Pilkington Library

Author/Filing Title JONES, R.M.

Accession/Copy No. 040147258

Vol. No. .... Class Mark ....

LOAN COPY

26 JUN 1998

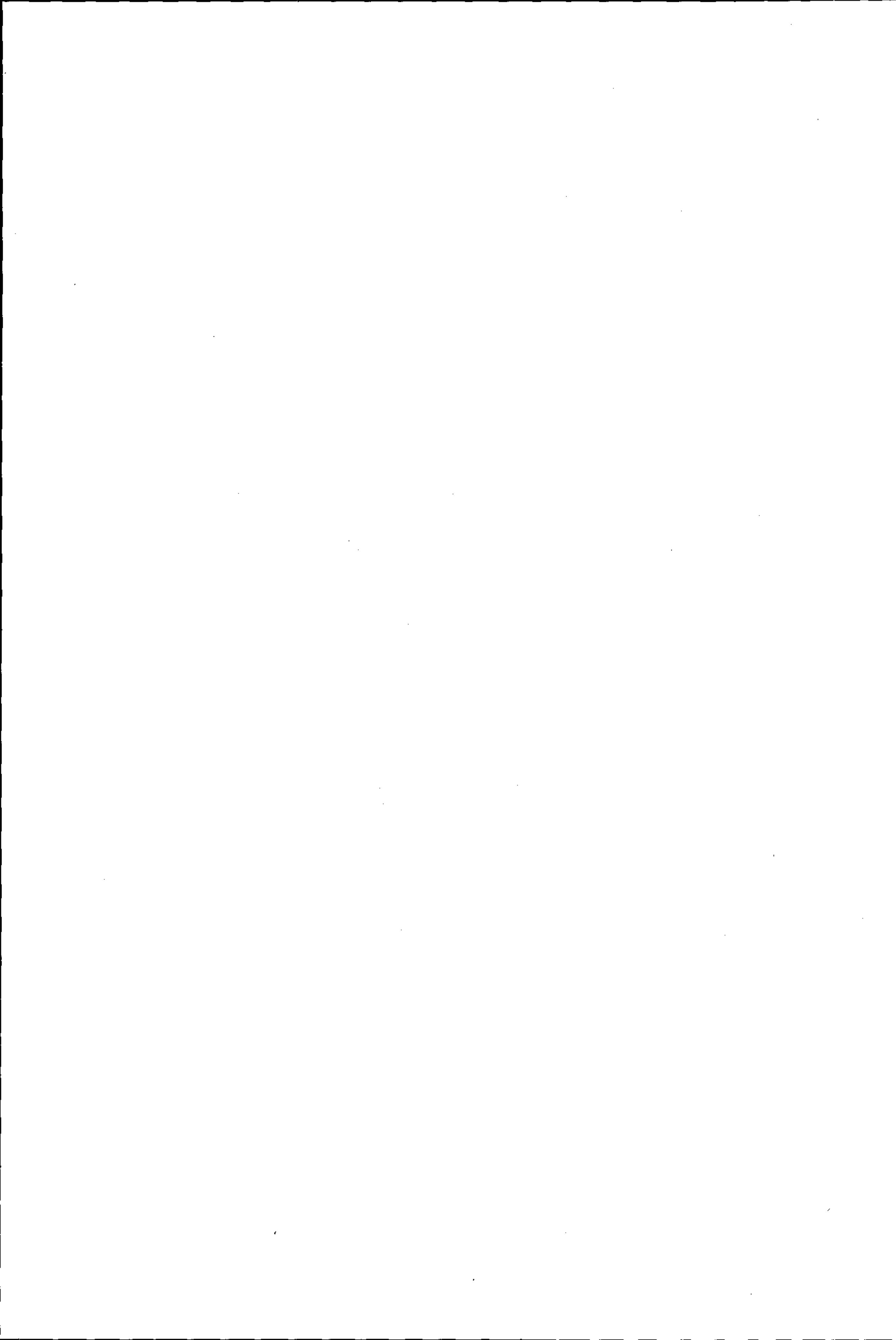
25 JUN 1999

23 NOV 1999

13 DEC 1999

0401472582

BADMINTON PRESS
UNIT 1 BROOK ST
SYSTON
LEICESTER. LE7 1GD
ENGLAND
TEL. 0116 260 2917



An Analysis Framework for CSCW Systems

by

Rachel M. Jones

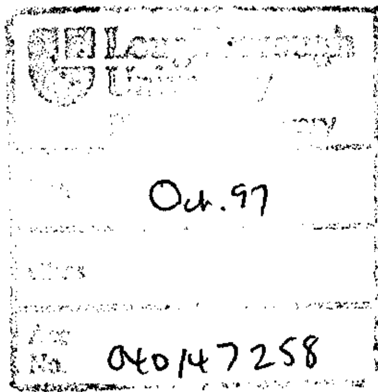
A Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy of Loughborough University

April 1997

© by Rachel M. Jones (1997)


Oct.97
040147258
9/9099299

Abstract

Software toolkits are under development to help construct applications that support group-working. Toolkit developers adopt different approaches to group-work support in order to tackle different issues and a toolkit is commonly characterised by the approach adopted. It is difficult to compare toolkits because of this lack of apparent commonality and it is difficult to decide which toolkits meet specific application requirements.

The aim of this thesis is to develop a framework that provides a means of understanding and comparing software toolkits which support applications in group-working. This Framework offers a description of the functionality required to support group-work and does not consider how the functionality is implemented. The description identifies a set of properties which formally address the ontological and functional aspects of group-working by applying extended entity relationship and state transition techniques to collaborative work. The description primarily focuses on group-work support for real-time systems and does not encompass the additional coordination mechanisms employed to facilitate the group process.

The Framework is used as an analysis tool to characterise seven existing software toolkits which are representative of toolkits in the Computer Supported Collaborative Work (CSCW) domain. The toolkits chosen are: Rendezvous, GroupKit, Dialogo, Suite, SEPIA, MEAD and SOL. The properties supported by each toolkit become apparent in the analysis, including some interesting insights into toolkit functionality. The set of properties correspond to a set of user behaviours, and as a result, it is possible to indicate those aspects of group-working that are supported or not supported by a particular toolkit. The functionality of group-work support provided by each toolkit is made explicit and can be compared.

The Framework is shown to be a useful analysis tool for characterising and comparing software toolkits in terms of the functionality they provide for supporting group-work. The characterisations of the toolkits provide a sound basis on which to assess a toolkit's suitability in meeting the specific group requirements of an application.

To my father, mother, brother and all my friends

Acknowledgements

My friends and my family have given me much love and support throughout this work and I would like to thank them all. A special thanks must go to Steve whose endless understanding and support helped me continue with this work.

I would like to thank Ernest Edmonds and Jim Alty for their professional interest. I would also like to thank Mick Rowlinson for his encouragement over lunchtime discussions in Nottingham eateries.

I would like to thank Steve Mead and Tom Rogers for spending many hours typing, and I would like to thank Nick Haines, Hazel Marshall and Peter Wells without whom I could not have completed this work.

I would like to thank Paul Clements, Paul Dourish, Steve Freeman, Chris Hinde and Allan MacLean for thoughtful and interesting discussions, and for reading a draft of this thesis and providing many helpful comments.

A very special thanks must go to my supervisor Chris Hinde without whose support this thesis would not have been produced.

"This above all: to thine own self be true,
And it must follow, as the day the night.
Thou canst not then be false to any man."

Shakespeare

Table of Contents

CHAPTER 1: INTRODUCTION 10

1.1. Introduction 10

1.2. CSCW 10

1.3. The need for an analysis tool 12

1.4. Focus of the work 14

1.5. Research method 15

1.6. The role of the Framework in CSCW 17

1.7. Structure of the thesis 18

CHAPTER 2: A REVIEW OF CSCW SYSTEMS 20

2.1. Introduction 20

2.2. Comparing systems 20

2.2.1. Five categorisations 21

2.2.2. Mechanisms for characterisation 35

2.2.3. The approach adopted 39

2.3. Existing systems 40
2.3.1. Rendezvous 41
2.3.2. GroupKit 45
2.3.3. Dialogo 48
2.3.4. Suite 54
2.3.5. SEPIA 56
2.3.6. MEAD 58
2.3.7. SOL 60

2.4. Summary 62

CHAPTER 3: A FRAMEWORK OF COLLABORATION 63

3.1. Introduction 63

3.2. Session Management 66
3.2.1. Entity relationship model 67
3.2.2. State transition diagrams 70

3.3. Data Sharing 80
3.3.1. Entity relationship model 80
3.3.2. States 85
3.3.3. State transitions 93

3.4. Concurrency control 96
3.4.1. Entity relationship model 96
3.4.2. State transition diagrams 102
3.4.3. Conflict reduction 103
3.4.4. Repair 104

3.5. Integrated Framework 106
3.5.1. Entity relationship model 107
3.5.2. State transitions 107

3.6. Conclusions 109

CHAPTER 4: CHARACTERISTICS OF EXISTING SYSTEMS 111

4.1. Introduction 111

4.2. Rendezvous 111
4.2.1. Session Management 111
4.2.2. Data Sharing 115
4.2.3. Concurrency Control 120
4.2.4. Comments 123

4.3. GroupKit 124
4.3.1. Session Management 125
4.3.2. Data Sharing 128
4.3.3. Concurrency Control 131
4.3.4. Comments 132

4.4. Dialogo 133
4.4.1. Session Management 133
4.4.2. Data Sharing 136
4.4.3. Concurrency Control 139
4.4.4. Comments 140

4.5. Suite 141
4.5.1. Session Management 141
4.5.2. Data Sharing 143
4.5.3. Concurrency Control 147
4.5.4. Comments 148

4.6. SEPIA 149
4.6.1. Session Management 149
4.6.2. Data Sharing 154
4.6.3. Concurrency Control 157
4.6.4. Comments 158

4.7. MEAD 159

4.8. SOL 160

4.9. Summary 161

CHAPTER 5: COMPARING SYSTEMS 162

5.1. Introduction 162

5.2. Session management 162

5.3. Data sharing 166

5.4. Concurrency control 170

5.5. Conclusions 172

CHAPTER 6: CONCLUSIONS 174

6.1. Introduction 174

6.2. Summary 174

6.3. Method of analysis 175
6.3.1. Research method 175
6.3.2. Method of use 176

6.4. Further work 177
6.4.1. Refinements to the Framework 177
6.4.2. Extending the scope of the Framework 178
6.4.3. Potential uses of the Framework 180

6.5. Concluding remarks 181

REFERENCES 183

Chapter 1:

Introduction

1.1. Introduction

The aim of this thesis is to develop a useful tool for understanding and comparing software toolkits which support the construction of applications in group-working. The tool considers the functionality required by a group of people to use computer-based tools (or software applications) to perform a common task. Such functionality includes the ability for users to view the output of an application, to interact with the application and to change the running application. This functionality is collectively called group-work support. The tool consists of a set of properties which describe group-work support and is presented as a framework. The Framework characterises software toolkits by the properties they support in a common descriptive form. Thus the functionality of toolkits becomes apparent and can be directly compared.

