concerned with only *what* is desired without referring to *how* it is to be implemented [Dav93, HJ89, KS98, SS97].

Once the specifications are accomplished, they must be signed off by users and developers. Thus, the specifications not only become a contract between users and developers on all issues associated with the problem which need to be solved by the system, but are also used as a blueprint to enable designers to develop the system. Although, a change in the requirements can be requested after the specification is finalised, each after-the-fact change will add to the costs and extend the schedule.

- Activity-oriented

An activity-oriented model concentrates on the process of RE itself. It is concerned with finding and executing a set of activities for requirements acquisition [And94, KS98, LK95, Rei92, SDV96, Sut96]. It is believed that engaging in these prescribed activities can help developers to capture requirements from users and represent them in formal notations. These actions are sequential in nature and provide a template for the manual management of projects.

In this type of model, the REP itself is typically separated into three stages: definition, description and validation [LK95, KS98], though different models may have their own separations. The *definition* stage draws attention to the need to understand the problem domain of the system that is to be developed. It is presumed that, after this stage, the developers are well aware of the domain knowledge. The *description* stage aims to document the understanding of requirements gained from the previous stage at an appropriate level of detail. These documents, expressed in a formal manner, are the main communication medium between developers and users. Finally, at the *validation* stage, there should be a thorough certification of the documented requirements to ensure consistency and completeness. The purpose of this stage is to detect problems in requirements documents before the documents are used by designers as a blueprint for the development of the system.

Due to the differences between individual systems in respect of scope, objectives, complexities and deliverables, and the difference between individual people in respect of their knowledge and experience, it is not surprising that developers employ many different methods to serve the purpose of each stage in the REP [LK95, Gog94]. Different people usually enact a process to tackle a problem in different ways, and even one individual may not be consistent in his/her choice of problem-solving strategy. This is because the methods used for enacting the activities of a process depend largely on the specific contexts of the people involved and the environment in which the process is enacted. If the stipulated stages of the activity-oriented models are followed, it is argued that the needs of users can be correctly captured, formalised and represented in requirements specifications.

- Decision-oriented

Unlike the context-free accounts provided by the product- and process-oriented models for REP, the decision-oriented type of model centres on the contextual aspect of decision [And94, JP93, Rei92, RL93, Rol94, STM95]. In general, the context of the domain knowledge of the software system under development is not clear in advance and is very unpredictable. Therefore, the decision-oriented models argue that developers should be able to react with flexible analysis decisions to rapidly changing situations. To achieve this, the enacted model should allow developers to advance the REP by taking advantage of the domain knowledge that they have established by analogy with the previous situations in which they have been involved. In other words, this kind of model couples the context of the domain knowledge associated with a decision to the decision itself within the REP. Such models explain not only how the process is carried on but also why there is a transformation of the output[1] of the process. As a result, the risk of misunderstanding the users' needs can be significantly reduced.

Typically, a decision-oriented model[2] views the REP as a sequence of building blocks, each of which is composed of a set of interrelated concepts, such as situation, decision, action and argument, which contribute to the context definition. A *situation* is most often a part of the output under development and serves to make sense of how a decision is made. A *decision* guiding the REP reflects a choice that developers make at a particular time in the REP. An *action* performing a transformation on the output changes the context of the domain knowledge and may reveal new situations that in turn are subjects for new decisions. *Arguments* are statements that lend support to or detract from decisions within a given context. Developers make progress in the REP through dealing with a context, that is, taking the appropriate decision in the right situation on the basis of the current domain knowledge. Developers can refine the context by considering various alternative scenarios, all of which have a bearing on the decision to be reached in this context. In other words, the aim of decision-oriented models is not only to capture the activities performed during the REP but also to record why these activities are performed and when. Such an approach is intended to make it possible to determine retrospectively what decisions were taken and what were the contexts for these decisions. With these decision-making blocks, it is argued that requirements can be understood and refined.

[1] The output of a process can be in different forms, e.g. a prototype, a conceptual schema, a logical schema or the implemented software system [Rol93]. Although the conventional output of REP is requirements specifications, here "output" is used as a broad term to include diverse outputs.

[2] This paragraph is based on the reports of a famous long-term project called NATURE (Novel Approaches to Theory Underlying Requirements Engineering) [Rol93, Rol94, JP94, JP93, JPRS94]. The decision-oriented process model proposed by this project is one of the most important models in this area.

7.1.2 Difficulties Within the REP

As described above, many models have been proposed to guide the REP. However, in many cases requirements are still gathered, analysed and implemented through a great amount of informal interaction between users and developers, trial and error, and the ingenuity of a few individuals [LK95]. One of the main reasons for this is that most models of the REP offer developers well-defined guidelines for specifying requirements rather than for solving the practical problems arising from developing requirements. For example, many models suggest that developers should collect domain knowledge from existing documents, but few of them tell developers what to do when these documents are not consistent with users' practices. The documents may stipulate a detailed procedure for users to follow, but what the users actually do in practice may reflect their precious experience accumulated during many years of work. Developers are often puzzled at having to decide which view of user practices should be considered. In effect, given different contexts, developers are required to take pragmatic actions that are best suited for solving particular problems. Since it is impossible for the suggested guidelines for the REP to take all possible contexts of users into consideration, it is not surprising that these guidelines are of limited use in the real world.

In practice, there is increasing consensus that requirements are not usually pre-existent and hidden in the experts' head waiting to be dug out and put into the specification cabinet [BCDS93]. Neither can they be completely described in any form of logical algorithm. On the contrary, requirements are designed and developed through the participants' interaction [Bub95] and are always liable to change. The simple distinction between *what* and *how* (traditional in discussing specification and implementation) is inappropriate and inadequate [Dav93, SB82, SS96] because complex requirements are rarely complete and are liable to evolve faster than the REP itself proceeds [Bub95].

As K. Ryan stated in [Rya95], RE is located at the intersection of a formally based technology and an essentially informal world. By concentrating only on the technical side, most models of the REP have left developers with a number of difficulties when these models are enacted in the real world. These difficulties mainly come from two sources: the process model itself and the participants involved in the REP.

RE is more easily described by its products than its process. Current understandings of the REP are dominated by phase-based models, whether they be product, activity or decision oriented (see Subsection 7.1.1), in which a degree of rational planning through a rigid sequence of prescribed phases is assumed. The character of each phase reflects engineering practice, that is, the application of proven methods, techniques and tools in a systematic and cost-effective fashion. However, experience shows that the REP might not be as simple as these traditional models suggest, in particular for open requirements which are poorly understood and dynamic [Blu93, Gog94, HED93].

The actual situation is usually that developers, according to the different contexts involved, exploit diverse activities to understand requirements. These activities cannot easily be invoked by following a predefined order or an algorithm. Instead, the sequence may be decided by accident and varies in accordance with different situations. Also, the activity undertaken may bridge several phases rather than be confined to a single phase in the REP. This is illustrated in the case of prototyping techniques that involve both high-level conceptual design and low-level implementation. These cannot then be assigned to one particular phase, but assist the performance of tasks in several phases: analysis, design and validation [And94, LK95]. In fact, there is a very popular trend in the software community towards regarding SSD as a non-linear phase-based life cycle [Boe88, Leh97, Pre97, Rac95]. In the same manner, the REP should not be restricted to step-by-step algorithms [Gog96, Rya95, SS96].

Furthermore, since RE is regarded as an early stage of SSD, one of the aims of most traditional models for the REP is to freeze the domain knowledge. Freezing domain knowledge is essential for sensible use of orthodox techniques in the remaining stages of SSD, such as design, implementation and validation, in the specified domain; otherwise, these techniques are not applicable. To achieve this objective, many models for the REP seek to specify the domain knowledge in a formal or semi-formal description. Such specification can not only clarify the developers' understanding of the domain but also record the users' needs for the system being developed.

Unfortunately, users' needs are usually represented by fragmentary, individual, ambiguous and unorganised statements. This is partly because of industrial specialisation: individual users, limited by their own particular professional knowledge, are only familiar with individual parts of the whole system. It is also partly because of tacit knowledge: users are often able to do things without being able to describe precisely and systematically *how* they do them [Gog96]. For these reasons, it is very difficult to prevent the domain knowledge from changing. Moreover, users' environments are characterised by uncertainty. Not only the solution domain (where the real needs of users are identified and represented) but also the problem domain (the application domain where the users' needs are produced and used) is likely to change. C. Potts's field study survey of 23 software-development organisations confirmed that in users' environments, requirements change rapidly [Pot93]. In this context, it is evident that changing domain knowledge is the norm rather than the exception within the REP [HED93, RL93].

The essential contradiction between most traditional models and their practices over the status of domain knowledge leads to a major gap between research and practice. Difficulties of enacting these traditional models in practice inevitably emerge.

One of the most important reasons for the gap between research and practice is that traditional models for the REP are context-free. Most phases of a manufacturing process are formalised by applying proven methods, techniques and tools in a systematic and cost-effective fashion. The engineering discipline confers a well-defined character upon each of these phases, so that they can be repeatedly invoked without taking account of their context. However, the REP differs from a manufacturing process: most activities invoked in the REP are inherently ill-defined and ill-structured, and are hence inseparable from their contexts, especially the social context [Gog94, HORRS95]. They are often associated with the knowledge and experience of the actors who undertake the activities, and with the contexts of the organisations and environments in which the activities are invoked. Since, in general, the rigorous formalisation of activities cannot keep pace with the rate of contextual change over time, formalising these context-dependent activities within the rapidly changing real world remains a major challenge for most process models. (For example, the activity of understanding and eliciting requirements from documents and users' statements is very hard to formalise, at least with the current state-of-the-art technology.)

In order to avoid this contextual problem, the conventional approach is to view these activities at a higher-level abstraction where the change is no longer significant. By means of such abstraction, context-free models can provide developers with instruction-like abstract activities to guide the REP, such as 'definition', 'elicitation', 'understanding', 'specification', and so on. No contextual details have to be considered in these abstract activities, since they are not the concern of these models. The product-oriented and activity-oriented models mentioned above are in principle based on this context-free abstraction. Although these models have been gradually improving and are definitely helpful for developing well-defined and well-structured software systems, some researchers have confirmed the difficulty of enacting such context-free models in the real world [Bub95, Eas93, EM95, Gog97, HED93, HORRS95]. In addition, even though decision-oriented models try to take the context into account, they are still of limited use in a real world of constant change [Rya95]. These models break up the REP into many decision-making building blocks, in each of which the temporal aspect is explicitly modelled. Within each block, developers, after making a choice, take an action to transfer the old output to a new output in order to keep up with the new context of the application domain. However, the application domain often changes too fast, so that the new context emerges before the transformation of the output is finished. This makes the new output out-of-date again in the new context.

A more serious difficulty results from poor communication between users and developers [Bub95, Eas93, Pot93, Son93, STM95, VPC98, Zav95]. It is commonly recognised that user participation is helpful for requirements development, in particular for those systems whose domain knowledge is not well understood [EQM96]. However, a well-known communication problem occurs: users have domain-specific knowledge and use the vocabulary of their domain, whereas developers are familiar with information requirements methodologies and use the vocabulary of software development. On the one hand, users may not be able to express their needs in the technical terms understood by developers. On the other hand, developers may have difficulties in understanding the professional terminology of users. For example, object-oriented techniques have been widely applied to software engineering [Boo94, CY90, Jac92, JCJO92]. Developers may be keen to understand requirements in an object-oriented fashion. However, it is very difficult for users to express their needs in terms of objects and classes [BE94, McG92, OS93, Pot93a, Zuc93]. As a result, the communication obstacle between developers and users inevitably gives rise to errors in understanding the acquired requirements. These errors, embedded in the developed system, should hopefully be detected before the system is in operation. This is especially important in the case of safety critical systems. The cost of detecting and correcting these errors is inevitably very high. Previous attempts to solve the communication problem have so far resulted in little progress towards a satisfactory solution [STM95].

In addition, most approaches to REP aim to achieve an agreed set of requirement specifications in text and diagrams [Poh93]. They seek to document requirements in a detailed fashion. However, paper is passive and can only serve as a repository for collected information. It is hard for users and developers to know whether or not there are differences between their interpretations of the same text. Many users sign off requirement specifications without fully understanding the implications. It is usually difficult for users to validate the technical documentation used by developers and designers. In fact, users can often identify their true requirements only by experiencing the operation of the system [HED93, RL93, LL94]. This is because they are familiar with the operation for solving a problem in practice, but are unable to recognise its specialised, abstract description in the specifications for the system.

Most process models for the REP fail to support group work effectively [Bub95, Eas93, JP94]. The REP is dominated by participants from different backgrounds. They may be responsible for different goals and may not be aware of each other's goals. They work together to embody their individual goals into the developed requirements. By means of the successive interactions between participants, requirements are evolved and hopefully move toward a consensus. (The trend of moving towards an agreement between participants is highlighted in one of RE's three-dimensional models proposed in [Poh93].) The evolution, especially for open requirements, must be supported by collaborative group work between participants.

In practice, the method of working collaboratively in a distributed environment has been an economically necessary and efficient means of production in modern industrial societies. Any process model for the REP should be able to support, in an effective and efficient manner, collaboration among participants in a distributed environment.

However, most models for the REP are *developer-centred*, in that users are passive and need only contribute to information provision. It is the developers, whose professional experience is in different fields, such as computer science, who determine what information is needed and how to integrate and embody the required information into the intended system [You83]. They wait for users' contributions before proceeding with further activities. Process models that are centred on the developers' tasks cannot easily support group work between all participants in a collaborative fashion.

An exception to this kind of developer-centred models is provided by viewpoint-oriented models, in which requirements are developed on the basis of different perspectives or views describing parts of the intended system [KS98, NJJZH96]. These models regard the combination of a participant and his/her view as a viewpoint, and seek to provide a framework for organising and structuring viewpoints for requirements development. Though these models implement a general feature of group work, they are carried out in a centralised manner. Developers are responsible for the integration of all viewpoints. The interaction between users and developers is to a large extent similar to that in other models, except that it can conducted in a distributed fashion.

The challenge that is addressed in this chapter is that of providing a framework for the REP which recognises the difficulties identified above and provides participants with an alternative means to support their work in developing requirements. To achieve this goal, reengineering of the REP is vitally important.

7.2 Reengineering the REP

As already explained, the main cause of the difficulties described in the previous section is that most models for the REP fail to take account of the situatedness of requirements. J. Goguen argues that requirements are situated – emergent, local, contingent, embodied, open and vague – and can only be understood in relation to the concrete situation in which they occur [Gog96, Gog94]. This situatedness demands that developers should take sufficient account of context in order to satisfy the actual needs of users, so that the developed system can solve the users' problems in the real world. The context for requirements is the real-world environment in which requirements are developed and the system is used. The environment is deeply affected by its social and organisational structure and the people therein.

Even though the issue of context has attracted widespread attention in the RE community for many years [Bub95, HED93, HORRS95, Pot93, Sid94, SS96], most models for the REP have made little progress on this issue due to the difficulty of supporting situatedness by a step-by-step algorithm [BL98, Gog97]. To avoid being trapped in the same situation that leads to the practical difficulties of traditional process models, it is worth reengineering the REP from scratch by considering the original process of requirements development without following the algorithms of any particular model.

In this thesis, the term 'requirements' is defined as "a condition or capability that must be met or possessed by a system to satisfy the condition or capacity needed by a user to *solve a problem* or achieve an objective" (paraphrasing the definition in [IEEE90] cited in Section 7.1.1). This definition acknowledges the usefulness of descriptive documentation as a resource for RE, but does not overstate the extent to which requirements can be captured via documented representation describing the behaviour, properties and constraints of the system which is to be developed [KS98, LK95, SS97]. Traditional definitions of requirements are more concerned with the advantages of using requirements as a contract between users and developers and as a blueprint for designers. The definition adopted in this thesis, which attempts to relieve or even eliminate the difficulties of enacting the REP, is more concerned with the real-world context, since it addresses the production and use of requirements for the intended system in the real world. It also recognises the importance of reconciling social and technical issues in RE.

In keeping with this definition, the process of developing requirements amounts to the process of providing solutions to identified problems that arise in conceiving the intended application in its domain. This problem-solving process should involve all relevant participants in order to collect all necessary information. An informal account of how this process appears to operate in practice follows below. Later sections will describe the way in which the situated process of requirements engineering (SPORE), when combined with DEM, can support this process.

At the outset, some *fragments* associated with solving the identified problem emerge from the subconscious minds of individual participants in the form of concepts, ideas, intentions, expectations, experiences, and so on. Many of these fragments may be ambiguous, chaotic, vague and very difficult to articulate or record. In order to clarify these fragments, participants must undertake certain activities that involve interacting with each other and introspecting about their own mental model. Very common activities include, for example, interviewing, brainstorming, video recording [HORRS95], prototyping [And94, LR91, Luq93, RL93], goal analysis [RSB98], form analysis, scenario analysis [Hol90, WPJH98], and so on. For the sake of convenience, as in Chapter 2, the term 'interaction' is used to refer to both interaction and introspection.

When participants start to interact with each other, their individual fragments of knowledge change: some of them disappear, but some new ones also emerge. The most significant change arising from the interaction is that some of these fragments move towards being unambiguous, ordered and clear. The interaction is continued until some fragments finally become intelligible to all participants, and can be identified and represented in terms of primary elements agreed and apprehended by participants. These elements could, for example, take the form of objects and classes in an object-oriented model [Boo94], entities and relations in an entity-relation model [Che76], viewpoints in a viewpoint-oriented model [KS98], or simply statements in natural language. No matter how they are represented, the intelligible elements are not fixed but are instead liable to change.

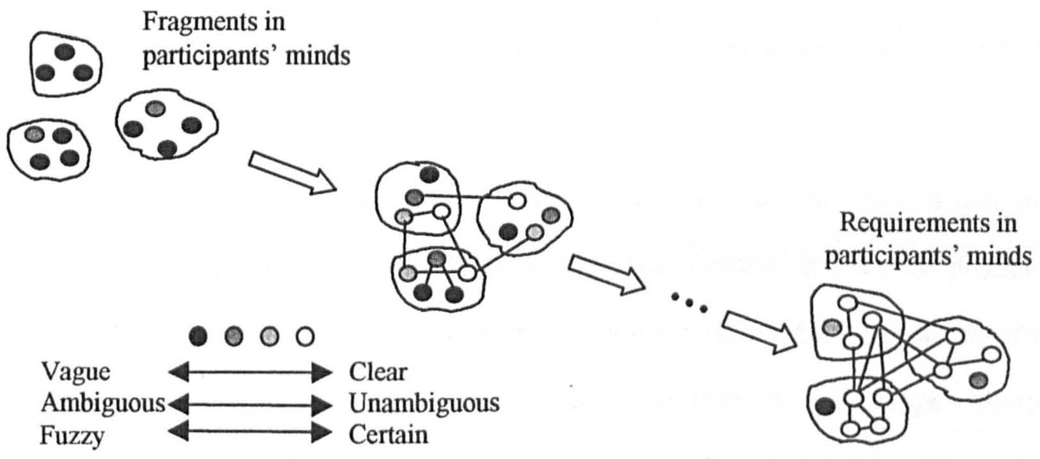

Fragments in participants' minds
Requirements in participants' minds
Vague ↔ Clear
Ambiguous ↔ Unambiguous
Fuzzy ↔ Certain

Figure 7-1. Requirements formulation: from fragments to requirements

As the interaction continues, more and more elements are obtained and coupled with the existing ones to form a web of interconnected elements. Incrementally, this web should converge to a provisional solution to the identified problem. At that point, requirements providing a solution to the identified problem are developed. Whilst there is no such convergence, the interaction must be continued until a provisional solution emerges. Otherwise, requirements for the identified problem cannot be obtained and the success of SSD becomes problematic. Clearly, the provisional solution is not fixed. Its integrity could be undermined or its form changed at any moment as yet more intelligible elements emerge. Section 7.3 will explain how EM can support the process of composing fragments into a provisional solution illustrated in Figure 7-1 through 'structural coupling' (as described in Section 3.2).

Such a REP based on structural coupling, whereby new elements are dynamically coupled with the existing elements, is difficult to achieve by the traditional top-down or bottom-up approaches that are typically used in models for the REP. Top-down decomposition presumes that the organisation of fragments is broadly established, and is not applicable until the requirements of the developing system are sufficiently well understood [Blu93]. Bottom-up analysis generates fragments of the requirement that are exactly prescribed, and are therefore not suitable for representing vague, ambiguous or fuzzy requirements.

One of the principles of reengineering the REP is to incorporate activities that are normally undertaken by participants together with their context. It must be possible within the REP to accommodate any interaction which it is within the competence of a participant to choose as the most effective way to improve the current provisional solution. Neither specific actions nor their sequence are rigidly stipulated in advance for serving such a purpose. Instead, the development of requirements is fulfilled through what L. A. Suchman has called 'situated actions', in which performance is matched to the specific task situations existing at the time [Suc87]. The fact that the interaction between participants is appropriately situated contributes significantly to the growth of understanding and experience as the interaction continues.

Requirements cannot be isolated from the subsequent development and operation of a software system. As explained earlier, requirements are closely associated with the context of use in which the system is operated to solve users' problems. On the one hand, the system is implemented in order to provide users with the solution represented by the description of the developed requirements. On the other hand, the developed requirements are validated and clarified through the operation of the system in the real world, and the operation may in turn bring out the need for new requirements or a change in old requirements. The contextual dependence between requirements and the system forces the REP to be intertwined with the process of SSD in a symbiotic fashion as illustrated in Figure 7-2. The interdependency between SSD and the REP fits in well with the increasingly popular arguments that the REP is never complete but should be continued throughout the whole life cycle of SSD [BL98, CGC96, Gog96, JP94, Rya95]

The metaphor shift in requirements development is analogous to that in software development. In [Bro87], Brooks highlights the fact that the *building* metaphor, which

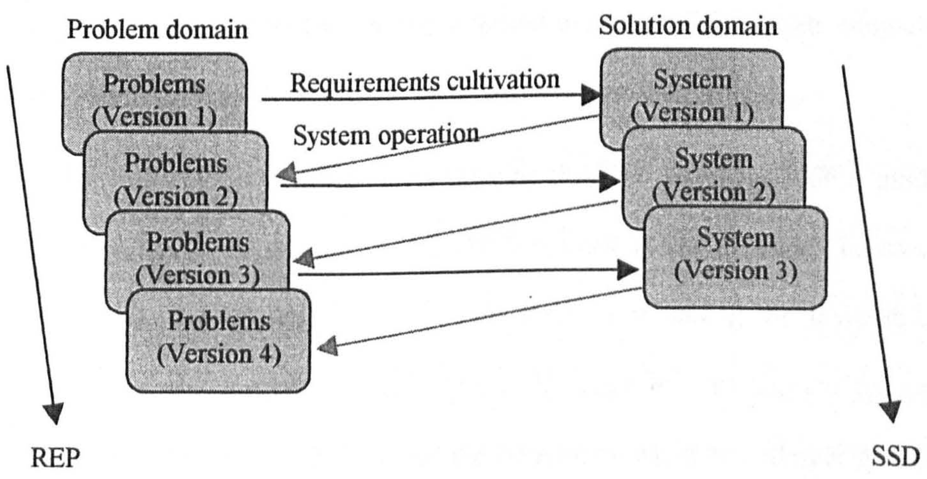


Figure 7-2 The interdependency between SSD & REP

likens the way in which software is constructed to a building process, has outlived its usefulness, since software systems have become so complex that they cannot be fully specified and designed in advance. He suggests that any software system should be grown by incremental development. In fact, the *growing* metaphor should be also applied to requirements development, due to the contextual dependence described above. It is more appropriate to think of developing incremental requirements as opposed to eliciting or acquiring ones from sources directly, which is the usual method employed by many traditional models for the REP [DBP93a, DBP93b, FHW94, Rei92, SDV96, Sut96, KS98]. The concept of incremental development is consistent with the concerns of changing requirements [Gog94, HED93].

It is evident that the interaction between participants provides the main impetus for the process of developing requirements discussed here. A well-known drawback of most RE models is that it takes the effective interaction between participants for granted. C. Potts's field study shows that interaction breakdown is the major problem in the REP [Pot93]. In fact, the information arising from the interaction between participants is the main resource for requirements development. A process model should not hinder the emergence of the essential information, but should facilitate such emergence by supporting the interaction between participants as much as possible. Recognising this need, therefore, the alternative framework proposed in Section 7-3 exploits computers as the best communication medium for achieving this purpose.

Another important principle of reengineering the REP is that the REP is guided by participants and not by a process model. Within most models, human behaviour is embedded into the mechanism of enacting the REP by assuming the invariability of human factors and the context of requirements. Although this assumption reduces the uncertainty surrounding human beings and their environment, it accordingly generates a gap between research and practice, thereby leading to the difficulties discussed in the previous section. This is because the inflexible mechanism hinders the essential ability of human beings to accommodate themselves to the rapidly changing environment. As the REP is located at the intersection of formal and informal, of objective and subjective, and of technical and non-technical approaches, it needs to rely to a large extent not only on the participation of human beings but also on their accommodating nature. Any process model should recognise the existence of uncertainty, and make due provision to autonomous interpersonal interaction to this end. For instance, it is not in general appropriate to presume that interpersonal interaction is so reliable that it can be replaced by a mechanism. It may also be necessary to allow human intervention to mimic the unreliability of mechanism. This echoes Tully's concern for enacting a software process model as a symbiosis of human agent and computer that does not hint at particular roles for either partner (see [Tul88] cited in Section 2.2.2).

In spite of the importance of individual experience and knowledge, the transition from informal, fuzzy statements to formal, unambiguous requirements usually needs to be carried out through interactions between all participants. The different relationships that can shape this interaction have provided the foundation for most models and methods in requirements engineering. There is a particularly significant distinction between coordinative and subordinative relationships. A *coordinative* relationship stresses the importance of user participation in design, and postulates responsibilities for all the participants. A *subordinative* relationship assumes that users should be responsible for providing all the knowledge required by designers because only they know what they want.

Traditional patterns of interaction favour relationships of these two kinds, since they presume a clearer separation between analysis, design and use that modern business practice and associated information technology promotes. In the development of information systems, it is standard practice for feedback from users to affect the product. This feedback operates both in validating and debugging the original design, and in its subsequent enhancement. In the concurrent engineering of other products, the use of information technology has subverted the rigid sequential stages of the traditional design process. The ease with which design representations can be visualised and modified enables wider and more opportunistic intervention from all kinds of participants. In these contexts, the interaction for developing requirements becomes exceedingly subtle [SKVS95]. In effect, the design of a software system and the shaping of the requirements satisfying the needs of users often have to be negotiated in a symbiotic fashion. The interaction amongst all participants that is appropriate in this context will be characterised as a *collaborative* relationship.

A useful analogy can be drawn between the relationships of all participants for developing requirements and the relationship between a teacher and pupils in a classroom. A subordinative relationship resembles the context of a lecture context, where the teacher imparts knowledge in the role of the expert, and there is no participation from the pupils. A coordinative relationship, in which a rigid agreement sets out the respective responsibilities of designers and users, resembles a tutorial context in which the teacher imparts knowledge through a prescribed pattern of small presentations, exercises for the pupils and evaluation of their performance. A collaborative relationship is concerned not only with responsibilities but also with expectations, beliefs and other psychological states that make developing by learning more feasible and powerful [DL91]. The appropriate context for such interaction resembles a seminar, where the precise learning goals are not set out initially, and the knowledge content is shaped dynamically by the contributions of the participants. In the same way that all three paradigms can be used in one educational context, each of the three different kinds of relationship amongst all participants can be represented in the same process of developing requirements.

Collaborative relationships are concerned with interaction that is socially distributed. They engage with issues of subjectivity and objectivity associated with distributed cognition [Hut95] and common knowledge [Cro94, Edw87]. This involves a reappraisal of distinctions that are taken for granted in other contexts. There is a potential for several kinds of conflation:

- between the roles of all participants,
- between the properties associated with individuals and with artefacts,

- between the characteristics to be attributed to the internal mind and to the external environment.

In a collaborative relationship, there is typically no possibility of relying entirely upon closed-world representations and preconceived patterns of interaction. The interaction between all participants has to be situated intelligent interaction that can only be planned in advance to a limited degree, and domain knowledge for the process of developing requirements emerges on-the-fly.

Supporting the situatedness of the REP is not a trivial task. Firstly, an environment that enables participants to interact with each other in a collaborative manner is necessary. All participants involved in the REP share the responsibility of developing the requirements that satisfy the users' need to solve problems in the real world. Not only developers but also users are responsible for the success of requirements development. The most common method, called introspection, that is embedded in most process models for developers to collect information about the users' needs and habit, cannot serve the purpose of supporting collaboration [Gog97].

Secondly, each participant must be sufficiently qualified to make his/her actions accountable to others. Considering the above example of a classroom, common knowledge will obviously not be established unless the teacher is capable of taking actions which make sense to a pupil, and *vice versa*. This may entail a coordinative relationship, or even a subordinative one in which the responsibilities of each participant are stipulated. Similarly, within the REP, if a participant is incapable of interacting with others, it is inevitable that information pertinent to that participant will be missed. Accordingly, this is likely to give rise to major problems. Fortunately, advances in end-user computing have increasingly reduced the problem of incapability [DL91].

Needless to say, the crux of supporting the situatedness of the REP resides in the construction of the provisional solution in an open-ended, interactive fashion. Obviously, if a fixed problem domain of the application is specified, it is not too difficult to find a solution by means of so many existing tools and techniques that contribute to the search process. However, in order to cope with the dynamics of context, the solution to the identified problem, and even the problem itself, must keep changing in response to new domain knowledge emerging from the interaction between participants. Therefore, the solution needs to be constructed incrementally and interactively. Few tools and models support the construction of a changing solution, or, more precisely, of changing requirements. This is partly because of the technical difficulty in coupling the old solution with a new context.

One of the main contributions of this thesis is to demonstrate that the computer-based interactive modelling technique discussed in Chapter 4 has the potential to support the situatedness of the REP in a significant way. The next section introduces a novel framework for the REP motivated by the perspective on reengineering the REP described here. The principles and concepts of DEM are applied to the framework in order to provide a human-centred, computer-based environment to support the REP.

7.3 A Situated Process of Requirements Engineering

Due to the need to model the real world in which the target systems reside, to manage many fragmentary yet interrelated requirements statements, and to cope with changing assumptions and perceptions of requirements, the REP must be situated and human-centred. This section provides a novel framework for the situated process of requirements engineering. First, the framework called SPORE is proposed. Within this framework, people participating in the REP are able to cultivate requirements through collaborative interaction with each other in order to solve the identified problems, instead of searching for requirements in the 'jungle' of users' needs. The environment supporting the framework is established by applying the principles and concepts of DEM. By means of a computer-based *interactive situation model* (ISM), participants can collaboratively interact with each other to 'grow' requirements in an incremental development fashion.

7.3.1 A Framework for the REP

According to the definition of requirements and the principles of reengineering the REP presented in the previous section, requirements may be seen to provide solutions to identified problems. The REP begins in the problem domain associated with the requirements of the developing system. This domain is generally informal, situated and open to the real world [Gog96, Blu93]; hence it cannot be specified completely in advance. Instead, the domain is represented by a situated, provisional, subjective, but computer-based, model. It is *situated* because it is represented as organically connected to its referent (the domain). Such connection is achieved by being continuously open to revision through a comparison between the experiences of interaction with the domain and those of interaction with the model. Accordingly, the model is not divorced from the domain, as required for a preconceived 'system' with boundaries made sharp by some form of idealisation or abstraction.

A human-centred framework, called SPORE, for building situated models for the process of requirements engineering, is depicted in Figure 7-3. Key problems of the domain are identified by the participants within the grey box in Figure 7-3 with reference to their concerns for the functional, non-functional and enterprise attributes of the

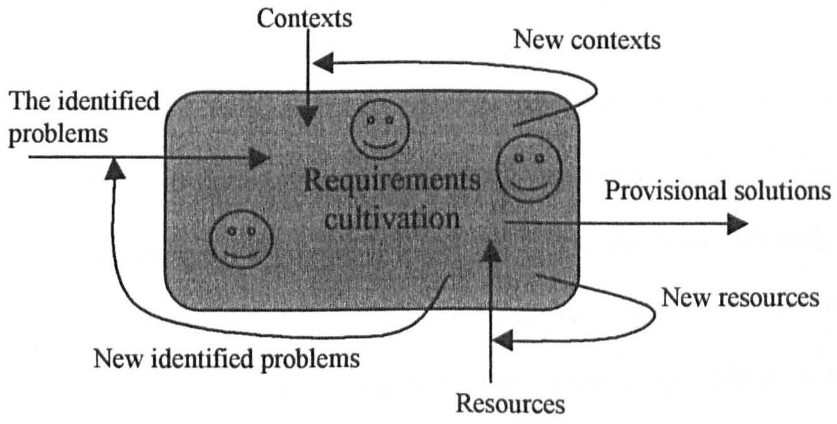


Figure 7-3. The SPORE framework

developing system. The identification of problems can occur at any time during the REP and is never regarded as being completed. Another two inputs of the SPORE model are the available resources and the current contexts. The resources, such as documents, technology and the past experiences of participants, are used by participants to facilitate the creation of the SPORE model's outputs. The contexts, such as the organisation's goals and policy, and the relationships between participants, act as motives and constraints for the participants in creating the outputs. These three kinds of input may impact on different parts of the model at different stages of its evolution. The arrows ending at the inside of the grey box in Figure 7-3 convey this idea.

A SPORE model has outputs of four kinds. The most important one consists of solutions to the identified problems. These are developed by participants on the basis of the available resources and the current contexts. Moreover, the other outputs, including new contexts, new resources and new problems, combine with their earlier versions and form new inputs for creating the next output. That is to say, all these contexts, resources and identified problems, even during the development of solutions, are still modifiable and extensible. In view of this, participants can develop requirements in a situated manner to respond to the changes in the contexts, resources and even the problems themselves. This implies that requirements are apt to change all the time and thus are never completed. In this respect, the SPORE framework is consistent with J. Goguen's concern for the situatedness of requirements [Gog94].

The SPORE framework determines neither specific activities nor their sequence. In many cases, several problems can be identified simultaneously. Some may be very difficult to solve under the current contexts and resources, but others are not. Some are interdependent and need to be solved concurrently, but some can be solved independently. Different problems are likely to need solution by different methods. No rule or algorithm can be postulated in advance to take all these factors into account. A generic strategy for taking actions is 'divide and conquer', where the highest priority is to undertake action for the easiest problem. But this is not a golden rule. Participants must still take their current context and available resources into consideration in order to cope with the diverse issues arising from the development of solutions.

The central activity in the SPORE framework is the requirements cultivation, in which participants interact with each other and with their environments to develop requirements, i.e. the emerging solutions to the identified problems. The term 'cultivation' is used to convey the idea that requirements (like plants) should grow gradually rather than be conjectured from their initially fragmentary, chaotic and rapidly changing states. It also emphasises the use of deliberate design activities by participants on the basis of their contexts and available resources in order to develop requirements in an effective and efficient fashion. Some models for the REP assume that requirements are pre-existent but hidden in some sources [LK95, SS97], just like grown plants in a huge jungle. The purpose of building these models is to search for (elicit) the right plants (requirements) in (from) the jungle (available sources, such as documentation and the expertise of users). Typically, the jungle is a mixture of numerous kinds of elements that are fluctuating in response to the changes in their environment. It is clear that searching in such a jungle for one element, which has never been seen before and might keep changing all the time, remains a very difficult challenge [LK95, Bub95].

The concept of requirements cultivation, unlike that of searching for requirements in a jungle, refers to the 'growing' of requirements for the developing system by participants themselves through their collaborative interaction. The cultivating process focuses on neither the problem domain nor the solution domain but instead on the interaction through which participants seek to solve the identified problems on the basis of their current context and available resources. For example, let us consider the development of a simplified automated teller machine (ATM) which contains an embedded software system to drive the machine hardware and to communicate with the bank's customer database. In order to acquire the requirements of the software system, a problem of accessing the service is identified. The participants relevant to the identified problem, such as customers, bank staffs, machine designers, database managers, security officers, software designers and so on, must work together to solve the problem. The solution is not located in someone's head but is socially distributed across all participants. Also, it is formed and shaped through the iterative and creative activities invoked by participants in their interaction with each other. In this sense, requirements are cultivated by participants through a variety of purposeful activities.

The work of requirements cultivation can be focused further on individual participants. Within the REP, each participant has his/her own individual insight into the identified problems and their solutions. This insight is based on the participants' various contexts and available resources. It is clear that individual insight is often of limited use and inevitably has a bias. For example, in the ATM example mentioned above, for reasons of security, bank staff may demand more rigorous security checking for access to the service provided by an ATM machine. But from the customer's viewpoint, the convenience of using the service might be the main concern.

7.3.2 Applying DEM to SPORE

There are several approaches to cultivating requirements, but one of the most efficient and cost-effective ways is by computer-supported modelling. Conventional computer-based modelling is better oriented towards assisting subordinative and coordinative, rather than collaborative, relationships. To fully support the collaborative interaction between participants discussed above, it is essential to establish an individual ISM for each participant within a distributed environment that:

- allows data about requirements to be collected in such a way that participants are engaged in activities in their customary context [Gog96, LK95];
- makes it possible to explore and experiment with individual insights for different participants;
- provides for open-ended interaction.

Given the principles and concepts of DEM described in Chapter 4, each participant can construct an ISM within an interactive, distributed environment supported by dtkeden. According to the principles of SPORE, the cultivation of requirements has to stem from a representation of those identified fragments that are pertinent to the identified problem being addressed. This representation will take the form of a *seed* ISM that incorporates matter-of-fact observations of the current context. An ISM to represent these observations will supply a visual representation for those identified fragments. Participants can thus interact with their own ISM to extend, expand and explore their individual insights through 'what if' experiments resembling interaction with a spreadsheet. The accumulated results of experiments not only change the participant's individual insight immediately but also are stored in the memory of the participant. The latter is the most important resource used by the participant for taking situated actions.