This chapter begins by briefly outlining the associated field of research Computer Supported Cooperative Work (CSCW). Section 1.3 examines the problem which this thesis addresses, that is, the lack of an analysis tool for assessing software toolkits. Section 1.4 describes the focus of the tool and thus defines the scope of the work. Section 1.5 describes the research method used to develop the tool and subsequently test its usefulness. Section 1.6 outlines the structure of the thesis.

1.2. CSCW

The phrase "Computer Supported Cooperative Work" was coined by Irene Greif and Paul Cashman for a workshop held at MIT in 1984 and refers to a set of concerns about multiple individuals working together with a computer system. Bannon and Schmidt (1991) review the meaning of the phrase and its core issues. They emphasise that the focus is to understand cooperative work, so as to better support it.

> "CSCW should be conceived as an endeavour to understand the nature and characteristics of cooperative work with the objective of designing adequate computer-based technologies."

The diversity of CSCW is indicated by the different disciplines involved and the different aims of those working in the field. CSCW is a multi-disciplinary field that arose in part from Human Computer Interaction (HCI) and currently involves the following disciplines: computer science, particularly HCI and distributed systems, psychology, sociology and anthropology, and management studies. There are those people working in CSCW who are focused on developing computer systems to support group-work. There are those who focus on the design of the technology in the hope that it will support cooperative work in the sense of a workplace democracy. There are social scientists who are interested in studying the use of novel CSCW applications and also in showing how their kind of analyses of group processes might affect the future design of CSCW systems. In addition, as the technologies and organisational structures change, so do the topics. For example, the availability of high-speed networks and the broad appeal of the Internet have led to new areas of study in CSCW.

A typical list of topics, including details of some of the changes in research issues in CSCW, might include:

• Places for collaboration. At one time this would have referred to multimedia spaces, although currently it is more likely to refer to virtual environments. Video conferencing issues are still discussed but are less topical.

• Technologies for sharing. This topic includes technologies that support a particular type of task, such as a meeting or collaborative editing. It has developed from specific applications to software toolkits for building applications and their associated research issues.

• Workflow and information sharing. As a result of personal computers proliferating throughout the workplace, networking, distributed organisations and new management techniques, such as Business Process Reengineering (BPR), workflow and information sharing have become very topical areas of research in recent years.

• Workplace studies. These provide a greater understanding of existing work practices.

• CSCW mechanisms. These refer to examining and providing support for the specific requirements of groups, such as awareness, tailorability and concurrency.

• Theories and models of coordination and collaboration. These explain and describe what Gerson and Star (1986) call articulation work, which consists of all the tasks needed "to coordinate a particular task, including scheduling sub-tasks, recovering from errors, and assembling resources".

• Evaluation techniques. Evaluating software support for groups brings its own set of difficulties which are associated with the different backgrounds and the different personalities of individual users that make up groups and the resulting variability. It requires the development of novel techniques for evaluating groupware.

• Ethnographic methodologies. These refer to the development of new techniques for analysing the workplace and informing the design of CSCW systems. The techniques originate in sociology, but the form of the analysis is appropriated to CSCW and the actors and methods employed subsequently changed.

1.3. The need for an analysis tool

Considerable effort is being put into developing software toolkits to help construct applications that support group-work. Given specific application requirements, an application developer must decide which toolkit is appropriate. Toolkit developers adopt different approaches to group-work support to tackle different issues and a toolkit is commonly characterised by the approach adopted. Because of this lack of apparent commonality, it is difficult to compare toolkits and to assess which toolkit meets the specific group requirements of an application. Toolkits need to be characterised in a form which expresses their functionality and makes their commonalities and differences evident.

The motivation for this work is to gain an understanding of CSCW systems to the extent that systems can be compared. This requires that features supported by systems are evident from the analysis. In addition, the depth of the analysis is determined by its use for comparison purposes; the analysis must be able to show the commonalities and differences between systems. The aim is to develop a means of answering questions such as: what does a system support? What does a system not support? How is a particular system related to other systems? What are the differences?

Up to now, there has been no described understanding of the functionality required to support group-work and therefore no common understanding of CSCW systems. The most relevant literature includes the various categorisations and characterisations of systems.

CSCW systems are often presented by category. Categories are formed as a result of generic classes of system developing in the field, such as meeting room support systems. In addition, categories are formed in an effort to position work in the field, such as systems that embody different models of communication. Categories are broad by their very nature; they are formed to encompass a number of systems. Categories are not usually associated with group requirements but with system features or underlying models. They give a high-level analysis of systems but little understanding of those systems.

Various models have been developed to characterise CSCW systems. The characterisation might be used for analysis purposes or to specify an implementation. Some models focus on a particular aspect of a CSCW system. All the models support a high-level analysis which does not give sufficient depth of understanding to enable systems to be compared.

The approach adopted in this thesis and the novel aspects of this work are:

1 To understand the components of a CSCW system.

2 To determine a suitable granularity of component that allows the features of a system to be expressed and provides a sufficient level of understanding to show the commonalities and differences between systems.

3 To understand how the components of a system are related and how they fit together.

4 To understand how the components and their relationships change over time.

5. To develop the notion of a framework for understanding and comparing CSCW systems.

1.4. Focus of the work

This section defines the scope of the thesis by clearly setting out the intentions of this work. To reiterate, the aim is to develop a framework that provides a means of understanding and comparing software toolkits which support the development of applications in group-working. This indicates the focus is on toolkits rather than applications, though it does not preclude applications. The intention is to understand the generic aspects of supporting group-work as opposed to the task-specific aspects. It is assumed therefore that group-work support can be considered separately to task-specific support. For example, the activity of starting an application in a group involves functionality not required by a single-user, such as synchronising multiple users' views of the application, and is separate from tasks the group wishes to perform with the application. A toolkit is expected to support a sub-set of the functionality described in the Framework. Applications built using a toolkit are expected to support a sub-set of a toolkit's functionality. Applications also support task-specific functionality and are considered to be context-dependant, that is, developed for a particular domain, task and group of users. In summary, the aim is to develop a framework which is context-independent and describes the generic aspects of group-work support.

The intention is to understand the functionality required to support group-work. It is not the intention of this work to describe additional coordination mechanisms which might be employed to facilitate the group process. For example, the activity of accessing a document requires a data consistency policy. Such a policy is described by the Framework. However, restrictions on accessing a document imposed by a coordination mechanism, such as a document requiring copyright protection before publication, are not described by the Framework. Similarly, the Framework describes the functionality required to enable a group to use an application developed to support a group-specific task, such as group decision-making, but does not describe the group-specific support which aids the decision-making process. As a result of this focus, a significant part of the functionality described in the Framework is required for real-time activities and supported by real-time applications, although functionality necessary for supporting activities at different times is also described.

The Framework is intended to be used as an analysis tool. The Framework is tested by using it to characterise and compare seven existing software toolkits. The thesis is not about demonstrating the utility of the Framework by implementing it.

The intention is to develop a framework which can be used to express the features supported by a software toolkit. The thesis does not address application requirements and no assessment is made of the applicability of a particular feature in a given context.

The intention is that the Framework can be used to analyse any CSCW system. The Framework is not derived or aligned with any one particular system but examines the functionality of a system required to support group-work. The Framework is not implementation-dependent and in some areas avoids further detailed specification in order to retain this feature.

The Framework does not attempt to describe how the functionality is presented to users. The form of presentation is considered to be context-dependent. Interface designers can use the Framework as a basis for their design and are not restricted to any particular realisation.

The Framework provides a description of computer support for group-work. Group-work can also be described by social models and cognitive models. The Framework is intended to contribute to an overall understanding of adequate support for collaborative work.

1.5. Research method

Having identified the problem area, indicated the approach and novel aspects of the work, and defined the boundaries to the work, it is necessary to examine how the aim of the work can be achieved. A framework that describes the functionality required to support group-work is needed.

The functionality is divided into three broad areas: session management, data sharing and concurrency control. Although these areas are inter-related, it is necessary to decompose the functionality into manageable parts in order to obtain a greater understanding. Each area is considered to support an important aspect of the functionality. Data sharing considers the different classes of data which might be expected in an application and the sharing of these classes of data amongst users. Concurrency control refers to the functionality that enables multiple users access to the same application and to the same data classes at the same time. Session management refers to the management of users and applications involved in a group activity. This decomposition is a common way of considering the functionality in CSCW systems.