In effect, this experimental interaction, using the computer as a modelling medium, can – when integrated with other methods – provide more accurate and more powerful resources for developing solutions to the identified problems. In this sense, interacting with the computer model becomes a very critical situated action for creating a resource for further actions. Figure 7-4 illustrates how a participant can take situated actions by interacting with his/her computer model and/or external environment on the basis of various contexts and available resources to explore individual insight. The insight into the identified problems and their solutions evolves with the interaction. This will be illustrated with reference to developing the requirements for an ATM system, as described in the next section.

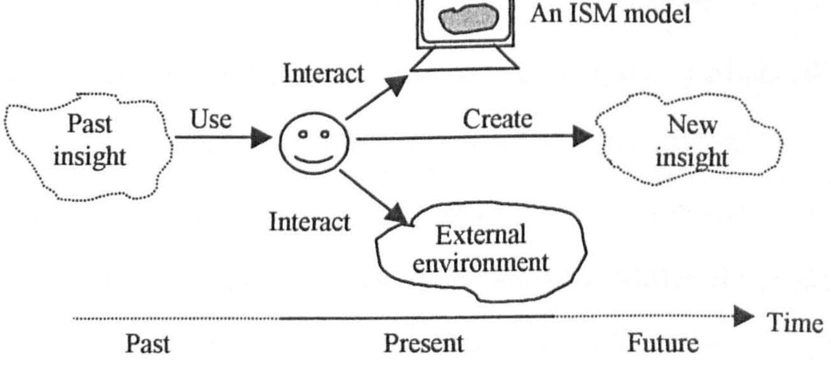


Figure 7-4. The experimental interaction of a participant

The ISM, with visualisation corresponding to the observed real world, plays an enabling role in SPORE. Given the visualised scenarios, participants can 'preview' the system in the current context to enhance and explore their understanding of the developing requirements. More importantly, *experiencing* these kinds of visualised scenarios can give users greater confidence that they understand their actions and those of the intended system. The confidence prompted by experience is very significant and useful for getting rid of user resistance in ISD. This concern is consistent with the result of an empirical study proposed in [Kuw93] in which users and developers are mostly concerned with how users themselves can recognise how the system will behave.

To some extent, an ISM is similar to a prototyping model [And94, BD93, DF98, LR91, Rei92, SAGSZ97]. Both focus on "evaluating the accuracy of problem formulation, exploring the range of possible solutions, and determining the required interaction between the proposed system and its environment" [LR91, p.77]. They are both working models, so that their users can have operational experience of what the system should do and how it should look. This experience enables more effective communication between participants to help requirements development, to reduce the risk of misunderstanding and to clarify a designed solution to an identified problem.

However, unless it supports the collaborative interaction between participants, a prototyping model is of limited use in exploring the domain knowledge of the developing system. Typically, a prototype demonstrating a part of the developing system is used in order to understand requirements for providing a solution to an identified problem. Users and developers are separately responsible for model validation and model development. Any improvement of the provisional solution must be fed back to developers so that a new prototype can be reconstructed in a traditional fashion. This 'backward' reconstruction is very different from the 'forward' reconstruction of an ISM in which any domain knowledge emerging from the REP is directly implemented into the ISM in an interactive manner. Feedback is too late and too passive. It constrains participants from exploring unknown territory.

What makes ISMs particularly powerful in this context is that they enable participants to interact with each other in an open-ended, interactive manner. Through the network communications facilities, all ISMs are connected together to create an environment that can be viewed as a radical extension and generalisation of a distributed multi-user spreadsheet. The connection makes it possible to propagate the experimental interaction of each participant with his/her ISM to those of others, so as to consequently affect their individual insights. Participants can interact with their own ISM privately by making a variety of definitions in order to explore their own insight into the identified problems and corresponding solutions. They can also interact with others and their ISMs by propagating definitions through communication networks. The propagated definitions first change the visualisation of others' ISMs (given suitable authorisation) and consequently may change their insights as well. Thus, participants can collaboratively interact with each other through their ISMs and communication networks. Figure 7-5 shows this collaborative working environment.

Within the collaborative working environment described above, a working understanding of the identified problems and their corresponding solutions, that is, of the requirements, is established. This working understanding is distributed across participants rather than in an individual mental representation. It is not expressed by a literal specification that establishes a fixed relationship between the individual ISMs and their referent, but as a commitment to constrain the interaction between participants in a way that respects their common insight, but does not prevent new distinctions from emerging.

The working understanding is then cultivated, that is, grown incrementally, through the successive interaction between participants for exploring and integrating individual insights. Generally speaking, greater consistency between the individual insights is associated with a better working understanding. For this reason, participants continually refine their interaction with a view to achieving more coherence and consistency. This process is open-ended, and consistency can only be achieved in relation to some restricted work activities and assumptions about reliability and commitment. In practice, there are likely to be singular conditions under which a higher viewpoint must be invoked to mediate or arbitrate where there is conflict or inconsistency. The 'global view' perspective depicted in Figure 7-5 represents such an overall viewpoint. It could also be the view of a requirements engineer when acting in the role of negotiator between differing or incompatible insights.

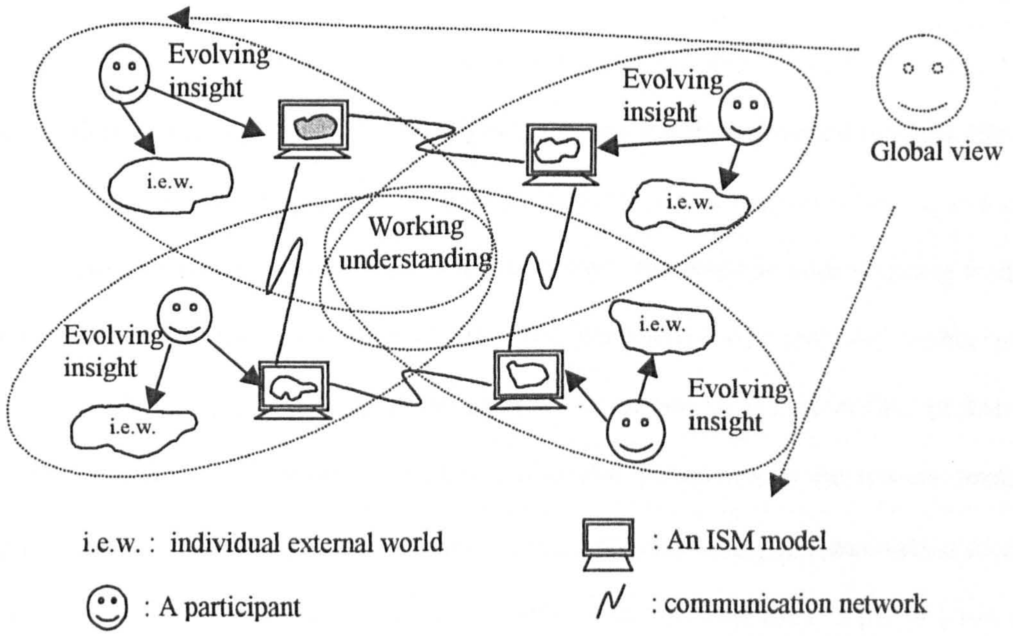


Figure 7-5. A Collaborative working environment for cultivating requirements

The most important benefit of interacting with computer models is to make individual insights and the working understanding between participants *visible* and *communicable*. Of course, most models for the REP involve the interaction between participants in order to facilitate the establishment of the working understanding, for example by requirements elicitation and validation [LK95]. However, the working understanding within these models is invisible and incommunicable. Even given a requirements specification, the visibility and communicability of the working understanding are still restricted to the boundaries of language description and comprehension. Also, paper documentation, as used in a repository or archive, fails to support the needs of its users in exploring and integrating information. In practice, it is very difficult to keep requirements specifications synchronised with the working understanding between participants [DS97, LR91, Luq93], because the latter emerges from experimental interaction and evolves much faster than the evolution of specifications.

In contrast, the experimental interaction between computer models invoked by participants immediately changes the visualisations of these models. The change leads quickly to the evolution of individual insights as well as to a working understanding. The synchronisation between the evolution of computer models and individual insights allows participants to 'see' the viewpoints of other participants and to 'communicate' with them by interacting with their own ISM. In the same manner, the working understanding is also embodied in these computer models. From the perspective of users, the visible and communicable computer models illustrating the solutions to the identified problems represent a crucial contribution to understanding that complements the passive textual descriptions of conventional specifications. In this sense, ISMs are communication media through which the commitments between participants are conveyed. This account of ISMs as 'communication media' fits in well with the view expressed by T. Winograd and F. Flores in [WF86, p.79]: "computers are not only designed in language but are themselves equipment for language".

Whilst the interaction between participants through ISMs has obvious advantages, it should not be thought that other more traditional methods of communication between participants need be foregone and replaced by the computer-mediated communication through ISMs. On the contrary, those methods, such as face-to-face communication, assume even greater importance because they are the best means for compensating for the limitations of computer-mediated communication, such as the absence of the normal social cues inherent in group work [Smi97].

7.4 Two Examples of SPORE

Two examples using the SPORE model for requirements cultivation have been studied. One is the software system embedded in the automated teller machine (ATM). This example has been studied by using the viewpoint-based requirements method (VORD) in [KS98]. A comparison between two models (i.e. SPORE and VORD) for developing requirements is also provided. In the second subsection, another example, relating to a warehouse information system, is given. The example attempts to compare the SPORE model with the use-case approach proposed in [JCJO92].

7.4.1 An ATM Software System[3]

The system embedded in an automated teller machine (ATM) has been used as an example by several researchers in the field of requirements development [KS98, RSB98, SDV96] The ATM system accepts customers requests, produces cash and account information, drives the machine hardware and communicates with the bank's customer database. Multiple participants are involved, such as bank tellers, bank managers, ATM operators, customers, hardware designers, bank database managers, bank security officers and so on. It is a good example to show the development of requirements in a distributed fashion.

The principles and concepts of SPORE have been used to cultivate requirements for the ATM system by the present author. First, participants, such as customers, bank

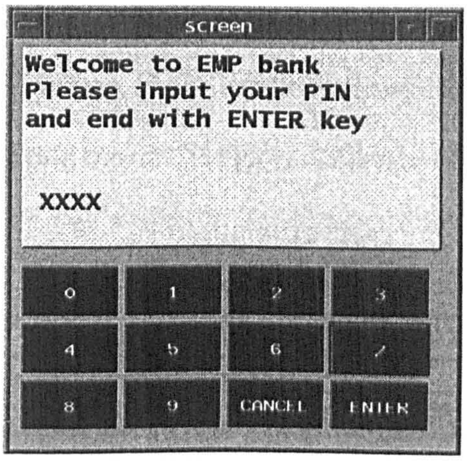
[3] Due to the limitation of the author's research time, the system is not completely accomplished (only the part discussed in this subsection has been finished).

tellers, bank managers, database managers, ATM hardware designers and software designers, are involved. Then, problems are identified by them in order to highlight their respective concerns. Table 7-1 shows some of these problems. It should be noted that they are not frozen, so new problems can be added and old problems may disappear or be changed. Also, it is unnecessary for them to be solved sequentially or independently. A typical strategy for solving problems is 'divide and conquer', where the easiest or the most important problem has the highest priority.

Participants	Identified problems for developing the system embedded in an ATM
Bank tellers	Start-up and shut-down an ATM machine Gaining access to an ATM for administrative services Available services Gaining access to customers' account details The identification of customers The notification of notes deficiency
Customers	The identification of customers Available services Acceptable response time
Hardware designers	Allowing customers to gain access to an ATM Allowing tellers to gain access to an ATM Security control Supporting available services
Software designers	Driving and communicating with hardware devices Interaction with tellers and customers Communication with bank database Supporting available services
Bank managers	Security control of the ATM The identification of customers Accuracy and performance of the requested services Reports of each transaction Reports of each administrative access Cost of each ATM
Bank database designers	Providing and updating the details of customer's account Recording each transcation Security control over database access

Table 7-1. Some problems identified by participants for an ATM system

To illustrate the collaboratively experimental interaction between participants, one of the identified problems (see. Table 7-1) is considered as an example: the identification of a customer accessing an ATM ('the ID problem'). Based on their different contexts and resources, each participant creates his/her own seed ISM and prepares for the


Welcome to EMP bank
Please input your PIN
and end with ENTER key

XXXX

Figure 7-6. The ISM of a bank customer

experimental interaction with others. Figure 7-6 shows a snapshot of the seed ISM of a bank customer, as developed using dtkeden. A collaborative working environment for participants in this example has also been constructed (cf. Figure 7-7, for which Figure 7-5 is the archetype). Here requirements engineers interact through 'God's view' to guide the negotiation between participants and the integration of their individual insights in the solution of the ID problem. Table 7-2 illustrates some of their initial insights expressed

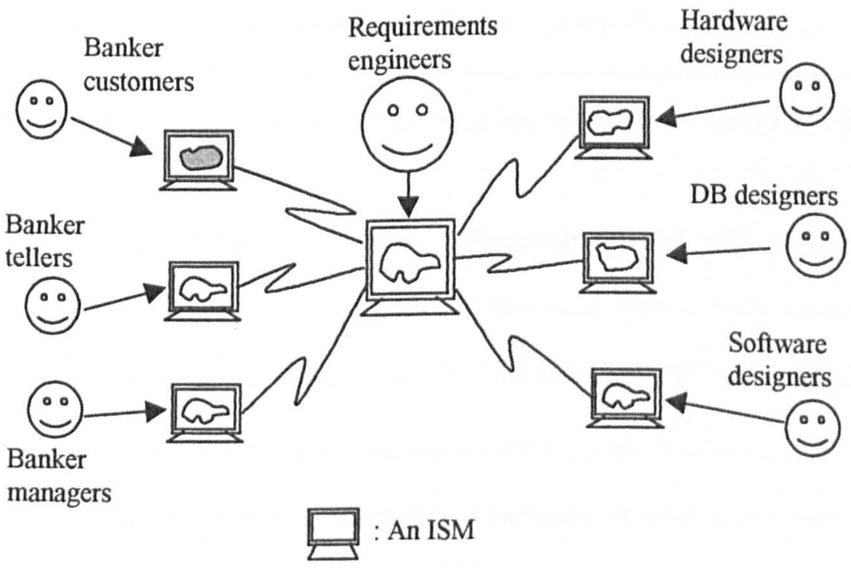


Figure 7-7. A collaborative working environment for an ATM system

using the EM definitive notations. For example, the bank manager is concerned with the safest control of access to the service. Hence, confirmation of user identity, to include the card, the card-holder and the card account, is rigidly demanded.

Participants	Individual insights

Bank tellers	customer_account_NO is card_NO;
	customer_ID is 6-char-PIN_received;
	customer_confirmed is check_customer_ID(customer_account_NO, customer_ID);
	accessing_to_service is customer_confirmed;
	current_screen is
	(accessing_to_service)? send_out_services_manu() : send_out_try_again_screen();

Customers	customer_account_NO is card_NO;
	customer_ID is 4-digit-PIN-received;
	customer_confirmed is check_customer_ID(customer_account_NO, customer_ID);
	accessing_to_service is customer_confirmed;
	current_screen is
	(accessing_to_service)? send_out_services_menu() : send_out_try_again_screen();

Hardware designers	checking_card_logo is (card_being_inserted)? (check_card_logo()) : FALSE;
	card_confirmed is card_logo_confirmed;
	card_magnetic_NO is (read_card_NO)? read_magnetic_NO() : FALSE;
	received_4_digit_PIN is (get_4_digit_PIN)?input_4_digit_PIN() : FALSE;
	received_6_char_PIN is (get_6_char_PIN)? Input_6_char_PIN() : FALSE;

Software designers	get_6_char_PIN is (card_confirmed)?send_input_6_char_PIN_screen(): send_invalid_card_screen();
	read_card_NO is card_confirmed;
	card_NO is card_magnetic_NO;
	6_char_PIN_received is (get_6_char_PIN)? Receive_6_char_PIN() : FALSE;
	get_4_digit_PIN is (card_confirmed)?send_input_4_digit_PIN_screen(): send_invalid_card_screen();
	4_digit_PIN_received is (get_4_digit_PIN)? Receive_4_digit_PIN() : FALSE;

Bank managers	*accessing_to_service is card_confirmed & customer_confirmed & account_confirmed;*
Bank database designers	account_confirmed is check_customer_account(customer_account_ID);

Table 7-2. Individual insights of different participants for an ATM system

On the basis of individual seed ISMs, participants interact with each other for cultivating requirements. For example, it is found that bank tellers, bank managers and customers have different perspectives on the ID problem (see the italicised entries in Table 7-2). For the sake of convenience, customers prefer to be identified by the account number on the inserted card together with a 4-digit personal identification number (PIN). For safety reasons, bank tellers instead suggest using a 6-char string as a PIN code for the verification, since many customers use their birthdays as PIN codes. However, bank managers have a broader insight into the identified problem and wish to take the status of the customer's account and the cost of building each ATM into consideration. In addition, since these insights of customers, bank tellers and bank managers impinge on the hardware and software systems of the ATM, the bank database, hardware and software designers and database managers are also involved in the interaction for cultivating requirements.

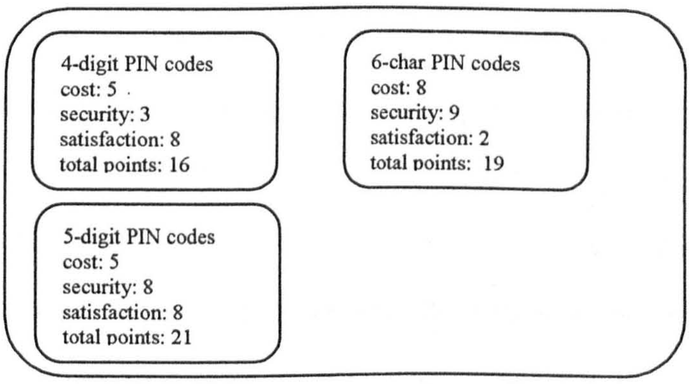


Figure 7-8. The ISM of a bank manager (snapshot)

At this stage, a new problem is identified: the data format of PIN codes ('the PIN problem'). To address this problem, the contexts and resources around participants are changed and mutually affect their ISMs in a situated manner. For example, for hardware designers, two different panels for inputting PIN codes have to be provided in order to support the conflicting ideas of bank tellers and customers. This provision can give insight into the cost of building each ATM, which is a main concern of the bank managers. Another concern of 'customer satisfaction' may also be introduced by bank managers to measure the feelings of customers about using both kinds of panels. The measures[4] of these concerns are shown in Figure 7-8. These situational changes highlight the fact that requirements are situated and depend greatly on the context and resources.

[4] The created ISM may involve the implementation of a decision support model, but the details are out of the scope of this thesis. It is assumed here that these measures can be accomplished and obtained by the bank manager through certain ways, when such measures are identified.

Now 'what if' experiments can be invoked. For example, customers can make use of their ISMs to explore different data formats of PIN codes, as when taking into account upper case and lower case characters in PIN codes.