The order in which each area is discussed results from a subsequent area requiring an understanding of a previous one. Session management was examined first in order to obtain a broad overview of the components in group-work. Data sharing identifies different classes of data and different shared states. Concurrency control considers managing user access to the data classes and shared states.

The Framework is developed by understanding the components of a CSCW system and the underlying structure of the components in each area of functionality. Entity relationship modelling is chosen as an appropriate technique to examine and describe the components. Entity relationship modelling is relatively accessible and broadly understood. State transitions are used to examine and describe how the components and their relationships change over time. The entities, entity relationships, states and state transitions constitute a description of the functionality of group-work support. Both entity relationship modelling and state transitions enable the final description of group-work support to be considered as both consistent and complete by derivation.

Once the Framework is specified, it is necessary to test its usefulness as an analysis and comparison tool. The first stage of the assessment is to characterise a representative set of systems in CSCW and the second stage is to compare these characteristics. The representative set of systems are chosen to provide a suitable test vehicle for the Framework based partly on their establishment as important, innovative systems in CSCW, and partly on the breadth of functionality that they encompass as a set.

Each system is characterised by identifying the entities, entity relationships, states and state transitions supported by a system and comparing them to those specified in the Framework. This indicates the properties supported by each system. The properties of the different systems can then be compared with one another.

The characterisations of the systems are expected to vary in detail according to the order in which the systems are analysed. The latter systems will be done in the context of having done the former. Therefore, the characterisations of the first few systems are expected to be more detailed.

1.6. The role of the Framework in CSCW

The aim of the thesis is to develop a Framework to understand real-time CSCW systems. Although toolkit developers have developed innovative mechanisms to support group-work, their approaches have primarily been technology-driven. The toolkits are commonly characterised by the innovative mechanism employed, e.g. "open protocols" in GroupKit (Roseman & Greenberg, 1996a), and the domain of interest, e.g. shared visual environments in GroupKit. It is difficult to compare toolkits and select which toolkit could be used to construct a particular application. Further, the toolkits are described in terms of the innovative technological mechanism employed and the implications for users are rarely stated, or even considered. The Framework intends to provide a common understanding of the functionality such systems support to enable systems to be compared. Further, this understanding is intended to provide a mapping between the technological approaches and their implications for the functionality provided to users. It aims to describe the system functionality in terms of its effect on supporting a group of users.

The broader aim of the thesis is to guide the development of groupware applications. The Framework could be used to choose the most appropriate toolkit for constructing an application. Alternatively, the Framework could be used as a description of the functionality that might be supported by a toolkit. In this case, it could be used by future designers of system toolkits, or it could be used to guide the assessment of group requirements for an application, or it could be used to guide the evaluation of a system.

The Framework describes the underlying system functionality in group-work support. *It focuses on understanding real-time systems in CSCW*, such as computer conferencing systems and shared applications, where there is a lack of understanding in the functionality provided and the implications it has for supporting users. The Framework focuses on three main aspects of functionality: session management, data sharing and concurrency control. To reiterate, session management refers to the management of users and applications involved in a group activity. Data sharing considers the classes of data that are shared amongst users. Concurrency control refers to the functionality that enables multiple users access to the same application at the same time. An example of the lack of understanding is demonstrated by the literature on Rendezvous which is sketchy on the concurrency control it supports and therefore the access users have to applications (Rendezvous is reviewed in section 2.3.1).

The Framework does not aim to fully describe systems such as, workflow systems, message systems and meeting room systems. The latter commonly contain an underlying model of cooperation, such as coordinating activities or decision-making processes. These models have arisen from fields such as sociology, psychology and management studies. Although the Framework describes the underlying mechanisms to support these models, it does not aim to describe the models themselves and therefore cannot be considered to fully describe systems that support the models.

The thesis does not attempt to describe the technological implications for addressing the functionality. It does not examine how the functionality is supported by the technological mechanisms in current systems or how it might or should be supported in future designs. For example, supporting latecomers to a running session requires that a system support dynamic reconfiguration, and supporting different views requires some notion of interface abstraction where the underlying information structures are separated from the interaction component. Similarly, the Framework prescribes neither a replicated or centralised architecture. The Framework specifies the functionality that a toolkit might support and its implications for a group of users.

1.7. Structure of the thesis

The thesis is structured into six chapters. An outline of the next five chapters is given below:

Chapter 2 is divided into two parts. The first part reviews the literature to explain how the approach adopted in this thesis extends previous work. It presents other ways of comparing systems and suggests adopting an approach whereby a system is characterised in terms of a description of the functionality required to support group-work. The second part of the chapter identifies seven existing systems considered to be representative of systems in CSCW and describes each system in detail.

Chapter 3 presents the description of the Framework. The description is divided into three broad categories of functionality: session management, data sharing and concurrency control. Within each category, a set of properties are described in terms of the entities, entity relationships, states and state transitions in group-work support.

Chapter 4 uses the Framework as an analysis tool to characterise each of the seven systems. The characteristics of each system are described in detail.

Chapter 5 compares the characteristics of the systems in order to assess the usefulness of the Framework as a comparison tool.

Chapter 6 concludes the thesis by assessing whether the aim of the work has been achieved. It also describes how others might use the Framework and outlines further work.

Chapter 2:

A Review of CSCW Systems

2.1. Introduction

The central argument of this thesis is that the Framework provides a useful tool with which to understand and compare software toolkits that support the development of applications in group-working. This chapter in divided into two parts. The first part discusses other ways of comparing systems to show why a framework of the sort developed in this thesis is needed. The second part of this chapter discusses which existing systems are representative of systems within the scope outlined, and thus, identifies which systems are to be compared with the Framework. Each system is described in detail.

2.2. Comparing systems

CSCW systems are commonly surveyed by categorising the systems into classes and comparing the classes. Although categorisation is a useful mechanism for examining the extent of systems in CSCW and capturing an overview of the characteristics of systems, it is shown not to be a suitable mechanism for analysing systems. More recently, models of collaboration have been developed to act as analysis tools to support the characterisation of systems. However, these models are shown to be insufficient for comparing systems. A need is highlighted for a framework for characterising and comparing systems.

This section gives an overview of the mechanisms which have been used to compare systems. It begins by outlining a range of categorisations. This has four purposes: firstly, to show that a useful mechanism for characterising systems is needed, secondly, to provide an overview of CSCW systems, thirdly, to set the scope of the Framework, and fourthly, to study the characteristics that have been used to

categorise systems. The section follows with a discussion of the mechanisms for characterising systems, including a description of existing models of collaboration. It ends with an outline of the approach adopted in this thesis.

2.2.1. Five categorisations

Five different categorisations are presented. Although there are many categorisations described in the literature, the categorisations presented here exemplify the different types. Commonalities between the categorisations are apparent. Further, no one categorisation can be seen as giving a complete overview of CSCW systems. The first categorisation is based on the well-known time and place division. The second categorisation is based on classes of systems that have emerged in the CSCW field. The third categorisation is based on the model of communication that a system employs. The fourth categorisation is based on the underlying theory that influences a system. The fifth categorisation is based on the underlying system model.

Each category is illustrated with appropriate systems. An exemplar system from each category is briefly described, indicated by *. Since some of the example systems are used in more than one framework, they are described in a separate section, either section 2.2.1.7 or section 2.3.

2.2.1.1. Time and place

Systems can be categorised by their support for particular characteristics of time and place. The characteristic of time is whether multiple users interact with a system at the same time or different times, and the characteristic of place is the geographical nature of users, that is, whether users are remote or co-located. Figure 2.1 shows the classification space populated with classes of examples.

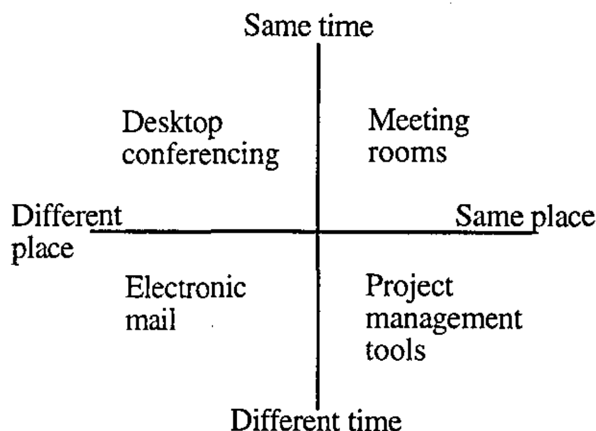


Figure 2.1 Time and Place

Meeting room systems are rooms dedicated to providing computerised support for a group of people. Two quite different examples are CoLab* (Stefik et al., 1987a; Stefik et al., 1987b; Bobrow et al., 1990; Tatar et al., 1991) and GroupSystems* (Nunamaker et al., 1991; Valacich et al., 1991). Other examples include CaptureLab (Mantei, 1988; Mantei, 1989; Halonen et al., 1990; Elwart-Keys et al., 1990) and Project Nick (Cook et al., 1987). Project management tools are located in the same place and require users to submit information at different times. Electronic mail enables users to send messages to one another via known addresses and is now used world-wide. Desktop conferencing refers to systems that enable users who are situated at their own workstations to share applications in real-time and to communicate via audio and video links. Examples include Dialogo* (Lauwers, 1990; Lauwers & Lantz, 1990; Lauwers et al., 1990), MMConf (Crowley et al., 1990), Rapport (Ensor et al., 1988; Ahuja et al., 1990) and SharedX (Gust, 1988).

One of the difficulties with this categorisation is fitting a class of system to a particular category. For example, users might find it useful to use a desktop conferencing system while located in the same room. Further, as Rodden (1991) points out, the division between different time and same time reflects a primarily technological perspective based on the distinction between store and forward and real-time communication systems.