If an agreed solution to the PIN problem cannot be obtained, it might either be left unsolved or managerial authority might be invoked to make a decision that suits the current context. In the former case, an unresolved conflict between participants occurs. Conflicts are not allowed in most traditional models, and are always viewed as errors that need to be corrected. In practice, a conflict need not always be regarded as an error. Conflicts disclose possible alternatives and are actually a very useful resource for making a decision. For the PIN problem, the conflict reveals individual concerns about PIN codes from different viewpoints. Customers focus on convenience of use, bank tellers pay attention to security control, but bank managers have an economic account of cost. In order to highlight its importance, the conflict is deliberately left unresolved here.

In a similar manner, it is clear that many problems can emerge and be either solved or unsolved. Details are beyond the scope of this thesis. It is supposed that some problems, such as the maximum number of permitted attempts to enter PIN codes, cancellation of PIN code input, validation and retention of a cash-card, and the diverse messages displayed to customers, have been identified and solved during the interaction between participants. It is not necessary for these problems and their respective solutions to be developed in a particular sequence; they can be addressed as they arise in particular contexts in a responsive manner. In this process, a provisional solution to the earliest problem – the ID problem – is obtained through collaborative interaction between participants.

At this point, with the ID problem provisionally solved, the PIN problem (still unresolved) may arise again, since a new context emerges: if the number of attempts to enter a PIN code exceeds the permitted number of attempts, the cash-card will be retained. In such a context, customers may strongly object to using 6-char PIN codes by awarding lower points of satisfaction. The change will immediately be propagated to the ISM of bank managers so that bank managers will understand the objection of customers to 6-char PIN codes. On the other hand, from the security perspective, a 4-digital PIN code is too simple to protect against fraud, especially given the evidence that customers are prone to use someone's birthday as a PIN code. Brainstorming activity supported by informal social interaction and 'what if' experiments thus commences. A new proposal that takes both viewpoints into account may be developed, for example, using a 5-digit number as a PIN code and providing customers with the facility to change their PIN codes on ATMs. This new proposal not only changes the provisional solution to the ID problem, but also has an impact upon the problem of 'available services' identified by banker tellers and customers in Table 7-1.

This special case of providing a solution to the ID problem illustrates the fact that changing requirements is the norm in developing requirements, even in such trivial a problem. Recognising this fact, SPORE deals with the rapid change of the identified problems, contexts and resources by the collaborative interaction between participants in a situated manner. Just as the knowledge of a human being grows in everyday life, requirements are grown by participants through interactive, iterative and creative activities on the basis of the principles and concepts of SPORE.

To explore the difference between SPORE and traditional process models, a viewpoint-based model called VORD is chosen. This is because some of the details of using VORD to formulate the requirements for the ATM are given by the authors of VORD and can be found in [KS98]. With these details, a fair comparison can be made, since the possibility of using VORD incorrectly can be eliminated. More importantly, VORD is a viewpoint-based model whereby requirements are principally developed in a distributed manner [FS96]. Within VORD, the information needed for developing requirements has been separated to different viewpoints from diverse perspectives. The general feature of supporting group work, which is given little or even no support in most models for the REP, is one of the main concerns in SPORE.

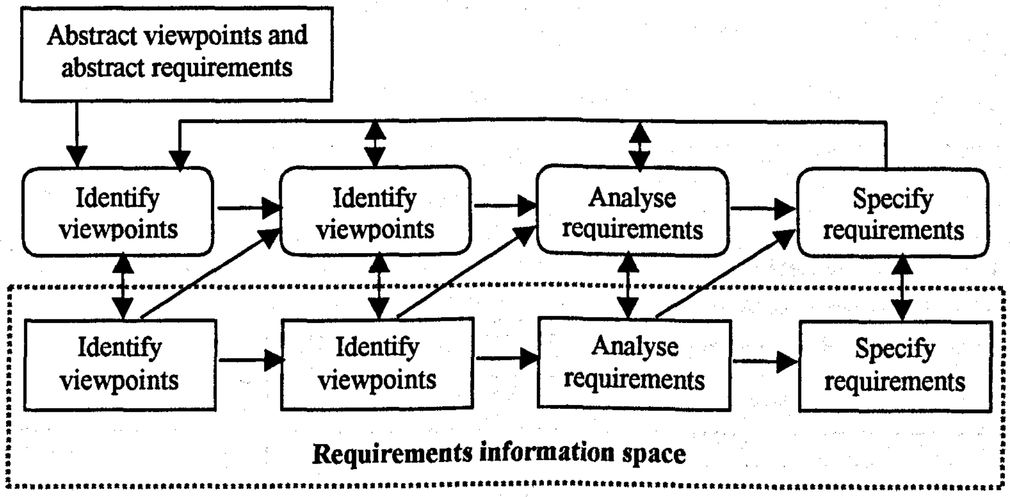


Figure 7-9 VORD process model. (from [KS98, p. 218])

VORD[5] is primarily intended for specifying an interactive system and is based on viewpoints from different perspectives. The following are its three main iterative steps:

1. viewpoint identification and structuring
2. viewpoint documentation
3. viewpoint requirements analysis and specification

Figure 7-9 shows the iterative process model of VORD. The processes are shown as round-edged boxes, and the products as square edged boxes. Each product can be viewed as the checkpoint for a review process. In the ATM example, VORD commences from the identification of abstract viewpoints by recognising what are called 'system authorities' from relevant perspectives. These abstract viewpoints can then be further

[5] The following paragraphs and some figures about VORD are mostly extracted from [KS98].

decomposed from the highest level into a number of less abstract modules in order to understand the structure and functionality of the whole software system that is to be developed. Information can be inherited by sub-class viewpoint, and so global requirements are represented in the more abstract classes and inherited by sub-classes. Figure 7-10 shows some of the ATM's viewpoints.

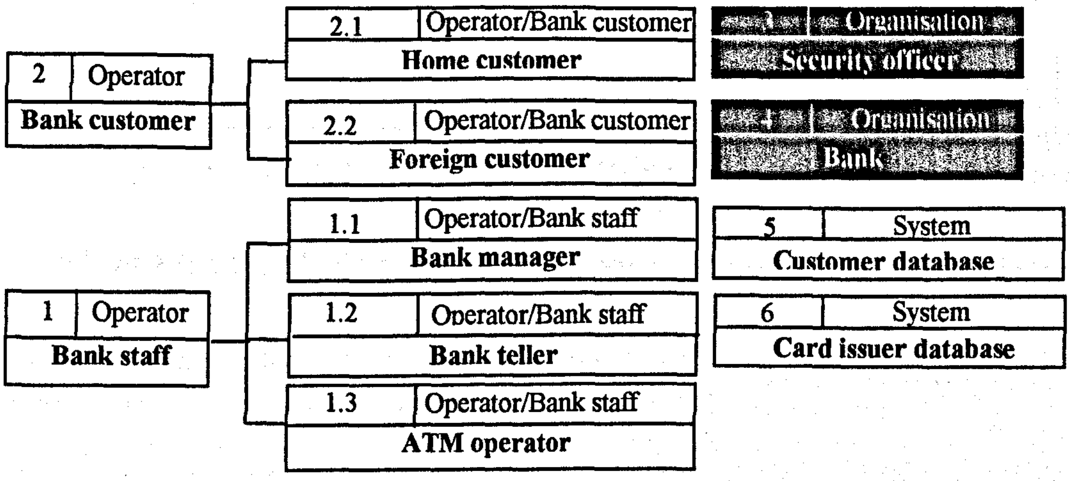


Figure 7-10. Viewpoints for an ATM system in VORD (from [KS98, p. 221])

The second step of VORD is to document the requirements of different viewpoints identified in the first step. Viewpoint requirements are made up of a set of functional, non-functional and control requirements. Control requirements describe the sequence of events involved in the interchange of information between a viewpoint and the intended system. These viewpoint requirements are documented in natural language or graphical notations. For example, Table 7-3 describes the initial requirements from the customer viewpoint. Also, Figure 7-11 illustrates an event scenario for service access. The method of using different notations to represent the same requirement in VORD is for the purpose of enhancing communication and aiding understanding between different participants.

The third step is concerned with validation by identifying errors and conflicts and resolving them. The end result is a requirements specification document.

Viewpoint		Requirements			
Identifier	Label	Description		Type	Source VP
1	Bank staff	1.1	Provide access to administrative service based on valid staff PIN and the access permission set out for the bank staff	sv	4
1.1	Bank manager	1.1.1	Provide transaction reports to bank manager	sv	1.1
		1.1.2	The bank manager requires transaction reports to be provided on a daily basis	nf	1.1
2	Bank customer	2.1	Provide access to ATM services based on valid cash-card, valid PIN and access permission set out for the bank customer	sv	4
		2.2	Provide for withdrawal of cash by bank customers	sv	4

Table 7-3. Initial requirements from some participants in VORD

VORD is a hybrid of the product- and process-oriented models discussed in the first section. It aims to specify requirements from multiple viewpoints in a distributed manner. The principle advantages offered by viewpoints [SS97] are:

- to extract more complete requirements,
- to avoid dealing with conflicts between viewpoints before they are well-informed,
- to enhance traceability.

However, VORD takes no account of context and does little to help cope with the relevant issues of changing requirements. For example, if a change (entering the cancel key) is added to Figure 7-11 as shown in Figure 7-12, other documents associated with this requirement become inconsistent with the changed documents. In addition, since VORD ends up with specification documents rather than a working model, some problems associated with system design and operation cannot be easily disclosed (a typical consequence of conventional wisdom concerning *how* and *what* in RE mentioned earlier). For example, the PIN problem discussed above could be left to designers or be neglected in VORD. When the problem is identified in the latter stage of SSD, the steps described in VORD will need to be revisited. Moreover, the understandability of these specification documents to all participants could be another problem for validation. A brief comparison of SPORE and VORD in developing requirements for the ATM system is given in Table 7-4.

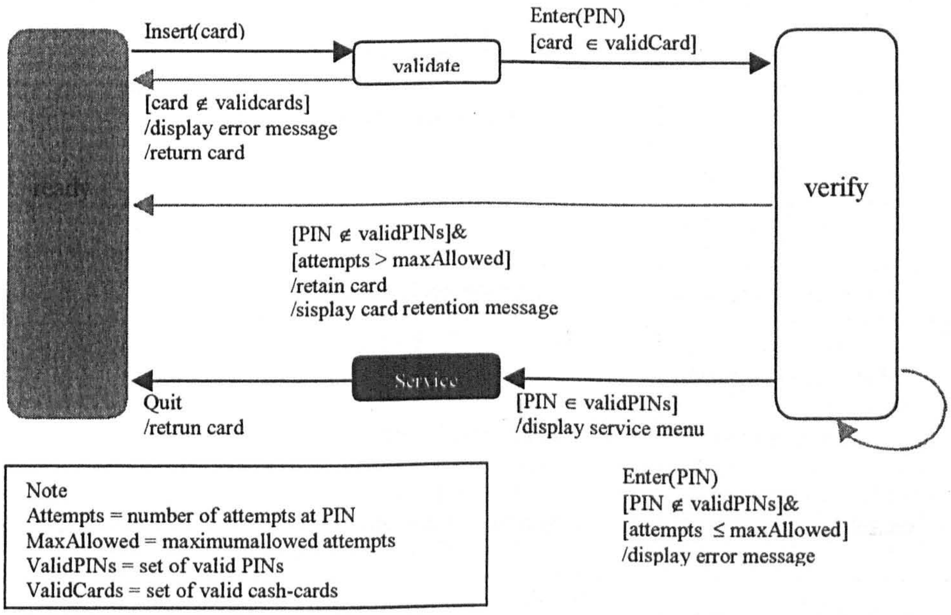


Figure 7-11. Event scenario for service access (quoted from [KS98, p.233])

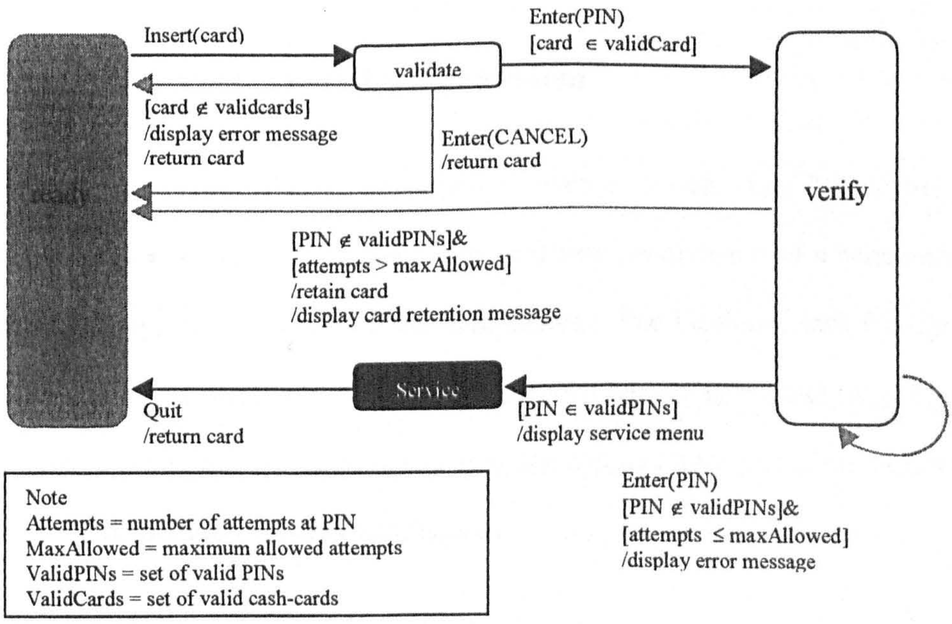


Figure 7-12. A modified event scenario for service access

	SPORE	VORD
Fundamental principle for the REP	Situatedness	Step-by-step algorithm
Main aim	Requirements development	Requirements definition
Final target	Working models	Documented specifications
Main information sources	Collaborative interaction, 'what if' experiment, and domain knowledge	Domain knowledge
Orientation	Problem-focused	Solution-driven
Participants' relationship	Collaborative	Subordinative/ Coordinative
Participation	Users and developers	Users and developers
Group work	Supported (human-centred)	Semi-supported (developers-centred)
Work style	Interactive, open-ended	Non-interactive
The relationship between the REP and design	Design has been embedded into the construction of working model	Support the transition to object-oriented design manually
The relationship between the REP and SSD	Throughout the whole life cycle of SSD	Only in the early stage
Context in the REP	Contextual dependence	Contextual independence

Table 7-4. A comparison between SPORE and VORD

7.4.2 A Warehouse Distribution System[6]

Specifying the requirements for a warehouse is taken as a case-study by Jacobson in [JCJO92]. Jacobson's concern is to identify the software requirements of a computerised system, and his approach is based on use-case analysis. For Jacobson, each use-case is associated with a particular kind of interaction between human agents and the computer system, such as might be directed towards one of the required functions of the warehouse (e.g. manual redistribution between warehouses).

[6] This case study is still proceeding. Most practical work described here has been conducted by another Ph.D. student, Y-C Chen. This subsection is closely based on Beynon's account of joint research reported in our paper: *Cultivating requirements in a situated process of requirements engineering* [SCRB99].

Within the framework of SPORE, the requirements engineering task can be seen in the broader context of developing a business process model and determining the role that computer technology can play in carrying out the characteristic transactions of the warehouse. The perspective proposed here is through-and-through agent-oriented in the sense that warehouse activity is conceived with reference to state-changing protocols for human and automated components with the system. In effect, where the action of human agents is constrained by the business process so that it follows reliable patterns, it is possible to regard their co-operative activity as a form of computation. The characteristic transactions of the warehouse are then analogous to use-cases in Jacobson's sense.

- Seed ISMs for the Warehouse State

In SPORE, the cultivation of requirements has to start from a representation of those elements of the warehouse state that are pertinent to the particular problem being addressed. This representation will take the form of a *seed* ISM that – because of the situated nature of SPORE – incorporates matter-of-fact observations of the current state of the warehouse. Typical observables that are significant in this view are the items and locations in the warehouse, and the inventory that connects items with locations. An ISM to represent these observables will supply a visual representation for items and locations, and the status of the inventory

Such a representation of the current state of the warehouse will be complemented by informal actions, for example: represent the relocation of items, look up an item in the inventory, or take receipt of a new item for storage. In some contexts, this will motivate visualisations to represent intermediate states in the operation of the warehouse that are associated with items in transit, or items located via the inventory but yet to be retrieved from the warehouse.

A model of the warehouse has to incorporate such aspects of state and state change in order to be faithful to its referent. If such aspects are neglected, there is no means to consider behaviours that, though undesirable or outside the scope of normal operation, have a profound influence on the requirement. For instance, the requirements activity has to address matters such as the loss of items or warehouse locations, the concept of items being mislaid, or the significance of perishable items.

There is no single ISM that can represent all the aspects of the warehouse state that are potentially relevant to a requirements identification. The state of the warehouse will typically be represented by different seed ISMs according to what problems are being addressed in the SPORE, and each will be introduced to mimic particular scenarios. For instance, it may be appropriate to construct seed ISMs to represent different varieties of perishable item, or to represent a very large number of items to assess the interface to an inventory database.

- The Warehouse Business Process Model (BPM)

Over and above the naive perception of states and state changes just considered, there is a business perspective on warehouse operation. This focuses on the particular agents that are intended to operate and the protocols that they follow in carrying out preconceived characteristic transactions. These define the business process model.

The observables in the BPM are different in character from items and locations. They relate to phases in preconceived transactions. The state changes are concerned with the systematic execution of protocols and the associated transition from one phase to the next. There may be no counterpart in the BPM for activities that might be possible in practice, such as the illicit retrieval of an item by its owner. An important aspect of the observables associated with the BPM is that they should not only serve to determine the current state, but must also incorporate a transaction history appropriate for auditing.

The ISM which is to be developed to represent the BPM is modelled on the practices that were used in the operation of the warehouse prior to the advent of computers. In that case, forms and paper inventories serve to record the operation of the BPM by rendering the abstract observables associated with phases and roles visible and tangible. Manual data entry, following systematic processes of form transfer, was the means to represent both the current status of all transactions (such as: which items were in transit) and the history of transactions.

To some extent, the forms and inventories can be interpreted as a paper-based ISM for the business process. In performing a particular transaction, specified procedures are to be followed in filling forms and transferring them between personnel. These manual activities effectively identify which agents have roles in the transaction, which are currently active in any phase, and how their interaction is synchronised (cf. Figures 7-13 and 7-14). The current status of any transaction is determined by what sections of forms are currently completed and who currently holds the forms.

The full details of how the BPM is construed to operate are reflected in the specific details of what each agent enters on a form. These details refer to the observational and interactive context for each agent: the observables the agent can refer to (its **oracles**), those that can conditionally change (its **handles**) and the protocol that connects these. Note that the relevant observables in this context may refer to the state of the warehouse itself (e.g. an item can be signed off only if it is presently to hand), and relate to the high-level context for interpretation (e.g. issues of legality, safety, etc.). The persistence of the record that the forms supply is also significant for auditing and traceability.

- Applications of SPORE to Warehouse Requirements

Just as paper records and protocols for interaction with them can be viewed as an ISM, so the process by which such procedures evolved can be construed as EM. The activities involved in this evolution are as described in the above discussion:

- the identification of agents: e.g. foreman, warehouse worker, driver, office clerk;
- the conception of the roles for these agents corresponding to their characteristic skills;
- the apportioning of responsibilities for particular phases within a given transaction;
- the refinement and formalisation of their precise observables and protocols.