2.2.1.2. Emerging classes

Systems can be categorised according to classes that have emerged in the field of CSCW. Rodden (1991) identifies four major classes of cooperative system:

• Message systems; these enable members of a group to cooperate by exchanging messages. They adopt an approach based on either a formal model, such as Amigo (Danielson et al., 1986), CHAOS (DeCindio et al., 1986; Bignoli & Simone, 1991), The Coordinator* (Winograd & Flores, 1986; Winograd, 1987; Flores et al., 1988), COSMOS* (Wilbur & Young, 1988; Dollimore & Wilbur, 1991) and Domino (Kreifelts et al., 1991), or a semi-formal model, such as, Information Lens (Malone et al., 1987) and Object Lens* (Lai et al., 1988).

• Computer conferencing systems; these are characterised by supporting a shared information space, often augmented by direct user-to-user communication. The computer conferencing systems class differs from the desktop conferencing systems class by including systems that support both interaction at the same time and at different times. Examples of computer conferencing systems include Dialogo*

(Lauwers, 1990; Lauwers & Lantz, 1990; Lauwers et al., 1990), MMConf (Crowley et al., 1990), Rapport (Ensor et al., 1988; Ahuja et al., 1990) and SharedX (Gust, 1988).

• Meeting room systems. Examples include CaptureLab (Mantei, 1988; Mantei, 1989; Halonen et al., 1990; Elwart-Keys et al., 1990), CoLab* (Stefik et al., 1987a; Stefik et al., 1987b; Bobrow et al., 1990; Tatar et al., 1991), GroupSystems* (Nunamaker et al., 1991; Valacich et al., 1991) and Project Nick (Cook et al., 1987).

• Co-authoring and argumentation systems; these are characterised by supporting and representing the negotiation and argumentation that is involved in group-working. Examples of co-authoring systems include Quilt (Fish et al., 1988), PREP (Neuwirth et al., 1990) and GROVE (Ellis et al., 1991). An example of an argumentation system is gIBIS* (Conklin & Begeman, 1988).

Freeman (1994) identifies an additional category:
• Media spaces; these provide visual and acoustic environments that span physically separate areas and have been found to provide awareness amongst participants. Examples of media spaces include the Portland experiment* (Bly et al., 1993), Cruiser (Fish et al., 1992), RAVE (Gaver et al., 1992) and CAVECAT (Mantei et al., 1991).

Ellis et al. (1991) identify similar classes. They extend argumentation systems to group decision support systems and class it with meeting rooms instead of co-authoring systems. They identify two additional categories: intelligent agents, which are responsible for a specific set of tasks and whose actions resemble those of other users, and coordination systems, which are subsequently categorised into the four models of communication listed in the next section.

There appears to be no particular characteristic for this categorisation; the classes have emerged over a period of time.

2.2.1.3. Models of communication

Numerous attempts have been made to categorise models of group communication (Bracchi & Pernici, 1984; Prinz. 1989; Hennessy, 1990). Systems can be categorised by the category of model they support. Prinz (1989) defines four types of model of group communication:

• Procedure-oriented models; these describe office activities as strictly regulated processes, where communication and data flow are rigidly defined. The presence of a coordinating agent which controls the progress of the activity is assumed. Systems that support this category of model include: Amigo (Danielson et al., 1986), COSMOS* (Wilbur & Young, 1988; Dollimore & Wilbur, 1991) and Domino (Kreifelts et al., 1991).

• Form-oriented models; these aim to make procedural knowledge parts of the messages or forms that are passed round during the performance of office activities. This leads to an object-oriented approach to modelling, and a requirement for a user agent to be associated with each communicator to interpret the procedural information inherent in the message. Systems that support this category of model include: Information Lens (Malone et al., 1987) and Object Lens* (Lai et al., 1988).

• Communication structure-oriented models ; the structure between role-players in an office is the central concern. Particular office activities cannot be specified. Systems that support this category of model include: AMIGO MHS+ data model* (Smith et al., 1989).

• Conversation-oriented models; this approach adopts the language/ action perspective (see theory section). Activities consist of one or more conversations with (usually) two participants. Systems that support this category of model include: CHAOS (DeCindio et al., 1986; Bignoli & Simone, 1991), ConversationBuilder* (Kaplan, 1990; Kaplan et al., 1992) and The Coordinator* (Winograd & Flores, 1986; Winograd, 1987; Flores et al., 1988).

Rodden and Blair (1991) suggest similar categories to these when classifying the representation and control of cooperation in CSCW systems, along with two additional categories, conferencing systems and control free systems. In conferencing systems, floor control mechanisms allow only one user to access the shared information space at any time and floor control policies dictate who this is. In control free systems, such as CoLab, no control mechanisms are provided and it relies on meeting participants formulating their own protocols. These two categories might be seen as communication structure-oriented models.

The characteristic of this categorisation is the model of communication underlying the system.

2.2.1.4. Theories / Techniques

Systems can be categorised by the theories or techniques which have influenced their development. A theory is defined as a statement of the principles on which a subject is based. A technique is defined as the method of doing something. Clearly, some cases are statements relating to the principles of collaboration, such as activity theory, and others are methods for analysing and representing collaboration, such as coordination mechanics. In other cases, it is less clear whether something is a theory or a technique, such as situated action or distributed cognition. Rather than enter into a debate, both theories and techniques are listed together.

Many systems are built under the influence of several theories/ techniques. For example, the developers of ConversationBuilder claim to be influenced by the language/ action perspective and situated action.

Some theories/ techniques arose in other fields and are relevant to CSCW, such as activity theory, whilst others have been developed in CSCW, such as coordination theory.

Some of the theories/ techniques that have contributed to the development of CSCW systems are outlined below:

• Activity theory. Kuutti (1991) broadly defines activity theory as a philosophical framework for studying different forms of human praxis as developmental processes, with both individual and social levels linked. The fundamental unit of analysis is an activity which exists in a material context and transforms that context. Activity components include a distinguished object, an active actor who understands the activity and a community who share the same object. The relations between activity components are always mediated by artefacts such as tools, rules and division of labour. An activity is realised through purposeful actions and subconscious operations by participants, resulting in a transformation of the object.

• Argumentation theory. Conklin and Begeman (1988) describe systems to help groups of people record the structure of arguments (e.g. positions, arguments and counter arguments) that are based in part on ideas from philosophy and rhetoric about the logical structure of decision-making. gIBIS is an example one such system (Conklin & Begeman, 1988).

• Contingencies. Bell and Johnson (1993) develop a contingency model which aims to integrate user-oriented design guidelines into the groupware design process, including the specification of the degree of end-user tailorability required to support the dynamics of the group. The contingency model is motivated by the work of Rao and Jarvenpaa (1991) who use Communication Theory, Minority Influence Theory and Human Information Processing Theory to explain the previous use of technology in group decision-making support and derive a set of prescriptive contingencies. Shannon and Weaver (Fiske, 1990) are a major influence on Communication Theory and have exposed three levels of potential difficulty in communication: firstly, how accurately can symbols be transmitted, secondly, how precisely do the symbols convey the desired meaning, and thirdly, how effective is the received meaning. Minority Influence Theory examines the way in which low participation due to minority effects can be rectified, in particular by reducing reticence due to inferiority or role-status effects (Desanctis & Gallupe, 1987). Human Information Processing Theory supports the use of quantitative models to increase the processing of large volumes of data with complex inter-relationships (Rao & Jarvenpaa, 1991).

• Coordination mechanics. Coordination mechanics is an exact scientific approach to coordination, motivated by the wish to turn linked computers into a means of supporting co-ordinated work. Petri nets exert a great influence over this approach. Petri nets are a formalism used to represent resource flows in distributed and parallel systems. They can be used to provide a parallel collaborative computation model. Holt (1988) has designed a Petri net based graphical specification language, Diplans, which makes coordination explicit by describing the logical as well as the physical organisation of co-ordinated activity. The example used is the manufacture of three parts to constitute two assemblies. Furuta and Stotts (1994) suggest a model based upon color time Petri nets. The example used is moderating a real-time session or meeting and changing the moderator.

• Coordination mechanisms. Simone, Divitini and Schmidt (1995) develop a notation for constructing computational coordination mechanisms in CSCW systems. Coordination mechanisms offer structured support for the articulation of distributed activities. Coordination mechanisms are defined as "a category of symbolic artefacts that stipulate and mediate articulation work". Evidence is cited that indicates which coordination mechanisms must be malleable by an end-user and interoperable. The notation is general and supports the specification of coordination mechanisms in terms of the semantic level of articulation work, by the actors themselves and in a cooperative manner.

• Coordination theory. (Malone & Crowston, 1990) view coordination as the act of managing interdependencies between activities performed to achieve a goal. They believe interdependence can be analysed in terms of common objects. Three kinds of interdependence are identified: prerequisite, where the output of one activity is required by the next activity, shared resource, where an object is required by multiple activities, and simultaneity, where more that one activity must occur at the same time. They identify four types of hierarchically dependent processes which underlie coordination: coordination, group decision-making, communication and perception of common objects.

• Distributed cognition. (Hutchins, 1990) focuses on work at a system level where the system is a collection of interacting individuals and artefacts in the propagation of knowledge. Hutchins argues that "tools do not amplify the cognitive abilities of the team members but instead transform what are normally difficult cognitive tasks into easy ones. The progress of various team members through the career cycle of navigation practitioners produces an overlapping distribution of expertise that makes it possible for the team to achieve training and job performance in a single activity."