In applying SPORE to developing warehouse requirements, this general process is emulated using computer-based technology. The ISM constructed for this purpose incorporates the seed ISMs for the warehouse; the form-based abstractions that capture the state of the BPM and the activities of the agents; and additional observations such as those associated with the wider significance of the warehouse operation (e.g. those concerned with the legality and the integrity of the business process). The transformation from a paper-based to a computer-based ISM illustrates the potential of SPORE as a framework for business-process re-engineering.

The distributed nature of dtkeden makes it possible to separate the viewpoints of the agents in the model, and to complement these with an external interpretation. In the first instance, computer-based forms are used to represent the environment for each agent's interaction. The mechanisms through which a particular kind of agent, such as a warehouse worker, interacts can be subsequently elaborated through the development of special-purpose interfaces. In this way, the distributed ISM serves as a medium in which to identify and enact appropriate transactions, and to debug and refine these through collaborative interaction between the various participants.

Examples of how requirements can be addressed by SPORE in this way include:

- Through experimentation at different workstations, it is possible to identify issues that are problematic from the perspective of particular agents: for instance, "how does the office know which drivers are available?" "How does the office determine whether a transaction is completed?"
- Through the elaboration of different seed ISMs, additional issues can be addressed, such as transportation costs, perishable goods, security and trust concerns.
- Through the modification of dependencies and communication strategies, the effects of different technologies, such as mobile communications, the Internet, optical bar code readers, or electronic locking agents, are considered,
- Through collaboration and synthesis of views, it is significant to distinguish between subjective and objective perceptions of a state e.g. to contrast "I remember doing X" with "I have some record of doing X" with "There is an official record of X", or to model misconceptions on the part of an agent.

Through intervention in the role of superagent, it is possible to examine the consequences of singular conditions that arise from opportunistic interaction or Acts-of-God, and to assess activities outside the scope of normal operation such as are associated with fraud, or manual back-up to automated procedures.

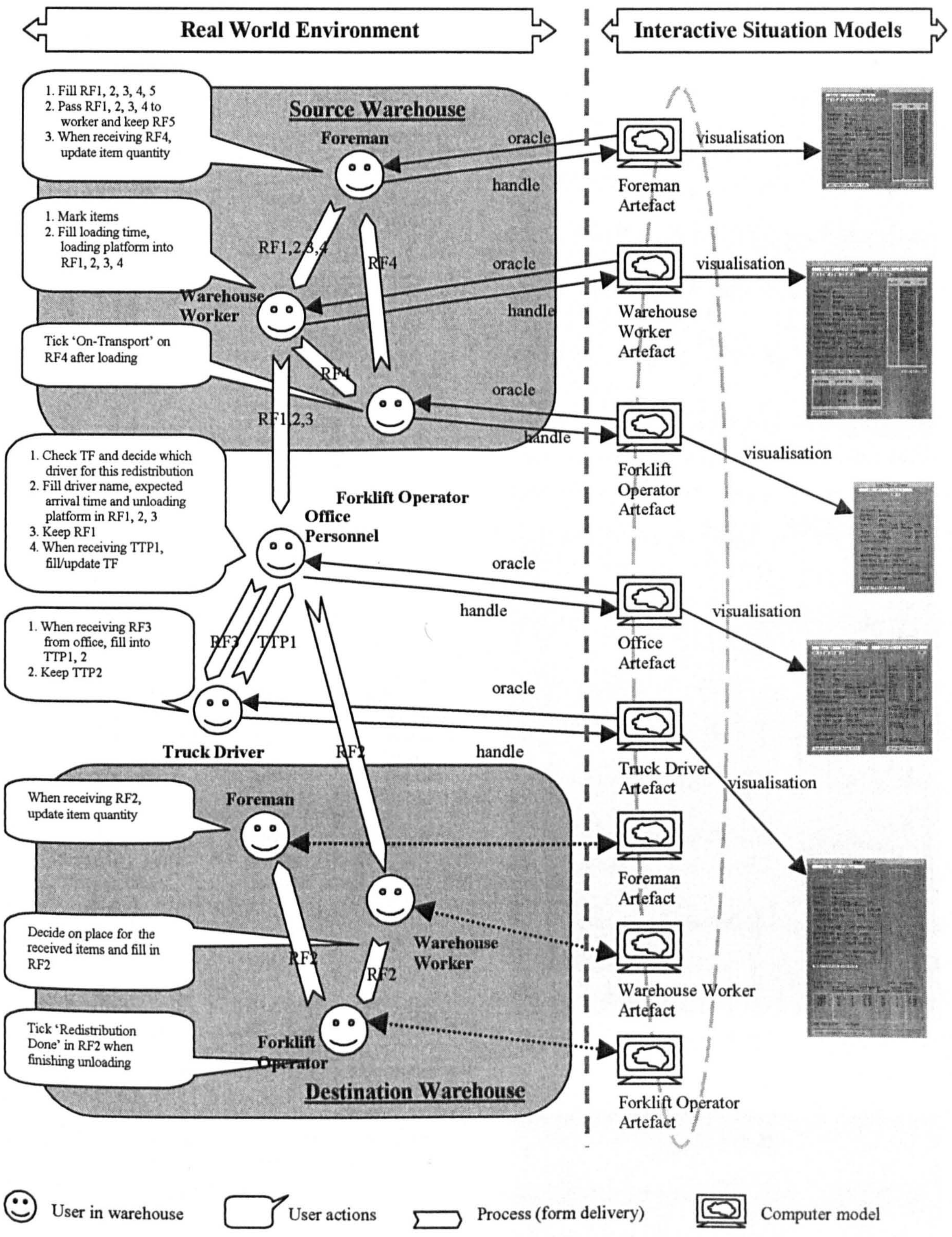
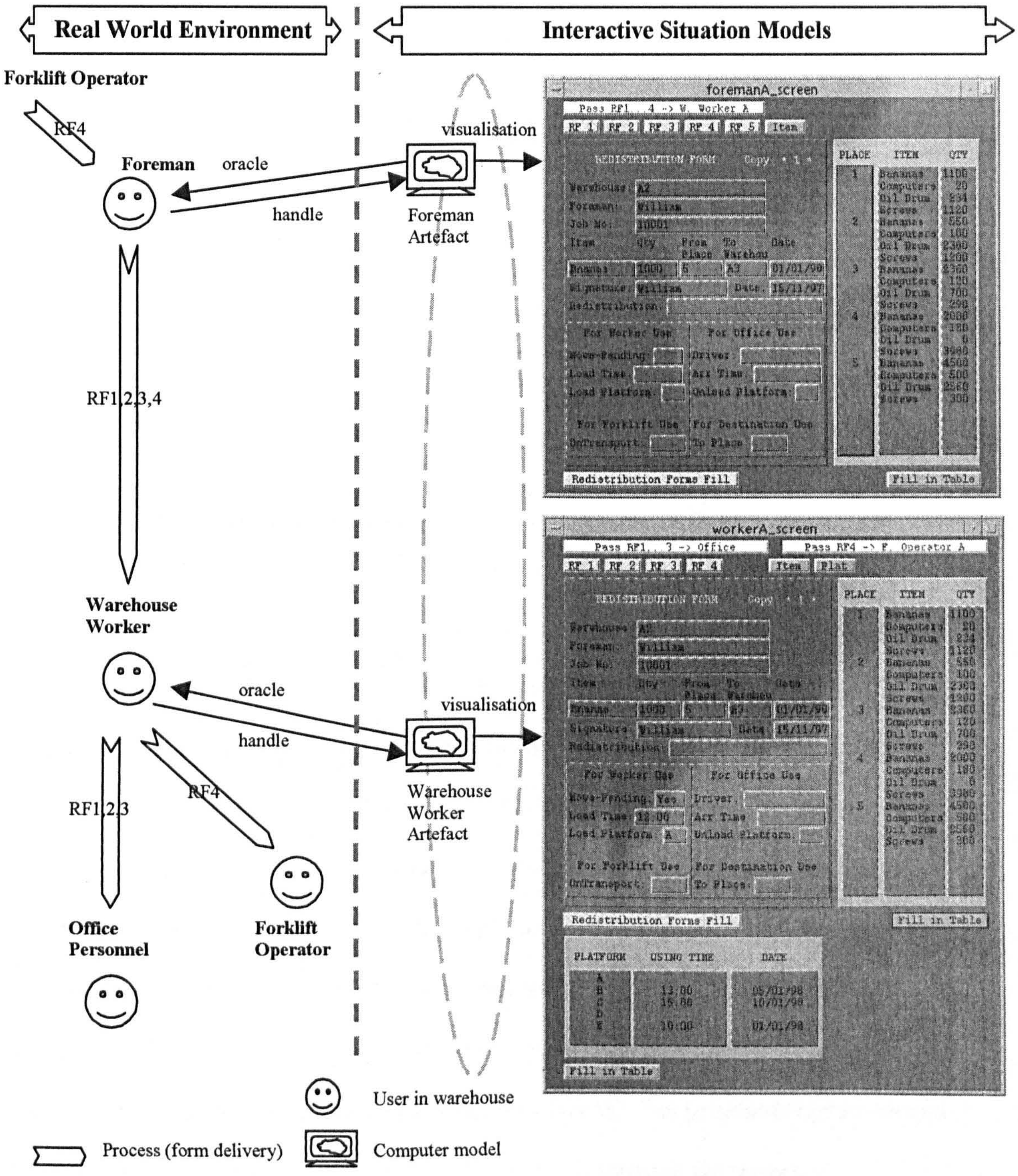

Real World Environment
Interactive Situation Models
Source Warehouse
1. Fill RF1, 2, 3, 4, 5
2. Pass RF1, 2, 3, 4 to worker and keep RF5
3. When receiving RF4, update item quantity
Foreman
oracle
handle
visualisation
Foreman Artefact
1. Mark items
2. Fill loading time, loading platform into RF1, 2, 3, 4
RF1,2,3,4
RF4
Warehouse Worker
Warehouse Worker Artefact
Tick 'On-Transport' on RF4 after loading
RF4
RF1,2,3
Forklift Operator Artefact
1. Check TF and decide which driver for this redistribution
2. Fill driver name, expected arrival time and unloading platform in RF1, 2, 3
3. Keep RF1
4. When receiving TTP1, fill/update TF
Forklift Operator
Office Personnel
Office Artefact
1. When receiving RF3 from office, fill into TTP1, 2
2. Keep TTP2
RF3
TTP1
Truck Driver
RF2
Truck Driver Artefact
When receiving RF2, update item quantity
Foreman
Foreman Artefact
Decide on place for the received items and fill in RF2
Warehouse Worker
RF2
RF2
Warehouse Worker Artefact
Tick 'Redistribution Done' in RF2 when finishing unloading
Forklift Operator
Destination Warehouse
Forklift Operator Artefact
User in warehouse
User actions
Process (form delivery)
Computer model

Figure 7-13: A collaborative working environment for manual redistribution between warehouses



foremanA_screen

Pass RF1.. 4 -> W. Worker A

RF 1 | RF 2 | RF 3 | RF 4 | RF 5 | Item

REDISTRIBUTION FORM Copy * 1 *

Warehouse: A2
Foreman: William
Job No: 10001

Item	Qty	From Place	To Warehou	Date
Bananas	1000	5	A3	01/01/98

Signature: William Date: 15/11/97
Redistribution:

For Worker Use		For Office Use	
Move-Pending:		Driver:	
Load Time:		Arr Time:	
Load Platform:		Unload Platform:	
For Forklift Use		**For Destination Use**	
OnTransport:		To Place:	

Redistribution Forms Fill

PLACE	ITEM	QTY
1	Bananas	1100
	Computers	20
	Oil Drum	234
	Screws	1120
2	Bananas	550
	Computers	100
	Oil Drum	2300
	Screws	1200
3	Bananas	2360
	Computers	120
	Oil Drum	700
	Screws	290
4	Bananas	2000
	Computers	180
	Oil Drum	0
	Screws	3980
5	Bananas	4500
	Computers	500
	Oil Drum	2560
	Screws	300

Fill in Table

workerA_screen

Pass RF1.. 3 -> Office | Pass RF4 -> F. Operator A

RF 1 | RF 2 | RF 3 | RF 4 | Item | Plat

REDISTRIBUTION FORM Copy * 1 *

Warehouse: A2
Foreman: William
Job No: 10001

Item	Qty	From Place	To Warehou	Date
Bananas	1000	5	A3	01/01/98

Signature: William Date: 15/11/97
Redistribution:

For Worker Use		For Office Use	
Move-Pending:	Yes	Driver:	
Load Time:	12:00	Arr Time:	
Load Platform:	A	Unload Platform:	
For Forklift Use		**For Destination Use**	
OnTransport:		To Place:	

Redistribution Forms Fill

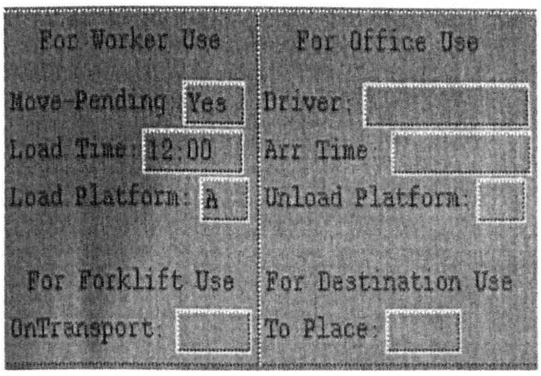
PLACE	ITEM	QTY
1	Bananas	1100
	Computers	20
	Oil Drum	234
	Screws	1120
2	Bananas	550
	Computers	100
	Oil Drum	2300
	Screws	1200
3	Bananas	2360
	Computers	120
	Oil Drum	700
	Screws	290
4	Bananas	2000
	Computers	180
	Oil Drum	0
	Screws	3980
5	Bananas	4500
	Computers	500
	Oil Drum	2560
	Screws	300

Fill in Table

PLATFORM	USING TIME	DATE
A		
B	13:00	05/01/98
C	15:00	10/01/98
D		
E	10:00	01/01/98

Fill in Table

Figure 7-14a (above): Detailed view of the forms used in the warehouse artefacts



For Worker Use		For Office Use	
Move-Pending:	Yes	Driver:	
Load Time:	12:00	Arr Time:	
Load Platform:	A	Unload Platform:	
For Forklift Use		**For Destination Use**	
OnTransport:		To Place:	

Figure 7-14b (right): Detail of panels representing observables (**handles** or **oracles**) for some warehouse agents

Chapter 8
Conclusion

8.1 Research Summary

The research presented in this thesis grew out of the author's previous experience of developing MISs (management information systems), and was initially motivated by his interest in seeking an amethodical approach for SSD. More particularly, the author was attracted by the potential of EM to serve as an open development model for developing an ill-defined, volatile software system (see Section 3.3). As explained in Chapter 1, an amethodical approach of SSD must take both technical but also social processes into account. Since technical support for social interaction in EM is not well developed, the research emphasis shifted to clarifying the distributed perspective on EM and establishing the framework for distributed Empirical Modelling (DEM). The proposed framework and the supporting tool dtkeden have fulfilled the initial objective of the research.

The thesis has examined the following fundamental issues for SSD detailed in Chapter 1: *essential character*, *real-world context*, *human factor*, *social factor* and *computer support*. The main findings of the research may be summarised as follows:

- **Essential character**

  SSD is a human activity but needs technical support in order to enhance the viability of given social goals.

Although an approach based purely on a technical perspective cannot adequately solve the social problems arising from SSD, an approach based purely on a social perspective runs the risk of withdrawing into contemplation and reflection. As recommended by J. A. Goguen in [Gog97], a clear need to reconcile the technical and social issues of information system development (i.e. SSD in the terminology of this thesis) is emerging. Even though the so-called socio-technical approaches have taken this reconciliation into account, they remain deeply rooted in an engineering discipline that can only accommodate social issues to a limited degree.

Instead of regarding SSD as a technical process with social behaviour to enhance the viability of given technical goals, this thesis regards SSD as a social process, but with technical practices to enhance the viability of given social goals. In the light of this fundamental stance, the framework for DEM is intended to address interpersonal interaction, the most primary but difficult social issue in SSD, through providing appropriate technical support (that is, the situated modelling in a collaborative work environment proposed in Chapter 4). By adopting the approach proposed in this thesis – which may be referred to as a 'techno-social' approach to denote its concern for social processes – SSD can move towards realism (from a social process perspective) and practicality (from a technical process perspective).

- **Real-world context**

  SSD is highly associated with its context, which must be considered in a situated manner.

The context of SSD is the real-world environment in which the software system is developed and operated. Obviously, it is situated (cf. J.A. Goguen's description of the qualities of situatedness given in Chapter 1) and in most cases cannot be specified in advance. Without taking context, in particular the social context, sufficiently into account, a purely technical view of SSD leads to practical difficulties (such as user resistance and managerial conflict) and has been seen as a distortion in the information technology community [AF95]. Moreover, few models take the context in the operational domain into account. The user who is using the developed system is typically not allowed to change the system in response to his/her evolving requirements. However, faced with a rapidly changing and radically competitive real world, the user often needs to adapt the system in time in order to cope with diverse situations. It is too late and expensive for the user to achieve this need simply through traditional system maintenance. The concepts of design-in-use [SH96, Fis93] and end-user computing [Nar93, DL91] have pioneered this new trend of taking the user's context in the operational domain into account in SSD.

The clear advantage of DEM is that it makes possible to take account of the context not only in the construction stage but also in the operational stage of a software system (see Sections 3.3 and 4.4). The knowledge captured by developers and users from the context in both stages can be structurally coupled with the existent knowledge embedded in computer-based models by means of definitive programming, and in turn used as the base of further interaction. In this way, the context of SSD can be considered in sufficient detail by both developers and users, as well as in a situated manner.

- **Human factor**

  Human agents are the most important dimension in SSD. The enaction of a process model should not hint at any particular roles of human agents in guiding the process of SSD.

It becomes apparent that even the use of the most advanced technologies cannot ensure the success of SSD [Gib94, RHHR98], so the influence of human beings in SSD, in particular in the success of SSD, should not be overlooked [Bro87, DS97, Leh98a]. Although human agents have negative characteristics (such as being error-prone and resistant to change) that are obstacles to the success of SSD, human agents also have positive characteristics (such as intelligence and ability to collaborate) that are very important for successful SSD and cannot be replaced by rigid algorithms and preconceived mechanisms. However, most models for SSD only focus on eliminating the negative characteristics rather than promoting the positive characteristics. They formulate the behaviour of human beings as a set of sequential activity patterns. Like the mechanism of an assembly line, these patterns are well-structured and well-defined as befits an engineering discipline. When enacting these models, human agents are thus obliged to develop software systems by following these rigid activity patterns.

Such rigidity is liable to be untenable in the real world, because the behaviour of human beings is oriented much more toward situated than postulated actions [Suc87]. There is also no reason to prohibit SSD from enjoying the positive characteristics of human agents simply because there are also negative characteristics. In fact, the so-called software crisis, which was often thought to be the result of these negative characteristics, does not result from failing to follow these well-defined activity patterns [cf. Fit96, Gib94]. Instead, it is often because these advanced technologies provide little support for making the best use of the human agents' positive characteristics. As a result, in spite of the use of this kind of model, SSD has yet to dispel the software crisis [Gib94].