• Ethnography. Hughes et al. (1994) claim that ethnography aims to understand the social setting as it is perceived by those involved in that setting. As a mode of social research it is concerned with the production of detailed descriptions of the "workaday" activities of social actors within specific contexts. MEAD is an example of a system that has employed an ethnographic study in its development process* (Twidale et al., 1994).

• Language/ action perspective or speech act theory. Flores et al. (1988) propose the following claim as the basis for the language/ action perspective: "..human beings are fundamentally linguistic beings: action happens in language in a world constituted through language." Kensing and Winograd (1991) articulate its relevance for the analysis of cooperative work: "Cooperative work is co-ordinated by the performance of language actions, in which parties become mutually committed to the performance of future actions and in which they make declarations creating social structures in which those acts are generated and interpreted". Examples of systems influenced by the language/ action perspective are CHAOS (DeCindio et al., 1986; Bignoli & Simone, 1991), ConversationBuilder* (Kaplan, 1990; Kaplan et al., 1992), The Coordinator* (Winograd & Flores, 1986; Winograd, 1987; Flores et al., 1988) and The Milan Conversation Model (De Michelis & Grasso, 1994).

• Situated action. Suchman (1987) claims that actions are always situated in particular social and physical circumstances and the situation is crucial to the action's interpretation. The aim is to explore the relation of knowledge and action to the particular circumstances in which knowing and acting invariably occur. The organisation of situated action is an emergent property of moment-by-moment interaction between actors, and between actors and the environments of their action. ConversationBuilder* is an example of a system that is influenced by situated action (Kaplan, 1990; Kaplan et al., 1992).

• Theory of action. Strauss (1993) gives a framework to describe cooperative work ensembles as social worlds that have been formed to meet some shared objective via a commitment to collective action and whose members can be co-located or distributed, to the extent that effective communication can be facilitated. Cooperative work is seen to take place in the context of particular structural conditions and contingencies. WORLDS* is an example of a system that adopts the theory of action (Fitzpatrick et al., 1995; Tolone et al., 1995).

2.2.1.5. System models

Systems can be categorised by their underlying system models. Systems might adopt more than one model. Some of the possible underlying system models include:

• Degree of replication. A shared real-time application must distribute user input and output to multiple sites. If software components are replicated and distributed so they run on local processors, network traffic can be reduced and response times improved. However, inconsistencies between sites can arise. Lauwers et al. (1990) offer a critique of replicated architectures. Systems can adopt different degrees of replication. For example, GroupKit* (Roseman & Greenberg, 1996a) supports a centralised core system and multiple, distributed modules.

• Client/ server model. Most real-time systems employ a window system to present information to users. Window systems adopt the client/ server model.

• Hypertext. Hypertext systems can support the organisation and clustering of documents as contexts of work. Systems such as SEPIA* (Haake & Wilson, 1992) and ConversationBuilder* (Kaplan, 1990; Kaplan et al., 1992) adopt the hypertext model for manipulating documents.

• Agent-based. LIZA (Gibbs, 1989), Object Lens* (Lai et al., 1988) and Oval* (Malone et al., 1992; Malone et al., 1995) employ agent-based technology.

• Degree of collaboration-awareness. Lauwers and Lantz (1990) make the distinction between collaboration-aware systems and collaboration-transparent systems. Collaboration-aware applications are special-purpose applications designed for simultaneous use by multiple users. Collaboration-transparent systems are single-user applications. Shared window systems, for example, are collaboration-aware in part and employ collaboration-transparent applications. Dialogo* (Lauwers, 1990; Lauwers & Lantz, 1990; Lauwers et al., 1990) is an example of a shared window system.

• User Interface Management Systems or UIMS. A UIMS is a software engineering tool which supports the construction of user interfaces while embedding human factors principles. Generally, UIMS support the notion of separability between the user interface and the application functionality. Some systems support aspects of UIMS, whilst others employ UIMS directly. Rendezvous* (Hill et al., 1994) is built on a constraint-based UIMS. Suite* (Dewan & Choudhary, 1992) supports the notion of separability and employs a dialogue manager which corresponds to the dialogue control component in the Seeheim Model (Pfaff, 1985) employed by many UIMS. SOL* (Smith & Rodden, 1995) supports the notion of separability.

The characteristic of this categorisation is concerned with the technology employed by a system and the system architecture.

2.2.1.6. Discussion

Four purposes were given for outlining categorisations. firstly, to show that a useful mechanism for characterising systems is needed, secondly, to provide an overview of CSCW systems, thirdly, to set the scope of the Framework, and fourthly, to study the characteristics that have been used to categorise systems. The second purpose has been achieved. The others are discussed below.

The need for a mechanism to characterise systems

Categorisation involves the identification of classes that are distinguishable by specific characteristics. Either systems are put into identified classes or classes emerge out of the distinguishing features of systems. Two problems arise with adopting categorisation as the mechanism for comparing systems. Firstly, classes become the focus of characterisation rather than systems. As a result, only the characteristics that are common to all systems in a particular class are described

which can be quite limiting, e.g. the characteristic of same place and same time. In addition, systems in the same class cannot be distinguished from one another. Secondly, many systems do not fit neatly into the specified classes. For example, systems may be influenced by more than one theory. It becomes apparent that there is a need to characterise systems, and categorisation can only do this in a limited way. Therefore, there is a requirement for an analysis tool that enables systems to be characterised.

Scope of the Framework

Providing an overview of CSCW systems naturally leads to setting the scope of the Framework. The Framework is an analysis tool for characterising the functionality of systems that support the development of applications in group-working. Many systems support the temporal sequencing of whatever entities are viewed as primary, such as, activities, actions or conversational utterances. The Framework does not describe coordination mechanisms that control the temporal sequencing of such entities; these mechanisms are commonly employed to facilitate and actively support the group process. (Session management functionality involves some sequencing, such as users cannot join a group until the group is formed, but it is minimal.) However, the Framework does describe the functionality necessary to support these mechanisms. As a result, a significant part of the functionality described in the Framework supports real-time systems, although the functionality necessary for systems supporting activities at different times is also described.

If the Framework can be used to analyse systems that support the development of collaborative applications, it can also be used to analyse collaborative applications. In the field of CSCW initially, specific applications were developed to support collaboration, such as RTCAL (Sarin & Greif, 1988), an application to support meeting scheduling. More recently, the collaborative components have been abstracted and used in the implementation of toolkits. Many of the development toolkits provide run-time environments for the applications. The Framework aims to analyse the generic aspects of collaborative support and, therefore, is context independent.

The Framework focuses on characterising group functionality, that is, functionality available to a group of users by way of support for collaboration. The Framework operates at a level which is neither at the social or communicative action level, or at the structural or implementation level. This distinguishes it from other frameworks which might characterise, for example, group communication or social behaviour. Abowd (1994) and Bass (1994), whose interest lies in software architectures and, in

particular, the evaluation of architectures, indicate that the first stage in the development of a reference model for collaboration is a description of the functionality.

The scope of the Framework can be examined in the context of the categorisations outlined above. In the first categorisation, time and place, much of the functionality described in the Framework is appropriate for systems that operate in the same time, different place class. This does not preclude the use of the Framework to analyse systems in other classes. In the second categorisation, emerging classes, much of the functionality described in the Framework is appropriate for computer conferencing systems. Again, the Framework does not preclude systems in the other categories. In the third categorisation, models of communication, the Framework does not describe any of the models of communication. The Framework operates at a different level and enables the analysis of mechanisms required to support any of the models of communication. In the fourth categorisation, theories, the Framework does not support any particular theory. As indicated above, the Framework supports the characterisation of group functionality. In the fifth categorisation, system models, the Framework does not describe any particular system model. The Framework operates at a different level and enables the analysis of mechanisms supported by any of the system models.

Characteristics used to categorise systems

A category is defined in this context as a class of systems which possess some quality or set of qualities in common. A characteristic is defined in this context as a distinguishing quality. Therefore, a category is a class of systems which possess certain characteristics. Since this thesis aims to develop a tool to characterise systems, categorisations can be examined partly from the point of view of studying the distinguishing characteristics of their categories. The characteristics used in the categorisations include: time and place, emerging systems, model of communication, influencing theory and underlying system model. Group functionality is not addressed.

2.2.1.7. Descriptions of example systems

<u>AMIGO MHS+ data model</u>

AMIGO MHS+ project (Smith et al., 1989) aimed to develop a coherent model of group communication support in distributed environments by investigating ways of extending and building on existing message handling systems and other services, such as the CCITT X.500 directory service. The fundamental concept of the data model is the notion of information sharing. There are three elements: information objects,

operations and environments. The data model is intended to provide a basis for a procedural model of group communication which specifies the group activities.

CoLab

CoLab (Stefik et al, 1987a; Stefik et al., 1987b; Bobrow et al., 1990; Tatar et al., 1991) aims to support between two and five people with a shared information space in an informal arrangement. Users see the same image on different workstations, described as WYSIWIS (What You See Is What I See). At hand, users have Liveboard, which is a large wall-mounted display onto which a workstation's display can be projected, Boardnoter, which supports freehand sketching, Cognoter, which supports outlining documents and annotations, and Argnoter, which supports the organisation and evaluation of proposals.

ConversationBuilder

ConversationBuilder or CB (Kaplan, 1990; Kaplan et al., 1992) is influenced by both the language/ action perspective and the situatedness of work practices. These motivate the aim of CB to maximise both active support, which requires knowledge of the activity at hand, and flexibility, which is how well a system responds to a change in activity. CB allows the specification of multiple protocols, unlike The Coordinator, and allows users to work with arbitrary numbers of instantiations of any of the protocols as appropriate to the situation. CB supports the notion of shared data models which can be worked into a hypertext. An obligation facility allows users to dynamically weave interconnections among activities and to model dependency structures.