Hence, the enaction of a process model should not hint at any particular roles of human agents in guiding the process of SSD. Characterised by the features of a situated activity, DEM allows the developer and the user to interact with the system that is being developed or used in an open-ended manner (See Section 2.2 and 4.2). No rigid activity patterns that limit the best use of the human agents' positive characteristics is given within DEM. The evolution of the system is utterly guided by human agents rather than by rigid algorithms and preconceived mechanisms. More significantly, by means of computer support, the human-centred process for SSD that is invoked by DEM can greatly promote the human agents' positive characteristics with the least risk of resurrecting the problems of an *ad hoc* approach, and at the same time help to reduce the influence of their negative characteristics.

- **Social factor**

  Interpersonal interaction is the most critical activity in SSD. A model for SSD should be able to provide effective support for interpersonal interaction to a large extent.

Interpersonal interaction that integrates both cognitive and social processes is the most critical activity in SSD [Pot93, Bro87, RHHR98]. Because of their failure to support effective interpersonal interaction, many projects have been cancelled or abandoned [ITC98, STM95, VPC98]. By drawing on an engineering discipline, traditional models for SSD pay little attention to this crucial activity in SSD. It is presumed that interpersonal interaction can be attained through well-defined, well-structured representational media. However, in practice, obstacles of interpersonal interaction arise very often and pose the main obstacles to the success of SSD [Bos89, Sal87, STM95, VF87, VPC98].

Within the framework for DEM, interpersonal interaction between modellers for shaping the agency of agents in SSD is supported in two levels. Being external observers, modellers can shape the agency in the context of their task roles (such as developers and users (cf. [Son93])) in a concurrent environment. On the other hand, being internal observers, modellers can act as agents to carry out the interaction between agents through pretend play as proposed in this thesis. This being-participant-observer approach serves to shape the agency of agents within the system in their (agents') customary context rather than the modellers' context (see Sections 4.2 and 4.3). By using networked computer-based models to support modelling in these two levels, DEM can provide modellers with computer-mediated interaction that is complementary to traditional interpersonal interaction. In this way, interpersonal interaction for SSD can be supported to an even larger extent.

- **Computer support**

  Computer-based support plays an enabling role in facilitating human agency and promoting interpersonal interaction. With this support in a distributed environment, human agents can explore, expand, experience and communicate shared knowledge in an open-ended, interactive, and situated fashion.

It is evident that hitherto the computer has been widely used to improve human interaction. However, the typical use of a computer as an application tool, in particular for knowledge representation, restricts its advantages. For example, most CASE (computer-aided software engineering) tools use the computer for documentation, automation and code generation. In this sense, the computer is at best simply a powerful word processor that helps its users to organise documents in certain forms and to translate documents from one form (text or diagram) to another form (program code). However, the computer can be best used as an open-ended artefact for facilitating knowledge construction by situated modelling (see Section 3.2) [Cro94, FP88]. In addition, with the aid of network communication, the computer has powerful potential to enhance interpersonal interaction (see Section 4.2) in ways that serve 'to be equipment for language' as highlighted in [WF86, p.79].

The enaction of DEM with computer support is a situated activity. Within DEM, modellers need to create computer-based models to support their situated modelling for SSD (see Chapter 2 and 4). This enables each modeller to explore, expand and experience individual knowledge by interacting with his/her own computer-based model. At the same time, modellers can also interact with each other through their networked computer-based models for exploring, expanding, experiencing and communicating shared knowledge. In this way, human agency and interpersonal interaction can be effectively enhanced and improved in an open-ended, interactive and situated fashion.

The above preliminary findings supply a promising basis for the application of DEM to SSD. A radically new objective in this application is to encourage the use of an open development model (ODM) for a software system throughout its development and use (see Section 3.3 and 4.3). With the openness and situatedness of an ODM, not only the developer but also the user can guide the evolution of the system in response to their rapidly changing needs. In this way, SSD becomes a human-centred activity with autonomy rather than a preconceived mechanism followed by human agents (cf. [Flo95, Flo87]). Moreover, in order to avoid taking the risk of resurrecting the problem of an *ad hoc* approach, it is definitely necessary to provide the computer support for exploiting the positive characteristics and neutralising the negative characteristics of human agents. In this manner, the formal technical issues and informal social issues of SSD can be to a large extent reconciled (cf. [Gog97, Gog96]).

Recognising the importance of the above fundamental issues for SSD in the real world, DEM as an ODM regards a software system as a computer-based model, and enables the developer and the user to guide the evolution of the system with computer support for modelling (see Chapter 2-5). Although the justification of applying DEM to SSD is not clearly given in this thesis, several case studies, such as a hotel booking system (see Section 2.4), railway accident animation and other examples (see Chapter 6 and Section 7.4), have illustrated promising potential for this application. Limitations of research time have prevented the author from investigating many issues in SSD, but the thesis has addressed one of the most difficult parts of SSD (that is, requirements development (see Chapter 7)) and includes practical case studies for SSD (see Chapter 6). Further work on applying DEM to other issues of SSD is on-going in the research group of Empirical Modelling (in Warwick University).

Requirements development is a labour-intensive task and is intertwined with SSD in a symbiotic manner. Its process is largely driven by human interaction. However, like the process of SSD, this process has often been examined from the perspective of an engineering discipline. It is not surprising that the technically-oriented models proposed to prescribe this human-centred interactive process face almost the same problems that arise in many software process models based on step-by-step algorithms. Following the view of SSD as a social process of human interaction, this thesis reengineers the process of requirements development in terms of problem-solving so as to highlight its situated, context-dependent character. The proposed SPORE framework describes this problem-solving process, in which human beings, on the basis of their current context and resources, interact with each other in order to solve problems as they are identified. DEM is applied to SPORE in order to create a collaborative work environment for participants taking part in the process of requirements development. Within this environment, individual insights into requirements and participants' shared understanding of requirements become visible and communicable through the use of networked computer-based models. In this way, the unsatisfactory nature of communication by traditional methods, such as documentation and conversation, can easily be improved, and as a result the communication obstacles within, among and between participants identified in [VF87] can be reduced.

Moreover, the case studies included in the present research (see Chapter 6) show that the tool dtkeden does exhibit DEM and provides a practical exercise environment for *knowing* DEM, in particular for novices. Since this tool is simply an academic product and is not technically elegant, it is inevitable that there will further improvements in its functionality (Section 8.3.3 describes some particular improvements). However, despite the need to improve its functionality for realistic practical use, dtkeden is already a system that effectively supports situated modelling in a collaborative working environment.

8.2 Research Limitations

Undoubtedly, there are several research limitations which need to be acknowledged. In part, these result from the deliberately restricted scope of the research. In part, they are due to the inevitable restrictions of time and resources.

- This research, that aims to establish DEM for supporting the social process of SSD, does not provide ways of addressing most of the social issues associated with this process. Some account has been taken of human interaction and learning, but this thesis does not deal with other important social issues, such as job satisfaction, user resistance and worker democracy. This is because they can only be adequately handled by non-technical methods which are beyond the scope of this thesis. However, the open-ended distributed modelling framework (DEM) and environment (dtkeden) do provide, to some extent, the technical support for such non-technical methods. For example, in order to improve users' job satisfaction, non-technical methods such as negotiation and brainstorming can be supported by 'what if' experiments in the proposed DEM framework and environment.

- Human interaction in the framework proposed for DEM is simplified to four primary modes when it is implemented in dtkeden. Obviously, human interaction is much too complicated to be reduced to these four modes. As D. Sonnenwald has shown in her analysis of both communication roles and task roles [Son93, Son96], the complexity of a participant's communication network is far beyond what has been presented in dtkeden. A possible extension of the existing structure is suggested in Section 8.3.2.
- From a technical viewpoint, it is assumed that dependencies between observables can be formalised by using definitive notations and represented by a computer-based model. However, many dependencies in the real world are very difficult to describe in this format. Even in this context, DEM can allow modellers to undertake social interaction with each other to advance the process of SSD without imposing rigid activity patterns.
- This research has not involved a detailed comparison of DEM with other methods. Each method has its advantages and disadvantages, and the choice of a method must depend on the project itself [AF95]. The main objective of this research, that is, clarifying and enhancing the distributed perspective on EM for supporting a social process of SSD, has been served by developing the proposed framework for DEM. However, further comparison of using DEM and other methods for SSD will help to classify the situations and domains in which DEM can effectively be applied to develop software systems.

8.3 Further Work

This thesis has proposed a framework for DEM to enhance the interaction between a group of people, and a tool to support this framework. The framework has been conveniently applied to requirements engineering in order to facilitate the cultivation of requirements in terms of a shared understanding between participants. The tool dtkeden has also been successfully used to develop several case studies. There remains the potential to

- apply DEM to other issues of SSD (cf. [BCRS98, BCSW99]);
- apply DEM to new subjects which rely heavily on effective interpersonal interaction;
- develop new case-studies and applications with the tool dtkeden;
- improve the functionality of dtkeden;
- evaluate the impact of computer-mediated interpersonal interaction

In the following subsections some suggestions are given for further work based on the research proposed in this thesis.

8.3.1 Possible Applications of DEM

The framework for DEM proposed in this thesis promises to provide a distributed environment of human interaction for facilitating mutual knowledge exploration, extension and communication through networked computer-based models. A main application of DEM to SSD has been achieved in requirements development (see Chapter 7). Other applications that have been initiated include human-computer interaction

[BRSW98] and program comprehension [BS98]. In addition to applying DEM to SSD, this framework is potentially applicable to the following research topics:

- Computer-Supported Collaborative Work (CSCW)

CSCW is a new field concerned with the research and development of software systems to support 'group working'. It usually takes face-to-face communication between developers as its natural form [LG97]. In CSCW applications, a common criticism is that there is a lack of user involvement, which causes severe problems [Kyn91, GKM93]. In CSCW, user involvement requires techniques that enable users to understand the possibilities for computer support and to envision work with a proposed system. Traditional requirements specification is not suited for this purpose, since most users are unable to bridge the gap between description and their professional knowledge and skills. In this case, by using the framework proposed for DEM and the tool dtkeden, further work situations for users can be envisioned through computer-based models, and thus users are allowed to gain hands-on experience.

- Group Decision Support System (GDSS)

Group discussion has become a very prevailing trend for making decisions in an organisation. It can facilitate not only the establishment of group consensus but can also inspire new thinking about innovation, thus enhancing the quality of a decision. Traditional decision support systems are intended to support a decision-maker by providing several alternatives. These alternatives are usually generated by decision models that are built in advance. However, decision-making in an organisation is deeply influenced by its context, in which a lot of people are involved. Accordingly, human interaction may become very critical for making decisions in ways that are more democratic, effective and creative but involve less conflict [WDP88]. The framework for DEM is able to provide such a social forum for GDSS in which participants (who are usually representative of a small group of people) are allowed to interact with each other in their specific contexts. Through the networked computer-based models, sensitivity analysis distributed across different people can easily be achieved. As a result, the influence of each alternative can be clarified, and more significantly, new alternatives can be created to help an organisation survive in today's increasingly competitive business world.

8.3.2 An Further Extension to DEM

The framework proposed for DEM presents a form of human interaction in an organisation. Due to the limits of research time, the interaction between modellers is driven only by their task. Therefore, in the context of modellers, the interaction presented in the framework for DEM is concerned with each modeller's role in performing a particular task, such as the role of a developer, a designer or a user. In the same way, in the context of the agents whom modellers pretend to play, only the task-oriented roles are taken into account, such as those of a train driver, a signalman, a button, or a signal. The interaction between participants arising from their communication (or interaction, in the author's terminology) roles proposed in [Son93] is simplified in this thesis; that is, only their main (or single) role is considered. Figure 5-7 illustrates this simplified idea.

Conceptually, the extension of Figure 5-7 by introducing role-oriented communication can be readily achieved, even though its structure may become extraordinarily complex. When communication roles can be classified and assigned to participants, participants can be divided into several small-scale communication groups (which are not the same as the task-oriented groups shown in Figure 5-7). According to his/her communication roles, each participant can belong to one or more groups and may change his/her roles in a group. Obviously a multi-layered architecture for supporting the complex interaction between participants can be anticipated.

However, the technical support for this architecture needs further research. First, the multi-layered communication environment has to be established. This complex environment may need a more sophisticated configuration for network communication than the star type proposed in Section 5-1. Then, role change in a group must be considered. Security control and richer interaction modes may also require more attention. In addition, if a participant may interact with more than one group at the same time, consideration must also be given to a multi-windows interaction model for the participant and to concurrent interaction across different network configurations.

8.3.3 The Improvement of dtkeden

As indicated in Chapter 3, dtkeden (as well as tkeden) still has some technical limitations which need to be overcome. For example, powerful data manipulation tools and friendly user-interface design tools would be very useful improvements. The former may be obtained by connecting dtkeden with a commercial database tool, and the latter then could be achieved by extending Scout and Donald to become icon/window-based event-driven tools. By characterising Scout with event-driven features, the author has tried to implement some VB-like event-driven functions, such as *click*, *change*, *setText* and *getText*, into Scout windows. The results are promising. These built-in functions allow a modeller to take less account of the internal mechanism of the computer and, more significantly, to define the dependency between a component's state and a built-in action or procedure. For example, by using a *change* event, a modeller can easily define the dependency between the change to the content in a textbox and a particular action or procedure, for example, data checking. This is very helpful for user interface design. Unfortunately, due to limited research time, this extension has not yet been completed.

In addition, an ambitious objective is to rewrite dtkeden in other advanced programming languages and techniques. In fact, attempts to do this are already under way

[Car98]. It is to be expected that these new tools, which are still being researched, may have the potential to provide yet more support for EM and DEM.

8.3.4 Evaluation of Computer-mediated Interpersonal Interaction

Although the author believes that the interpersonal interaction can be significantly improved through the additional use of computer-based models that complement traditional face-to-face interaction, further evaluation would be helpful. This evaluation might be carried out by studying the performance of two groups of people who respectively use dtkeden and tkeden to develop a complicated software system. The interpersonal interaction in the latter case is achieved through traditional face-to-face communication, but it is accomplished in the former case by using both computer-mediated interaction and traditional communication.

Bibliography

[AA87] P.A. Adler and P. Adler. Membership roles in field research. SAGE Publications, 1987.

[ABCY94] V. D. Adzhiev, W. M. Beynon, A. J. Cartwright, and Y. P. Yung. A computational model for multi-agent interaction in concurrent engineering. In *Proc. CEEDA '94*, pp.227-232, Bournemouth University, 1994.

[ABH86] D. Angier, T. Bissell, and S. Hunt. DoNaLD: a line drawing notation based on definitive principles. Research report RR-86, Department of Computer Science, University of Warwick, 1986.

[AC98] C. Avgerou and A. Cornford. *Developing information systems: concepts, issues and practice*. 2nd edition, Macmillan Press, 1998.

[AF95] D. E. Avison and G. Fitzgerald. *Information system development: methodologies, techniques and tools*. 2nd edition, McGraw-Hill, 1995.

[Agr95] P. E. Agre. Computational research on interaction and agency. In P.E. Agre and S.J. Rosenschein (eds.), *Computational theories of interaction and agency*, pp.1-52, The MIT Press, 1995. Also in *Artificial Intelligence*, 72(1): 1-52, 1995.

[AL89] P. M. Allen and M. J. Lesser. Evolution: travelling in an imaginary landscape. In J. D. Becker et al (eds.), *Workshop on Evolutionary Modells and Strategies*, pp.420-441, Neubiberg, Germany, March 1989.

[Alt96] S. Alter. *Information systems: a management perspective*. 2nd edition, The Benjamin/Cummings Publishing Company, 1996.

[And+90] N. Andersen, F. Kensing, J. Lundin, L. Mathiassen, A. Munk-Madsen, M. Rasbech, and P. Sorgaard. *Professional systems development: experience, ideas and action*. Prentice-Hill, 1990.

[And94] S. F. Andriole. Fast, cheap requirements: prototype, or else. *IEEE Software*, 11(2):85-87, 1994.

[Awa88] E. M. Awad. *Management information systems: concepts, structure and applications*. The Benjamin/Cummings Publishing Company, 1988.

[Aye95] S. Ayer. *Object-oriented client/server application development*. Mcgraw-Hill, 1995.

[Azm88] M. Azmoodeh. *Abstract data types and algorithms*. MacMillan Education Ltd, 1988.

[Bac93] J. Bacon. *Concurrent systems: an integrated approach to operating systems, database, and distributed systems*. Addison-Wesley, 1993.

[BBY92] W. M. Beynon, I. Bridge, and Y.P. Yung. Agent-oriented modelling for a vehicle cruise control system. In *Proc. ESDA '92*, pp.159-165, 1992.

[BCDS93] A. J. C. Blyth, J. Chudge, J. E. Dobson, and M. R. Strens. ORDIT: a new methodology to assist in the process of eliciting and modelling organisation requirements. Technical report No.456, Department of Computer Science, University of Newcastle upon Tyne, 1993.

[BCRS98] W. M. Beynon, R. I. Cartwright, J. Rungrattanaubol, and P. H. Sun. Interactive situation model for system development. Research report RR-353, Department of Computer Science, University of Warwick, 1998

[BCSW99] W. M. Beynon, R. Cartwright, P. H. Sun, and A. Ward. Interactive situation models for information system development. In *Proc. of 5th Inter. Conf. on Information Systems, Analysis and Synthesis*, pp. 9-16, 1999.

[BD93] C. Britton and J. Doake. *Software system development: a gentle introduction*. Mcgraw-Hill, 1993.

[BE94] T. Bryant and A. Evans. OO oversold: those objects of obscure desire. *Information and Software Technology*, 36(1):35-42, 1994.

[Ber94] J. Bergin. *Data abstraction: the object-oriented approach using c++*. McGraw-Hill, 1994.

[Bey+89] W. M. Beynon, M. T. Norris, S. B. Russ, M. D. Slade, Y. P. Yung, and Y. W. Yung. Software construction using definitions: an illustrative example. Research report RR-147, Department of Computer Science, University of Warwick, 1989.

[Bey86] W. M. Beynon. The LSD notation for communication systems. Research report RR-87, Department of Computer Sciences, University of Warwick, 1986.

[Bey94] W. M. Beynon. Agent-oriented modelling and the explanation of behaviour. In *Proc. Inter. workshop on shape modelling parallelism, interactivity and applications*, pp.54-63, University of Aizu, Japan, 1994.

[Bey97] W. M. Beynon. Empirical Modelling for educational technology. In P*roc. of Cognitive Technology* 1997, pp.54-68, university of Aizu, Japan, 1997.

[Bey98] W. M. Beynon. Empirical Modelling and the foundation of artificial intelligence. In C. Nehaniv (ed), *Pro. Inter. Workshop on Computation, Metaphor, Analogy and Agents*, University of Aizu, Japan, April 1998.

[BeyMsc] W. M. Beynon. Lecture notes on the MSc module "Empirical Modelling for Concurrent Systems", Department of Computer Science, University of Warwick, 1997.