The Coordinator

The Coordinator (Winograd & Flores, 1986; Winograd, 1987; Flores et al., 1988) was designed to support communication between people in a distributed office environment by making explicit the actions implicit in language. Drawing on speech act theory, especially its elaboration by Searle (1969), Winograd and Flores suppose that communication is constituted of a number of different conversations, each of which involves speech acts, such as requesting, questioning, informing, etc., in a sequential order. The different conversations are represented by transition network diagrams specifying the speech acts which are possible at any given moment in a conversation. These networks are embodied in The Coordinator so that the range of speech acts available to a user at any particular moment is displayed on a menu, and, having selected from the menu, a standard form is presented to the user for completion.

COSMOS

COSMOS (COnfigurable Structured Message Oriented System) (Wilbur & Young, 1988; Dollimore & Wilbur, 1991) was designed to support a number of activities or "communication structures" through a notation called Structure Definition Language or SDL. SDL allows users to define the tasks allocated to different members of a group by associating member roles with actions. SDL allows the expression of constraints on the execution of actions, such as the resources required. Relations of dependence between actions or groups of actions can also be notated. As a result, SDL can express how individual actions get co-ordinated to support socially distributed work. The implementation of SDL in COSMOS permits automated support for the communication structures. The system might, for example, track deadlines, enable individual users to obtain a view of how the work is unfolding and what their place is in it, and ensure that the right actions are prompted for in the right order.

gIBIS

The gIBIS system (Conklin & Begeman, 1988) supports the argumentation process undertaken by system designers. It is based on a model of design deliberation called Issue Based Information System or IBIS. IBIS embodies a model of rhetoric which enables users to raise issues, take positions on them, support or object to positions by means of arguments, ask and respond to questions, etc.. The model specifies how these rhetorical elements can legally be combined to compose an argument over the design rationale. The gIBIS tool provides the necessary consistency maintenance mechanisms to support simultaneous access and update by multiple users in real-time.

GroupSystems

GroupSystems (Nunamaker et al., 1991; Valacich et al., 1991) adopts a formal approach based upon decision support models aimed at reducing dysfunctions in large groups of people (of around fifty). It enforces a process-related structure with three activities: idea generation, idea synthesis and prioritising. It supports a shared central display, and meetings may support different degrees of verbal communication and parallel computer input.

Object Lens

Object Lens (Lai et al., 1988) is described as a semi-formal system, a system that represents knowledge in a way that both people and their computational agents can process intelligently. The knowledge is exposed to users in a way that makes it visible and changeable. Passive information is represented in semi-structured objects with template-based interfaces. Aggregate information from collections of objects is summarised in customisable folders. Active rules for processing information are

represented in semi-autonomous agents. Rules are triggered by events such as the arrival of mail in a folder. Object Lens is based on Information Lens (Malone et al., 1987), a rule-based semi-structured messaging system that can be tailored by end-users.

Oval

Oval (Malone et al., 1992; Malone et al., 1995) consists of a set of primitives which can be tailored by end-users to support asynchronous message-based applications, such as gIBIS, The Coordinator, Lotus Notes and Information Lens. It is based largely on Object Lens* (Lai et al., 1988). Users can create applications by combining four types of primitives: objects, views, agents and links (oval). Semi-structured objects represent things in the world, such as people, tasks, messages and meetings. Views collate objects into particular formats, such as a table or a network, and allow users to edit individual objects. Agents apply a set of rules to a collection of objects which can result in actions on these objects. Links represent relationships between objects. Oval provides three ways of sharing information: by saving objects in a file, by mailing messages containing objects, and a rudimentary version of "live" sharing of objects stored on remote databases.

The Portland Experiment

The Portland Experiment (Bly et al., 1993) arose out of the need to support cross-site work including the necessary social conditions. The media space consisted of a two-way video and audio link which was constantly "on" between two sites connecting two common areas. The media space demonstrated it was possible to support social activity and that a working group can collaborate effectively cross-site as a single entity.

WORLDS

WORLDS (Fitzpatrick et al., 1995) is a CSCW system based on Strauss' theory of action (Strauss, 1993). The basic premise in WORLDS is that people interact within, with and across social worlds to get their work done. The fundamental component of WORLDS is the "locale". A locale provides the setting in which members of social worlds can undertake their collective action or in which different social worlds can meet together for some purpose. Introspect is a specification language for WORLDS which describes a locale as: the primary work activity/ activities, the particulars of the setting, the roles of the people who are participating, domain-specific actions and processes (the co-evolution of setting and action over time). Domain-specific actions provide access control by requiring a user to play specified roles and the actions depict temporal relationships within a process model. Process modelling can be used to depict temporal relationships of activities within and among locales.

2.2.2. Mechanisms for characterisation

A need has been identified for a mechanism to characterise systems, that is, a mechanism to highlight the distinguishing qualities of systems. Such mechanisms are sometimes described as models. Four particular mechanisms contribute to characterising systems.

2.2.2.1 Descriptions of example models

<u>Ellis and Wainer</u>

Ellis and Wainer (1994) develop a conceptual model of groupware in order to characterise and compare systems from the point of view of the user. The scope of the model encompasses a wide range of groupware, from electronic meeting rooms and video conferencing, through to work flow systems. The model consists of three complementary models: an ontological model, a coordination model and a user interface model. The ontological model is a description of the objects and operations in a system. The coordination model is a description of activities (and their orderings) that can be performed by the system. An activity is a potential set of operations (and their corresponding objects) that an actor playing a particular role can perform with a defined goal. A set of activities (and ordering) make up a procedure. The user interface model is a description of the interface with the system. The models take the viewpoint of the user and, as a result, do not capture implementation aspects, such as distribution.

The ontological model, like the other models, can be applied to any system, not necessarily a groupware system, and is application specific. It refers to a description of the functional objects and operations in my model. The ontological model incorporates the concepts of operation rights and access rights.

Coordination can take place at two levels: the object-level and the activity-level. Object-level coordination is concerned with how sequential or simultaneous access to the same objects are managed. Activity-level coordination is concerned with the sequencing of activities that make up a procedure. Each of the moments of the collaboration is called a stage. A stage is a global description of the instances of activities that are active. Some systems are viewed as single-stage, that is, the collaboration model does not provide any temporal sequencing of activities; each of the participants is always engaged in a single task.

The user interface model has three components: views of information objects, views of participants and views of context. Views of information objects refers to the

potential for users to have different views of the same object, to view the collaboration model and to synchronise views. A view includes both input and output interaction capabilities. Views of context refer to structural, social and organisational aspects.

The scope of the Framework focuses on single-stage object-level coordination. This is not characterised by the model proposed by Ellis and Wainer to enable systems to be compared effectively. The Framework does not focus on supporting activity-level coordination. The Framework supports views of information objects and views of participants. It supports an awareness of what other users are doing but does not support views of the social and organisational aspects.

Patterson

Patterson (1994) describes a taxonomy of architectures for real-time groupware applications. Although this might be seen as a characterisation of system models, it can also be viewed as a characterisation of real-time data sharing. Patterson contributed to the development of Rendezvous*, a toolkit that supports the construction of real-time applications (Hill et al., 1994). Rendezvous contains an implicit model of collaboration and this taxonomy contributes to making that model explicit.

The levels of state envisioned within any application are shown in figure 2.2. The display state is defined as the information that drives the user's display. The view state is defined as the information that relates the user's display to the underlying information in the application. The model is defined as the underlying information. The file is defined as the persistent representation of the underlying information.

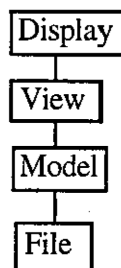


Figure 2.2 Levels of Application State

Patterson indicates two ways of keeping the state consistent between multiple users in real-time groupware: firstly, by maintaining one copy of the application state, and secondly, by using consistency maintenance protocols between multiple copies of the

application state. Patterson proposes three architectures for the former, shown in figure 2.3, and three for the latter, shown in figure 2.4.

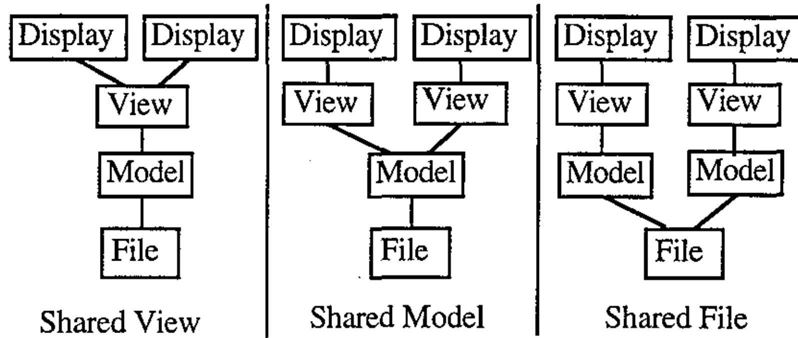


Figure 2.3 Centralised System Architectures

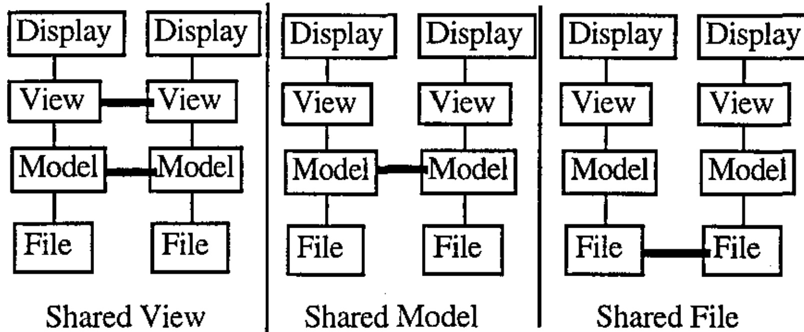


Figure 2.4 Replicated System Architectures

Patterson suggests the most appealing architecture is one that can toggle between view and model sharing, as shown in figure 2.5.