[BG96] R. Buck-Emden and J. Galimow. *SAPR/3 system: a client/server technology.* Addison-Wesley, 1996.

[BL98] D. M. Berry and B. Lawrence. Requirements engineering. *IEEE Software* 15(2):26-29, 1998.

[Blu93] B. I. Blum. Representing open requirements with a fragment-based specification. *IEEE trans. on Systems, Man, and Cybernetics*, 23(3):724-736, 1994.

[Blu94a] B. I. Blum. Characterizing the software process. *Information and decision technologies*, 19:215-232, 1994.

[Blu94b] B. I. Blum. A taxonomy of software development methods. *Communications of the ACM*, 37(11):82-94, 1994.

[BNOS90] W. M. Beynon, M. T. Norris, R. A. Orr and M.D. Slade. Definitive specification of concurrent systems. In *Proc. UKIT1990, IEE Conf. Publications 316*, pp.52-57, 1990.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61-72, 1988.

[Boo94] G. Booch. *Object-oriented analysis and design with application.* Benjamic/Cummings, 1994.

[Bos89] R. P. Bostrom. Successful application of communication techniques to improve the systems development process. *Information & management*, 16:279-295, 1989.

[BR94] W. M. Beynon and S. Russ. Empirical Modelling of requirements. Research report RR-277, Department of Computer Science, University of Warwick, 1994.

[Bro87] F. P. Brooks. Jr. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10-19, 1987.

[Brö95] P. Brödner. The two cultures in engineering. In B. Göranzon (ed.), *Skill, technology and enlightenment: on practical philosophy*, pp.249-260, Springer-Verlag, 1995.

[BRSW98] W. M. Beynon, J. Rungrattanaubol, P. H. Sun, and A. E. M. Wright. Explanatory models for open-ended human computer interaction. Research report RR-346, Department of Computer Science, University of Warwick, 1998.

[BS98] W. M. Beynon and P. H. Sun. Interactive situation model for program comprehension. Research report RR-352, Department of Computer Science, University of Warwick, 1998.

[BS99] W. M. Beynon and P. H. Sun. Computer-mediated communication: a distributed Empirical Modelling perspective. In *Proc. of the 3rd Inter. Conf. of Cognition Technology (CT'99)*, pp.115-132, 1999.

[BT94] R. I. Brafman and M. Tennenholtz. Ascribing beliefs. Technical Report CS-TN-94-6, Department of Computer Science, Stanford University, 1994.

[Bub95] J. A. Bubenko Jr. Challenges in requirements engineering. In *Proc. of the 2nd Inter. Sym. on Requirements Engineering*, pp.160-162, 1995.

[Bur91] P. Burnard. *Experiential learning in action.* Academic Publishing Group, 1991.

[Car98] R. I. Cartwright. *Geometric aspects of empirical modelling: issues in design and implementation.* PhD thesis, Department of Computer Science, University of Warwick, September 1998.

[CCH80] G. A. Champine, R. D. Coop, and R. C. Heinselman. *Distributed computer systems: impact on management, design and analysis*. North-Holland Publishing, 1980.

[CDK94] G. Coulouris, J. Dollimore, and I. Kindberg. *Distributed system: concepts and design*. Addison-Wesley, 1994.

[CGC96] M. Christensen, N. Grumman, and C. Chang. Blueprint for the ideal requirements engineer. *IEEE software*, 13(2):12, March 1996.

[Che76] P. Chen. The entity relationship model – toward a unified view of data. *ACM Trans. on Data Base Systems*, 1(1):6-36, 1976.

[Cla97] W. J. Clancey. *Situated cognition: on human knowledge and computer representations*. Cambridge University Press, 1997.

[Clar90] A. Clark. Connectionism, competence and explanation. In M. A. Boden (ed.), *The philosophy of artificial intelligence*, pp.281-308, Oxford University Press, 1990.

[Cle86] J. C. Cleaveland. *An introduction to data types*. AT&T Bell laboratory, 1986.

[Cou95] A. Coulon. *Ethnomethodology*, translated from French by J. Coulon and J. Katz, SAGE Publications, 1995.

[Cro94] C. Crook. *Computers and the collaborative experience of learning*. Routledge, London, 1994.

[CS90] P. Checkland and J. Scholes. *Soft system methodology in action*. John Wiley and Sons, 1990.

[CWG93] E. Carmel, R. D. Whitaker, and J. F. George. PD and joint application design: a transatlantic comparison. *Communications of the ACM*, 36(4):40-48, June 1993.

[CY90] B. J. Cox and E. Yourdon. *Object-oriented analysis*. Prentice-Hall, 1990.

[Dav93] A. Davis. *Software requirements: objects, functions and states*. Prentice-Hill, 1993.

[DBP93a] E. Dubois, P.D. Bois, and M. Petit. O-O requirements analysis: an agent perspective. In O. M. Nierstrasz (ed.), 7th *European Conf. in Object-oriented Programming, ECOOP'93*, pp.458-481, Germany, 1993.

[DBP93b] E. Dubois, P. D. Bois, and M. Petit. Elicitating and formalising requirements for C.I.M. information systems. In C. Rolland et al (eds.), 5rd *Inter. Conf. on Advanced Information System Engineering CAiSE '93*, pp.252-274, Paris, France, 1993,.

[DDH72] O. Dahl, E. Dijkstra and C.A.R. Hoare. *Structured programming*, , Academic Press, London, 1972.

[Den87] D. C. Dennett. *The intentional stance*. The MIT Press, 1987.

[Dep92] Department of Computer Science, University of Warwick. *Technical Document for the Scout system*. October 1992.

[DF98] W. Dzida and R. Freitag. Making use of scenarios for validating analysis and design. *IEEE trans. on Software Engineering*, 24(12):1182-1196, 1998.

[diS88] A. diSessa. Knowledge in pieces. In G. Forman and P. Pufall (eds.), *Constructivism in the computer age*. Lawrence Erlbaum Associates, pp. 49-70, 1988.

[DL91] L. Davies and P. Ledington. *Information in action: soft system methodology*. MacMillan Education Ltd, 1991.

[Dor98] H. P. D'ornellas. *Agent-oriented modelling for collaborative group learning*. Master thesis, Department of Computer Science, University of Warwick, 1998.

[Dow87] M. Dowson. Iteration in the software process. In *Proc. 9th Int. Conf. Software Engineering*, pp.36-39, San Francisco, Ca, 1987.

[Dre79] H. L. Dreyfus. From micro-worlds to knowledge representation: AI at an impasse. In J. Haugeland (ed.), *Mind design II: philosophy, psychology, artificial intelligence*. The MIT Press, 1997.

[DS94] A. M. Davis and P. Sitaram. A concurrent process model of software development. *ACM Software Engineering Notes*, 19(2):38-51, 1994.

[DS97] S. E. Donaldson and S. G. Siegel. *Cultivating successful software development: a practioner's view*. Prentice Hall , 1997.

[DW96] N. Dale and H. M.Walker. *Abstract data type: specification, implementation and application*. D.C. Heath and Company, 1996.

[Eas93] S. Easterbrook. Domain modelling with hierarchies of alternative viewpoints. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, pp.65-72, 1993.

[Edw87] D. Edwards and N. Mercer. *Common knowledge*. Methuen, London, 1987.

[EM95] K. E. Eman and N. H. Madhavji. A field study of requirements engineering practice in information systems development. In *Proc. of the 2nd Inter. Sym. on Requirements Engineering*, pp.68-80, 1995.

[EQM96] K. E. Emam, S. Quintin, and N. H. Madhavji. User participation in the requirements engineering process: an empirical study. *Requirements engineering*, 1(1):4-26, 1996.

[Fai96] J. Fairclough. *Software engineering guides*. Prentice-Hall, 1996.

[FC96] M. Fayad and M. P. Cline. Aspects of software adaptability. *Communications of the ACM*, 39(10):58-59, 1996.

[Fey75] P. Feyerabend. *Against method.* NLB, 1975.

[FHW94] B. Fields, M. Harrison, and P. Wright. From informal requirements to agent-based specification. *SIGCHI Bulletin* , 26(2), April 1994.

[Fin94] A. Finkelstein. Requirements Engineering: a review and research agenda. In *Proc. 1st Asian & Pacific Software Engineering Conf.*, pp.10-19, IEEE Press, 1994.

[Fis91] G. Fischer. The importance of models in making complex systems comprehensible. In M.J. Tauber and D. Ackermann (eds.), *Mental models and human-computer interaction 2*, pp.3-36, North-Holland, 1991.

[Fis93] G. Fischer. Shared knowledge in cooperative problem-solving Systems – integrating adaptive and adaptable components. In M. Schneider-Hufschmidt et all (eds), *Adaptive User Interfaces: Principles and Practice*, pp.49-68, North-Holland, 1993.

[Fit96] B. Fitzgerald. Formalised system development methodologies: a critical perspective. *Information System Journal*, 6:3-23, 1996.

[Flo87] C. Floyd. Outline of a paradigm change in software engineering. In G. Bjerkness, P. Ehn, and M. Kyng (eds), *Computer and democracy: a Scandinavian challenge*, pp.191-212, Avebury, Aldershot, 1987.

[Flo95] C. Floyd. Theory and practice of software development. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach (eds.), *TAPSOFT'95: theory and practice of software development, Lecture Notes in Computer Science*, 915:25-41, 1995.

[FP88] G. Forman and P. Pufall. *Constructivism in the Computer Age.* Lawrence Erlbaum Associates, 1988.

[FS96] A. Finkelstein and I. Sommerville. The viewpoints FAQ. *Software Engineering Journal,* 11(1):2-4, 1996.

[FSCSF88] G. G. Fein, E.K. Scholnick, P.F. Campbell, S.S. Schwartz, and R. Frank. Computing space: a conceptual and developmental analysis of LOGO. In G. Forman and P. Pufall (eds.), *Constructivism in the Computer Age.* Lawrence Erlbaum Associates, 1988.

[Gar67] H. Garfinkel. *Studies in ethnomethodology.* Prentice-Hall, 1967.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[Gib94] W. W. Gibbs. Software's chronic crisis. *Scientific American,* pp.72-84, September 1994.

[GK94] J. Galegher and R. E. Kraut. Computer-mediated communication for intellectual teamwork: an experiment in group writing. *Information Systems Research,* 5(2):110-138, 1994.

[GKM93] K. Grønbaek, M. Kyng, and P. Mogensen. CSCW challenges: cooperative design in engineering projects. *Communications of the ACM,* 36(4):67-77, June 1993.

[Gog94] J. A. Goguen. Requirements engineering as the reconciliation of technical and social Issues. In M. Jirotka and J. Goguen (eds), *Requirements Engineering: Social and Technical Issues*, pp.165-200, Academic, 1994.

[Gog96] J. A. Goguen. Formality and informality in requirements engineering. In *Proc. 2nd Inter. Conf. on Requirements Engineering*, pp.102-108, 1996.

[Gog97] J. A. Goguen. Toward a social, ethical theory of information. In G. C. Bowker et al (eds.), *Social science, technical systems and cooperative work: beyond the great divide*, pp.27-56, Lawrence Erlbaum Associates, 1997.

[Goo90] D. Gooding. *Experiment and the making of meaning*. Kluwer Academic Publishers, 1990.

[GS70] H. Garfinkle and H. Sacks. On formal structures of practical action. In J. C. Mckinney & E. A. Tiryakian (eds.), *Theoretical sociology*, pp.338-366, Appleton-Century-Crofts, 1970.

[GYCBC96] D. K. Gehring, Y. P. Yung, R. I. Cartwright, W. M. Beynon, and A. J. Cartwright. Higher-order constructs for interactive graphics. In *Proc. Eurographics UK Chapter, 14th Annual Conf.*, pp.179-192, 1996.

[Hal89] M. I. Hale. *The mind: Its origin, evolution, structure and functioning*. Hale-van Ruth, Pittsburgh, 1989.

[Haml78] D. W. Hamlyn. *Experience and the growth of understanding*. Routledge & Kegan Paul, 1978.

[Hau97] J. Haugeland. What is mind design. In J. Haugeland (ed.), *Mind design II: philosophy, psychology, artificial intelligence*. The MIT press, 1997.

[HED93] S. D. P. Harker, K. D. Eason, and J. E. Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, pp.266-270, 1993.

[Hen96] H. Hendriks-Jansen. *Catching ourselves in the act*. The MIT Press, 1996.

[Her84] J. Heritage. *Garfinkel and ethnomethodology*. Polity Press, 1984.

[HJ89] I. J. Hayes and C. B. Jones. Specification are not (necessarily) executable. *Software Engineering Journal*, pp.330-338, November 1989.

[HKL95] R. Hirschheim, H. K. Klein, and K. Lyytinen. *Information systems development and data modeling: conceptual and philosophical foundations*. Cambridge University Press, 1995.

[HKN91] R. Hirschheim, H. K. Klein, and M. Newman. Information systems development as social action: theoretical perspective and practice. *OMEGA*, 19(6): 587-608, 1991.

[Hol90] C. H. Holbrook III. A scenario-based methodology for conducting requirements elicitation. *Software Engineering Notes*, 15(1):95-104, 1990.

[HORRS95] J. Hughes, J. O'Brien, T. Rodden, M. Rouncefield, and I. Sommerville. Presenting ethnography in the requirements process. In *Proc. of the 2nd Inter. Sym. on Requirements Engineering*, pp.27-34, 1995.

[Hug97] L. Hughes. *Introduction to data communication.* Jones and Bartlett, 1997.

[Hut95] E. Hutchins. *Cognition in the Wild.* MIT Press, 1995.

[IEEE90] IEEE-Std. '610'. *IEEE standard Glossary of Software Engineering Terminology*. Institute of Electrical Electronics Engineers, New York.

[ITC98] Defence system never worked. *The IT Newspaper:Computing*, 26 November 1998.

[Jac92] I. Jacobson. *Object-oriented software engineering: a use case driven approach*. Addison-Wesley, 1992.

[Jal97] P. Jalote. *An integrated approach to software engineering.* 2nd edition. Springer-Verlag, New York, 1997.

[Jam96] W. James. Essays in radical empiricism. Bison books, 1996.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-oriented software engineering*. Addison-Wesley, 1992.

[Joh94] G. J. Johnson. Of metaphor and the difficulty of computer discourse. *Communications of the ACM*, 37(12):97-102, 1994.

[JP93] M. Jarke and K. Pohl. Establishing Vision in Context: Towards a model of Requirements Processes. The *14th Inter. Conf. on Information Systems*, USA, 1993. Also in *NATURE* Project Report NATURE-93-10.

[JP94] M. Jarke and K. Pohl. Requirements Engineering in the Year 2001: On (Virtually) Managing a Changing Reality. *Software Engineering Journal*, 9(6):257-266, 1994.

[JPRS94] M. Jarke, K. Pohl, C. Rolland, and J. Schmitt. Experience-based method evaluation and improvement: a process modelling approach. *IFIP trans. Computer Science and Technology*, 55:1-27, 1994.

[Kir91] D. Kirsh. Foundation of AI: the big issues. *Artificial Intelligence*, 47(1):3-30, 1991.

[KJ98] M. Knapik and J. Johnson. *Developing intelligent agents for distributed systems*. McGraw-Hill, 1998.

[KS98] G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques*. John Wiley & Sons Ltd., 1998.

[Kuw93] E. Kuwana and J. D. Herbsleb. Representing knowledge in requirements engineering: an empirical study of what software engineers need to know. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, pp.273-276, 1993.

[Kyn91] M. Kyng. Designing for cooperation: cooperating in design. *Communications of the ACM*, 34(12):65-73, 1991.

[Lan87] F. Land. Adapting to changing user requirements. In R. Galliers (ed.), *Information analysis: selected readings*, pp.203-221, Addison Wesley, 1987.

[Law98] B. Lawrence. Designers must do the modelling. *IEEE Software*, 15(2):31-33, 1998.

[LdI95] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proc. of 1st Inter. Conf. on Multi-Agent Systems*, pp.254-260, AAAI Press, 1995.

[Leh94a] M. M. Lehman. Introduction to FEAST. 1st *FEAST workshop*, pp.6-10, 1994.

[Leh94b] M. M. Lehman. Some characteristics of S-type and E-type software. *2nd FEAST workshop*, pp.11-26, 1994.

[Leh97] M. M. Lehman. Process models – where next? In *Proc. ICSE 19*, pp.549-552, Boston, 1997.

[Leh98] M. M. Lehman. Software's future: managing evolution. *IEEE Software*, 15(1):40-44, 1998.

[Leh98a] M. M. Lehman. Feedback, evolution and software technology – the human dimension. *ICSE 20 Workshop on Human Dimensions in Successful Software Development*, Japan, 1998.

[LG97] M. Lea and R. Giordano. Representations of the group and group processes in CSCW research: a case of premature closure? In G. C. Bowker et al (eds.), *Social science, technical systems and cooperative work: beyond the great divide*, pp.5-25, Lawrence Erlbaum Associates, 1997.

[LH94] N. G. Leveson and M.P. Heimdahl. Requirements specification for process-control systems. *IEEE trans. on Software Engineering*, 20(9):684-707, September 1994.

[Liv87] E. Livingston. *Making sense of ethnomethodology*. Routledge &Kegan Paul, 1987.

[LK95] P. Loucopoulos and V. Karakostas. *System requirements engineering*. McGraw-Hill, 1995.

[LL94] V. Lalioti and P. Loucopoulos. Visualisation of conceptual specifications. *Information systems*, 19(3):291-309, 1994.

[Lou94] P. Loucopoulos. The F3 (from fuzzy to formal) view of requirements engineering. *Journal of Ingénierie des systèmes d'information*, 2(6):639-655, 1994.

[LR91] Luqi and W. Royce. Status report: computer-aided prototyping. *IEEE Software*, 8(6):77-81, 1991.

[LR98] M. M. Lehman and J.J. Ramil. Feedback, evolution and software technology – some results from the FEAST/1 project. In *Proc. of 11th Inter. Conf. on Software Engineering and its Application*, Paris, 1998.

[Luq93] Luqi. How to use prototyping for requirements engineering. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, pp.229, 1993.

[LW91] J. Lave and E. Wenger. *Situated learning: legitimate peripheral participation*. Cambridge University Press, 1991.

[Mar91] J. Martin. *Rapid application development*. Macmillan, 1991.

[Mat80] H. R. Maturana. Biology of cognition. In H. R. Maturana and F. J. Varela (eds), *Autopoiesis and cognition: the realization of living*, pp.2-62, Reidel, 1980.

[McC78] J. McCarthy. Ascribing mental qualities to machines. Technical report. AI Lab., Stanford University, 1978.

[McD90] D. C. McDermid. *Software engineering for information systems*. Blackwell Scientific Publications, 1990.

[McG92] S. McGinnes. How objective is object-oriented analysis?. In P. Loucopoulos (ed.), 4rd *Inter. Conf. on Advanced Information System Engineering CaiSE' 92*, pp.1-16, Manchester, UK, 1992.

[MGP60] G. A. Miller, E. Galanter, and K.H. Pribram. *Plans and the structure of behavior*. Holt, Rinehart and Winston, 1960.

[Min74] M. Minsky. A framework for representing knowledge. In J. Haugeland (ed.), *Mind design II: philosophy, psychology, artificial intelligence*. The MIT press, 1997.