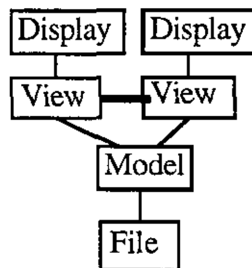


Figure 2.5 Shared Model, Synchronised Views

Patterson indicates that session management and overlays need to be addressed but they remain residual issues in the taxonomy.

This characterisation of data sharing makes a significant contribution to a complete description of data sharing. However, although data sharing is an important aspect of real-time collaboration, other aspects also need to be characterised, such as concurrency control and session management.

DePaoli and Tisato

DePaoli and Tisato (1991) develop a model of cooperation for specifying and designing real-time conferences. They introduce the concepts of role, group and coordinator. A group is introduced for every role. A coordinator is defined as an object whose interface provides a set of groups with a set of primitives to modify group memberships. Actions performed by a coordinator result in the movement of users amongst groups. The actions of a generic coordinator are join, leave and select, where select returns a member of a specified group. For example, where two or more people are word processing the same document, there are two roles: reader and writer. The situation is modelled by defining two groups: a reader group and a writer group. A conversation coordinator moves users between groups and is driven by a floor control strategy. Similarly, a conference coordinator, controls the membership of a conference. DePaoli and Tisato define how coordinators can be combined into a hierarchical structure to form the architecture for a conference control system.

DePaoli and Tisato's model supports the partial modelling of particular aspects of collaboration, such as aspects of session management and concurrency control. For example, concerning session management, the notion of groups supports functions such as join and leave group but is not able to support functions such as start and stop application. The model separates conferencing from the applications, so data consistency is only considered to be maintained through user input control and not, for example, data locking. Further, aspects of data sharing, such as multiple views, are not accounted for.

Edwards

Edwards (1994) presents a model of session management and Edwards (1996) presents a model of access control for collaborative applications. The aims are, firstly, to develop a generic set of session management functions that are application-independent and re-usable, and secondly, to define access control policies which are responsive in real-world settings.

Application developers are provided with an application program library of generic session management services. Activity-information is used as the foundation and contains the following details: the users, the applications or tasks they are currently

engaged in and the objects in those tasks. Activity-information is thought of as the tuple:

Activity = (User, Task, Object)

When two activity tuples exist which contain the same object token, then the session management service can take action to allow the users to enter into a collaborative situation. For example, when two users edit the same file, the act of both editing the same document allows them to enter a collaborative activity. The libraries offer mechanisms for policy controls to allow users to alter the behaviour of the session management service so that users are not automatically thrown into a shared editor when they open a file that another user has opened. The activity information supports user and group awareness and stores information that is potentially useful to applications.

Edwards specifies access control policies for a group of users by associating access rights with an object and a task, defining roles and mapping policies to roles. Access rights have the value: read, write, remove, exist or none. For example, consider a policy where laboratory users have restricted access to group awareness information. Firstly, a restricted policy is defined by specifying the awareness information object with the access right set to exist, allowing the existence of the information to be tested, and the task with the access right set to none, denying all access. Secondly, the restricted policy is mapped to a role that encompasses all laboratory users. Roles can be defined dynamically enabling policies to respond to different contexts.

Edwards provides a partial description of group-work support. There is no discussion of data sharing and the description of session management and access control does not discuss a generic set of functions.

2.2.2.2 Discussion of models

Although these mechanisms contribute to characterising systems, they only partially support particular aspects of collaboration. A mechanism is needed to fully describe the functionality that a group of users might be offered in order to collaborate using a system.

2.2.3. The approach adopted

The approach adopted in this thesis is to develop a mechanism to analyse and compare systems, called the Framework. The Framework characterises the functionality that systems can offer users and is used as a tool with which to carry out in-depth analyses of systems. The distinctive qualities of such systems can then be

used to compare systems. The Framework focuses on the generic aspects of user functionality supported by collaborative systems. The Framework is developed using entity relationship modelling and state transition techniques, and as a result, the Framework is viewed as a complete and consistent description of collaboration within its scope. Chapter 3 describes the Framework.

2.3. Existing systems

It is necessary to demonstrate that the Framework is a useful tool for analysing and comparing systems. If the Framework can characterise systems and enable those characteristics to be used to compare systems, then the Framework is considered to be a useful analysis tool. This part of the literature review examines which existing systems might prove a suitable test vehicle for the Framework.

It is necessary to ensure that the selected systems are representative of CSCW systems and support the breadth of functionality available in CSCW systems within the scope outlined in section 2.2.1.6. Seven systems are chosen: Rendezvous, GroupKit, Dialogo, Suite, SEPIA, MEAD and SOL. All these systems support the construction of real-time applications in group-work. The systems also provide run-time environments for the applications and so the characteristics of the systems encompass the characteristics of the applications.

A brief explanation as to why each of these systems is chosen is given below:

• Rendezvous is one of the most established and fully developed group-aware toolkits in CSCW. Its applications support direct manipulation graphical interfaces.

• GroupKit uses "open protocols" which resemble the use of transitions in the Framework. It supports the development of shared visual environments.

• Dialogo is a shared window system; that is, it takes single-user applications and makes them available to a group. Dialogo originates from historically one of the first shared window systems. Other shared window systems are mentioned both in the description and in the analysis of Dialogo.

• Suite makes a significant contribution to supporting interface "coupling"; that is, the sharing of presentation characteristics amongst users. It supports the development of editor-based applications, such as spreadsheets and form editors.

• SEPIA supports browsers interacting with a hypermedia-based data model, similar to rIBIS (Rein & Ellis, 1991) and the World-Wide Web.

• MEAD supports a shared information space which is applicable to command and control domains.

• SOL separates the user interface components from the application functionality and allows the interface components to be tailored by each user.

The Framework is used to analyse the first five systems in depth and these systems provide the main test vehicle. The last two systems are analysed in outline essentially to ensure that their functionality is addressed by the Framework. The extent of the description of each system that follows, reflects the extent of analysis in Chapter 4.

2.3.1. Rendezvous

Rendezvous is a groupware toolkit which can be used to build multi-user, real-time applications with direct manipulation graphical interfaces. This section discusses some of its key features. The analysis is based on the more recent publications about Rendezvous and the progression of its development is not discussed. Publications that have contributed to this work include: Patterson et al. (1990), Patterson (1991), Brinck & Gomez (1992), Hill (1992), Brinck & Hill (1993), Hill et al. (1993) and Hill et al. (1994).

Rendezvous is a groupware *toolkit* and, therefore, the *group support functionality* is built into the toolkit which is then used to build group-aware applications. Rendezvous' scope includes the ability to build applications that support tightly-coupled work and is not intended to support loosely-coupled work. Further, the applications support direct manipulation graphical interfaces as opposed to, for example, form-based or text-based interfaces. Three examples of applications built using Rendezvous are Conversation Board which is a multi-person electronic whiteboard (Brinck & Gomez, 1992, Brinck & Hill, 1993), CardTable which is an electronic card table (Patterson, 1991) and a Tic-Tac-Toe game (Hill et al., 1994).

Applications built using Rendezvous are able to support the following user features: customised views, the ability to handle simultaneous input by multiple users, and the ability for users to join and leave running applications.

Five key technical features underlie Rendezvous: the notion of a dialogue independent structure, object-orientation, a hierarchical declarative 2D colour graphics system for generating and maintaining the user interface, an event-handling system for reacting to user input and a constraint maintenance system to help maintain object state. These technical features, along with session management, are discussed in greater detail below and it is shown how they deliver the user features.

The developers of Rendezvous have separated the user interface from the application functionality for its many engineering advantages and have attempted to resolve the resulting issues. Such issues include: supporting effective user feedback, knowing what properly belongs in the user interface and what belongs in the application functionality, and dealing with inconsistent information arising from redundancy. Three types of entity are introduced: abstraction, link and view. An abstraction is a model of the application. A view is what a user sees on a display and through which a user interacts with the application. A link binds an abstraction and a view together. Redundant information contained in both an abstraction and a view is kept consistent by constraints in the links. Constraints define relationships between properties of objects so that when a value of a property in one object is updated, the constraint automatically updates the value of the property in corresponding objects. For example, in Tic-Tac-Toe, the value of a cell is stored in both a view and in the shared abstraction. Each user has a view entity and there is one underlying abstraction entity, linked as shown in figure 2.6.

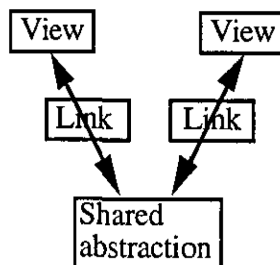


Figure 2.6 Basic Rendezvous Architecture

Although an abstraction is defined as a model of an application, it can also contain display information that is common to all users. The display information in the abstraction is linked to the display information in the multiple views and in this way, the information is shared. For example, if each user needed to have an object in the same place on their display, the position details would be contained in the abstraction and linked to the multiple views. Information that is not shared between users belongs in the views alone. Therefore, it is possible to have some aspects of the views shared

and other aspects customised. Hence, it is easily decided whether information belongs in a view or an abstraction.

To support WYSIWIS views, most of the display information would need to be contained in the abstraction and thus shared between users. This must be the case in Conversation Board. An alternative proposed in Patterson (1990) is to introduce a sharing object which contains the common display information and is linked to the multiple views, as shown in figure 2.7. In this case, the abstraction only need contain a model of the application and constraints between the sharing object and the views keep the consistency.