[Min88] M. Minsky. *The society of mind*. Picador, 1988.

[Mum95] E. Mumford. *Effective system design and requirements analysis: the ETHICS approach*. Macmillan Press Ltd, 1995.

[Nar93] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. The MIT Press, 1993.

[Nau95] P. Naur. *Knowing and the mystique of logic and rules*. Kluwer Academic Publishers, 1995.

[Nes97] P. E. Ness. *Creative software development -- An Empirical Modelling framework*. PhD thesis, Department of Computer Science, University of Warwick, 1997.

[NJJZH96] H. W. Nissen, M. A. Jeusfeld, M. Jarke, G. V. Zemanek, and H. Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, 13(2):37-47, March 1996.

[Nor83] D. A. Norman. Some observations on mental models. In D. Gentner and A.L. Stevens (eds.), *Mental Models*, pp.7-14, Lawrence Erlbaum Associate Inc, 1983.

[OS93] A. L. Opdahl and G. Sindre. Concepts for real-world modelling. In C. Rolland et al (eds.), 5rd *Inter. Conf. on Advanced Information System Engineering CAiSE'93*, pp.309-327, Paris, France, 1993.

[Ous98] J. K. Ousterhout. Scripting: higher-level programming for the 21st century. *IEEE Computer*, 31(3):23-30, 1998.

[Oxf89] *The Oxford English Dictionary*, pp.248, 2nd edition, Oxford University Press, 1989.

[Pau97] J. S. Poulin. *Measuring software reuse: principles, practices and economic models*. Addison-Wesley, 1997.

[PF87] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6-16, January 1987.

[Poh93] K. Pohl. The three dimensions of requirements engineering. In C. Rolland et al (eds.), *5rd Inter. Conf. on Advanced Information System Engineering CAiSE'93*, pp.275-292, Paris, France, 1993.

[Pot93] C. Potts. Software engineering research revisited. *IEEE Software*, 10(5):19-28, September 1993.

[Pot93a] C. Potts. Panel: 'I never knew my requirements were object-oriented until I talked to my analyst'. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, pp.226, 1993.

[PR95] V. Plihon and C. Rolland. Modelling ways of working. CAiSE'95, Finland, 1995. Also in *NATURE* Project Report NATURE-95-09.

[Pre97] R. S. Pressman. *Software engineering: a practitioner's approach*. 4th edition, McGraw-Hill, 1997.

[Pri93] R. Prieto-Diaz. Status report: software reusability. *IEEE software*, 10(3):61-66, May 1993.

[Puf88] P. B. Pufall. Function in Piaget's system: some notes for constructors of microworlds. In G. Forman and P. Pufall (eds.), *Constructivism in the Computer Age*, pp.15-35, Lawrence Erlbaum Associates, 1988.

[Rac95] L. B. S. Raccoon. The chaos model and the chaos life cycle. *Software Engineering Notes*, 20(1):55-66, 1995.

[Rac97] L. B. S. Raccoon. Fifty years of progress in software engineering. *Software Engineering Notes*, 22(1): 88-104, 1997.

[Rao94] A. S. Rao. Agent-oriented programming: an approach to developing distributed real-time systems. In *TOOLS Pacific '94*, 1994.

[RB74] J. Radford and A. Burton. *Thinking: its nature and development*. John Wiley & Sons, New York, 1974.

[Rei65] W. R. Reitman. *Cognition and thought: an information-processing approach*. John Wiley & Sons, New York, 1965.

[Rei92] B. Reinhard. *Prototyping, an approach to evolutionary system development.* Springer-verlag, 1992.

[Rei97] D. Reisberg. *Cognition: exploring the science of the mind.* W.W. Norton, 1997.

[RHHR98] H. Robinson, P. Hall, F. Hovenden, and J. Rachel. Postmodern software development. *The Computer Journal*, 41(6):363-375, 1998.

[RK95] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein (eds.), *Computational theories of interaction and agency*, pp.1-52, The MIT Press, 1995. Also in *Artificial Intelligence*, 73:149-173, 1995.

[RL93] B. Ramesh and Luqi. Process knowledge based rapid prototyping for requirements engineering. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering*, , pp.248-255, 1993.

[Rob99] P. N. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87-92, 1999.

[Rog83] M. F. Rogers. *Sociology, ethnomethodology, and experience: a phenomenological critique*. Cambridge University Press, 1983.

[Rol93] C. Rolland. Modelling the requirements engineering process. In *3rd Europen-Japanese seminar on information modelling and knowledge bases*. Budapest, Hungary, June 1993.

[Rol94] C. Rolland. A contextual approach for the requirements engineering process. In *Proc. 6th Inter. Conf. on Software Engineering and Knowledge Engineering*, 1994.

[Rolt82] L. T. C. Rolt. *Red for danger*. Pan Books, 4th edition, 1982.

[RSB98] C. Rolland, C. Souveyet, and C. BenAchour. Guiding goal modelling using scenarios. *IEEE trans. on Software Engineering*, 24(12):1055-1071, 1998.

[Rus97] S. Russ. Empirical Modelling: the computer as a modelling medium. *Computer Bulletin*, pp20-22, April 1997.

[Rya95] K. Ryan. Let's have more experimentation in requirements engineering. In *Proc. of the 2nd Inter. Sym. on Requirements Engineering*, pp.66, 1995.

[SAGSZ97] M. Stytz, T. Adams, B. Garcia, S. M. Sheasby, and B. Zurita. Rapid prototyping for distributed virtual environments. *IEEE Software,* 14(5):83-92, 1997.

[Sal87] G. Salaway. An organizational learning approach to information systems development. *MIS Quarterly,* 11(2):245-264, 1987.

[SB82] W. Swartout and R. Balzer. On the inevitable intertwining of specification and design. *Communication of ACM,* 25(7):438-440, July 1982.

[SB91] W. Sharrock and G. Button. The social actor: social action in real time. In G. Button (ed.), *Ethnomethodology and the human science*. Cambridge University Press, 1991.

[SB98] P. H. Sun and W. M. Beynon. Empirical Modelling: a new approach for understanding requirements. In *Proc. of 11th Inter. Conf. on Software Engineering and its Application*, Paris, 1998.

[Sch90] S. Schach. *Software engineering*. Aksen, Homewood, 1990.

[SCRB99] P.H. Sun, Y. C. Chen, S. B. Russ, and W. M. Beynon. Cultivating requirements in a situated requirements engineering process. Research report RR-357, Department of Computer Science, University of Warwick, 1999.

[SDV96] S. Some, R. Dssouli and J. Vaucher. Toward an automation of requirements engineering using scenarios. *Journal of Computing and Information,* 2(1): 1110-1132, 1996.

[SH96] R. Silverstone and L. Haddon. Design and the domestication of information and communication technologies: technical change and everyday life. In R. mansell and R. Silverstone (eds), *Communication by design*, pp.44-74, Oxford University Press, 1996.

[She98] C. R. Sheth. *An investigation into the application of the distributed definitive programming paradigm in a teaching environment: the development of virtual electrical laboratory*. Master thesis, Department of Computer Science, University of Warwick, 1998.

[Sho90] Y. Shoham. Agent-oriented programming. Technical report STAN-CS-1335-90, Computer Science Department, Stanford University, 1990.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial Intelligence,* 60(1):51-92, 1993.

[Sid94] J. Siddiqi. Challenging universal truths of requirements engineering. *IEEE Software*, 11(2), 1994.

[SKVS95] I. Sommerville, G. Kotonya, S. Viller, and P. Sawyer. Process viewpoints. In W. Schäfer (ed.), *Software Process Technology: Proc. of 4th European Workshop, EWSPT'95*, pp.2-8, The Netherlands, April 1995.

[Sla90] M. Slade. *Definitive parallel programming*. Master thesis, Department of Computer Science, University of Warwick, 1990.

[Slo90] A. Sloman. Must intelligent systems be scruffy? In J.E. Tiles, G.T. McKee, and G. C. Dean (eds), *Evolving knowledge in natural science and artificial intelligence*, Pitman Publishing, 1990.

[Smi97] A. Smith. *Human computer factor: a study of users and information systems*. McGraw-Hill, 1997.

[Som92] I. Sommerville. *Software engineering*. 4th edition, Addison-Wesley, 1992.

[Som95] I. Sommerville. *Software engineering*. 5th edition, Addison-Wesley, 1995.

[Son93] D. H. Sonnenwald. *Communication in Design*. PhD thesis, The State University of New Jersey, May 1993.

[Son95] D. H. Sonnenwald. Contested collaboration: a descriptive model of intergroup communication in information system design. *Information processing & management*, 31(6):859-877, 1995.

[Son96] D. H. Sonnenwald. Communication roles that support collaboration during the design process. *Design studies*, 17:277-301, 1996.

[SS95] J. Salasin and H. Shrobe. Evolutionary design of complex software (EDCS), *Software Engineering Notes*, 20(5):18-22, 1995.

[SS96] J. Siddiqi and M. C. Shekaran. Requirements engineering: the emerging wisdom. *IEEE Software*, 13(2):15-19, March 1996.

[SS97] I. Sommerville and P. Sawyer. *Requirements engineering: A good practice guide*. John Wiley and Sons, 1997.

[Ste94] D. E. Stevenson. Science, computational science, and computer science: at a crossroads. *Communications of the ACM*, 37(12):85-96, 1994.

[STM95] P. Sallis, G. Tate, and S. MacDonell. *Software engineering: practice, management, and improvement*. Addison-Wesley, 1995.

[Suc87] L. A. Suchman. *Plans and situated actions: the problem of human-machine communication*. Cambridge University Press, 1987.

[Sut96] A. Sutcliffe. A conceptual framework for requirements engineering. *Requirements engineering,* 1(3):170-189, 1996.

[Tri85] R. Trigg. *Understanding social science: a philosophical introduction to the social science*. Blackwell Publishers, 1985.

[Tul88] C. Tully. Representing and enacting the software process. In *Proc. 4th Inter. Software Process Workshop*, UK, May 1988. Reprinted as *ACM SIGSOFT software Engineering Notes*, 14(4):3-4, 1989.

[Tul95] C. Tully. The software process and the modelling of complex systems. In W. Schäfer (ed.), *Software Process Technology: Proc. of 4th European Workshop, EWSPT'95*, pp.2-8, The Netherlands, April 1995.

[VF87] J. R. Valusek and D. G. Fryback. Information requirements determination: obstacles within, among and between participants. In R. Galliers (ed.), *Information analysis: selected readings*, pp.139-151, Addison Wesley, 1987.

[Vli93] H. V. Vliet. *Software engineering: principles & practice*. John Wiley & Sons Ltd. 1993.

[VPC98] S. Valenti, M. Panti, and A. Cucchiarelli. Overcoming communication obstacles in user-analyst interaction for functional requirements elicitation, *ACM Software Engineering Notes,* 23(1):50-55, 1998.

[WDP88] R. Watson, G. DeSanctis, and M. S. Poole. Using a GDSS to facilitate group consensus: some intended and unintended consequences. *MIS quarterly,* 12(3):463-478, 1988.

[Web61] *Webster's 3rd New International Dictionary*, pp.40, G. & C. Merriam Co., 1961.

[Weg97] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM,* 40(5):80-91, 1997.

[Wei99] M. A. Weiss. *Data structures & algorithms analysis in Java*. Addison-Wesley, 1999.

[Wel97] B. B. Welch. *Practical programming in Tcl and Tk*. 2nd edition, Prentice-Hall, 1997.

[WF86] T. Winograd and F. Flores. *Understanding computers and cognition.* Addison-Wesley, 1986.

[Win84] P. H. Winston. *Artificial intelligence*. Addison-Wesley, Reading, MA, 1984

[WJ95] M. Wooldridge and N.R. Jennings. Intelligent agents: theory and practice, *The Knowledge Engineering Review*, 10(2):115-152, 1995.

[WPJH98] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: current practice. *IEEE Software*, 15(2):34-45, 1998.

[XIA98] F. XIA. What's wrong with software engineering research methodology. *ACM Software Engineering Notes,* 23(1):62-65, 1998

[Yeh90] R. T. Yeh. An alternative paradigm for software evolution. In P.A. Ng and R. T. Yeh (eds.*),* *Modern software engineering: foundations and current perspectives*, pp.7-22, Van Nostrand Reinhold, New York, 1990.

[Yeh92] R. T. Yeh. Notes on concurrent engineering. *IEEE trans. On Knowledge and Data Engineering,* 4(5):407-414, 1992

[You83] R. M. Young. Surrogates and mappings: two kinds of conceptual models for interactive devices. In D. Gentner and A.L. Stevens (eds.), *Mental Models*, pp.7-14, Lawrence Erlbaum Associate Inc, 1983.

[You98] E. Yourdon. A tale of two futures. *IEEE Software*, 15(1):23-29, 1998.

[Yun90] Y. W. Yung. *EDEN: an engine for definitive notations*. Master thesis, Department of Computer Sciences, University of Warwick, September 1990.

[Yun92] Y. P. Yung. *Definitive programming: a paradigm for exploratory programming*. PhD thesis, Department of Computer Science, University of Warwick, 1992.

[YY88] Y. P. Yung and Y. W. Yung. *The EDEN handbook.* Department of Computer Sciences, University of Warwick, 1988. Updated in 1996.

[Zav95] P. Zave. Classification of research efforts in requirements engineering. In *Proc. of the 2nd Inter. Sym. on Requirements Engineering*, pp.214-216, 1995.

[Zuc93] L. Zucconi. 'I never realized my requirements were object-oriented until I talked to my analyst'. In S. Fickas and A. Finkelstein (eds.), *Proc. of the 1st Inter. Sym. on Requirements Engineering, pp.230,* 1993.

Glossary

ADM (Abstract Definitive Machine): a tool that is used to animate an LSD account..... 37

ADTs (abstract data types): a programmer-defined data type whose logical behaviour is defined by a set of values and a set of operations on those values......... 163

agency: an attributed responsibility (or privilege) for a state change to an agent............... .... 25, 104, 106, 107

agent: an instigator of change to observables and dependencies............ 25, 104, 106, 107

a-modeller: the modeller in the E-modelling who performs modelling activity from the perspective of the component agent.. 87

A-modeller: the modeller in the I-modelling who performs modelling activity from the perspective of the component agent.. 91

A-node: the points of the star-type logical configuration in dtkeden that are occupied by the A-modellers 131

being-participant-observer: the person who participates in pretend play..................... 93

broadcast mode: an interaction mode that allows each arriving message to be propagated to all A-modellers as if all A-modellers are in a meeting 147

client-server communication: a technique of network communication 132

construal: a means of interpreting unfamiliar experience and communicating one's trial interpretation 30, 78

definitive notation: a simple programming notation for formulating definitions.......27,39

definitive script: a set of definitions... 39

dependency: an empirically established relationship between observables.................... 25

distributed cognition: cognition for a group activity as a process socially distributed between individuals and artefacts [Hut95].. 82

dtkeden: a tool that is developed to support the distributed perspective on EM........... 127

E-modelling: an approach to modelling in which each modeller is an external observer.... 87

ethnomethodology: the empirical investigation of the methods people use to make sense of and at the same time accomplish communication, decision making, reasonableness, and action in everyday life 83

E-type software: software such that the software system itself and its operational domain are conceptually unbounded and continually change..................... 66

evolutionary strategy: a strategy that develops a software system by treating it as an evolving system whose behaviour is adapted to its rapidly changing environment in a situated manner.. 120

experimental strategy: a strategy that develops a software system through experiments in order to provide the developer with experiential information about how the system will behave.. 115

experimentation: the modeller's exploratory interaction with the computer-based model or the referent in the real world.. 34

external observer: the modeller who focuses on attributing state change to a particular agent.. 88

external process: a process that is invoked by a human agent to change the state of his/her external environment 59

generalisation: a process that identifies certain observables with the same characteristics and leads the modeller to create a GO 158

GO (generic observable): an observable that is created to correspond to the modeller's experience, which is inside the modeller's mind and emerges from repeated description of certain observables with the same characteristics 157

Gruber and Sehl's shadow-box experiment: an experiment that indicates the features of construal 79

I-modelling: an approach to modelling in which modellers are internal observers......... 90

intentional strategy: a strategy for design that is informed by an intentional stance, in which an object is treated as a rational agent with beliefs, desires and other mental states exhibiting intentionality 112

interaction mode: the distributed environment that is provided to support the interpersonal communication between A-modellers 145

interference mode: an interaction mode that allows the S-modeller to intervene in the interaction between A-modellers.. 148

internal observer: a modeller who models the agency of agents from the perspective of a participating agent.. 92

internal process: a process that is invoked by a human agent to change his/her mental model 59

ISM (interactive situated model): a computer-based model that enables participants to collaborate interactively in order to cultivate requirements in an incremental fashion.. 232

knowledge construction: a process of capturing or constructing knowledge 60

knowledge representation: a process of recording knowledge by means of representational media 60

localisation: a process that associates a given definition with a particular virtual agent context 155

LSD (language for specification & description): an open-ended notation used to describe the modeller's observation of the referent in the real world 36

LSDagent: a system agent that is implemented in dtkeden for dealing with access privilege to observables 151

normal mode: an interaction mode in which the interaction between A-modellers is mediated by the computer with reference to specified privileges of A-modellers to access observables 149

observable: a characteristic of a subject to which an identity can be attributed 25

observation: the activity that is associated with identifying, monitoring and classifying features of the referent in a particular situation 33

ODM (open development model): a model that supports open development for software system development 53, 69

particularisation: a process that reuses the definitive pattern of a GO 160

pretend play: an A-modeller pretends to be an agent within the application by enacting the ordinary interaction with other agents 91

private mode: an interaction mode that allows each A-modeller to interact with the S-modeller individually 148

REP (requirements engineering process): a process of developing requirements 222

requirements cultivation: a central activity in the SPORE framework in which participants can collaboratively interact with each other and with their environment to develop requirements 234

requirements: a condition or capability that must be met or possessed by a system to satisfy the condition or capacity needed by a user to solve a problem or achieve an objective 222

S1-modelling: a modelling activity that is centred around an external observer who can examine the system behaviour 77

S2-modelling: a modelling activity that is performed by human agents from the perspectives of the component agent of a system 77

situated activity: a coherent sequence of situated actions 19

situated structural coupling: a structural change to the relationships amongst observables effected by coupling a new definitive script with the current structure through dependency maintenance 63

s-modeller: the modeller in E-modelling who performs modelling activity from the perspective of the whole system 86

S-modeller: the modeller in I-modelling who performs modelling activity from the perspective of the whole system 96

S-node: the centre of the star-type logical configuration in dtkeden that is occupied by the S-modeller 131

SPORE (situated process of requirements engineering): a human-centred framework whereby requirements as solutions to the identified problems in the application domain are developed in an open-ended and situated manner 233

star-type logical network configuration: a logical configuration for the network communication that is used to represent the logical interconnection between software components 131

S-type software: a software system that is completely defined by a fixed specification.. 66

tkeden: a tool that is used to support the concepts and principles of Empirical Modelling 37

virtual agent: a mechanism that generates definitions in a context *as if* they were being generated by an agent in a similar fashion 154