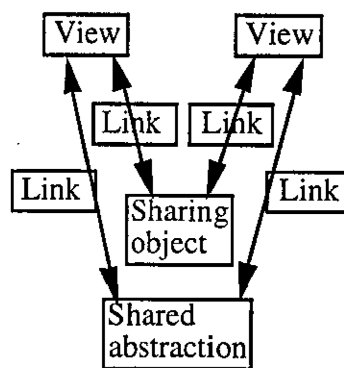


Figure 2.7 Rendezvous Sharing Object

Each view and each abstraction is a pre-emptively scheduled process. Since each user has their own view entity, Rendezvous can handle simultaneous input by multiple users. Interaction between the multiple views and the shared abstraction is serialised. This can lead to inconsistencies and has necessitated the development of an undo facility (Brinck & Hill, 1993).

User input can be controlled by using constraints. A view entity needs to include a precondition which references a guard variable. If the value of the guard variable is constrained to be equal to the value of some other variable, such as the view entity's process value, then the interaction is allowed; otherwise, it is not.

Rendezvous supports a hybrid architecture in which all the entities are contained in one process and located on one processor. In a multi-user application, many users require service at the same time, for example, to update their displays. There can be a lag in user feedback as a result of running on one processor. The need for Rendezvous to operate over multiple processors is recognised by the developers.

Rendezvous considers the interface as a collection of objects. The abstraction, link and view entities are structured as trees of composite objects. For example, the Tic-Tac-Toe board has a view and abstraction object connected by a link object. Both the view and abstraction board object have (at least) nine child objects, one for each cell, linked by nine link objects. The toolkit is based on Common Lisp and CLOS, which supports multiple inheritance. Rendezvous supports a set of base classes and a set of mix-in classes. It is recognised that the base classes give unneeded complexity and it would be better to reduce some of the inherited attributes to make them simpler to use.

Rendezvous includes a set of graphics primitives based on the X window system. A developer puts together a view to appear on the display based on these primitives and must indicate which objects are mouse sensitive. The graphics system automatically updates the display on re-draw and informs the objects about mouse activity. It supports a sprite mode for objects that move rapidly, such as a telepointer and objects that can be dragged. It does not enable a developer to use existing widgets provided by toolkits; widgets must be built from the graphics primitives. Further, the graphics measure is millimetres where X is pixel-oriented and so, for example, positions of objects relative to the display might vary between display types.

Constraints do not require a developer to indicate when information is conveyed. This has the advantage that the developer does not need to specify the procedural aspects of the communication but limits the range of possible designs. For example, more refined modes of interaction between the abstraction and view entities are difficult to implement, such as the sophisticated coupling mechanisms found in Suite. Further, more direct interaction, such as re-setting the application, which is easily implemented by an event, has to be done using constraints.

Patterson et al. (1990) describe how a Rendezvous application is started but indicate that the start-up architecture has not been incorporated into the Rendezvous toolkit. The start-up architecture decouples invoking and joining an application so that not all users need be known when the application is started and users have the ability to join and leave running applications. A Rendezvous session is a Rendezvous application. Users have access to a Rendezvous Access Point (RAP) from their display which enables them to start or join a session, refer to figure 2.8. The Rendezvous server is a name server as well as the parent process from which sessions are started. The RAP asks the server to start a session. Once a Rendezvous session is running it will register an address with the server indicating a communication channel that will be monitored

for incoming calls. RAP requests this information from the server and then calls the Rendezvous session. Upon establishing communication with the session, the RAP informs the session of the address of the user's display. The Rendezvous session can now connect to the user's display and the session is joined.

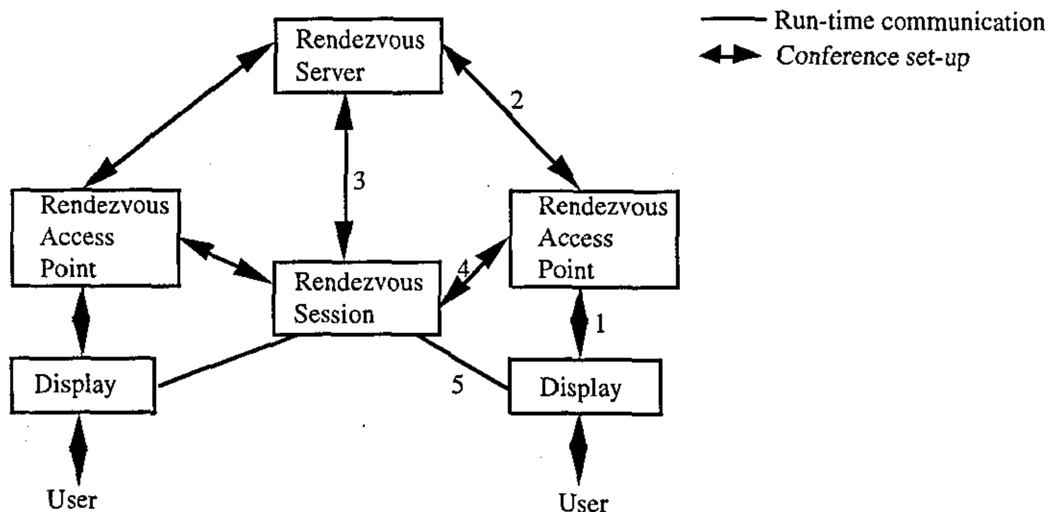


Figure 2.8 Rendezvous Start-up Architecture

Given a well-known address for each user's RAP, users can act as recipients of invitations to join ongoing Rendezvous sessions. In addition, the presence of the Rendezvous name server permits users to obtain a list of active sessions that allow anyone to join at any time. It is possible to have multiple sessions running concurrently.

2.3.2. GroupKit

GroupKit is a groupware toolkit for building real-time applications. In particular, the toolkit assists in constructing real-time work surfaces which are shared visual environments where one user's actions are immediately visible to other users. Work surfaces include generic applications such as shared windows, whiteboards, structured drawing systems and shared editors. GroupKit attempts to incorporate the design principles from human factors studies of face to face design teams into real-time applications (Roseman & Greenberg, 1996a), particularly the design criteria derived by Tang (1989). GroupKit is noted for its groupware widgets that support group awareness and its "open protocols" that support flexible session management.

GroupKit is the most recent product in a line of developments which reflect similar design aims. Previous products are important in that the publications emphasise

particular aspects which enable a fuller picture of the toolkit to be gained. The first development was Share, a shared window system which implemented personalisable floor control (Greenberg, 1991). This was followed by three group drawing tools: GroupSketch, XGroupSketch and GroupDraw (Greenberg et al., 1992). GroupSketch is a multi-user sketchpad that takes over the entire display and provides WYSIWIS views. Anyone can draw, type, erase, move their cursors around the drawing, and save or restore the image. XGroupSketch runs in the X window system and has a resizable, scrollable drawing surface. It also has greater functionality, such as colours and pen sizes. GroupDraw is an object-oriented drawing program in which users create objects that can be manipulated. (Roseman & Greenberg, 1992) (Roseman & Greenberg, 1993) (Greenberg & Marwood, 1994) (Gutwin et al., 1996) (Roseman & Greenberg, 1996a) (Roseman & Greenberg, 1996b) describe aspects of the groupware toolkit GroupKit.

GroupKit has four main features: firstly, a runtime infrastructure, which manages the creation, interconnection and communication of distributed processes, secondly, groupware programming abstractions, which give developers control over the behaviour of distributed processes, thirdly, groupware widgets, which provide additional interface features, and fourthly, session managers (Roseman & Greenberg, 1996a).

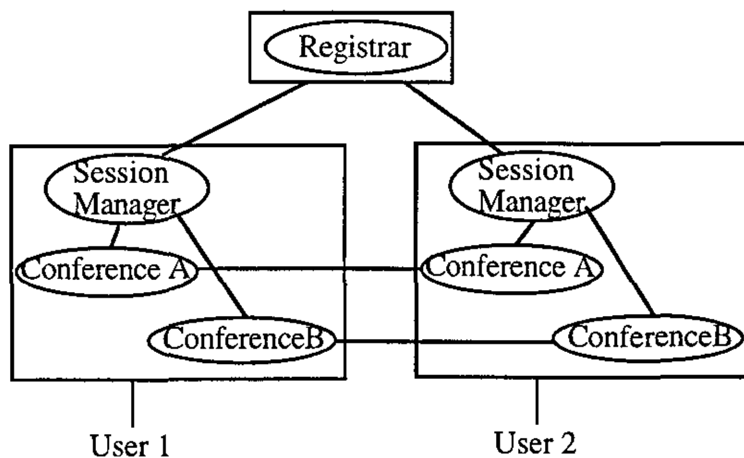


Figure 2.9 The GroupKit Runtime Infrastructure

GroupKit's runtime infrastructure consists of multiple processes distributed across different machines. Three types of GroupKit process can be identified: a registrar, session manager and conference application, as shown in figure 2.9. A registrar is a centralised process with a well-known address and is invoked first. The registrar maintains a list of all conferences and the users in each conference. A session

manager is a replicated process, one per participant, that lets users create, delete, monitor, join or leave conferences. It provides a user interface and a policy dictating the session management functionality. A conference application is invoked by the user through the session manager. It is created by an application developer and is typically replicated, one per participant.

GroupKit provides developers with three programming abstractions: multicast remote procedure calls, events and environments. An example of a multicast remote procedure call is "gk_toAll" which multicasts a procedure to all conference processes in a session. An event provides a way for a conference application to be notified when something happens. Programmers trap particular types of events through an event handler or binding. For example, the code "gk_bind newUserArrived action", traps a session event sent automatically by the runtime infrastructure and performs the specified action. The action in a text chat application is to provide a window for the new user. Another session event is used to handle latecomers and could result in updating the latecomer with the state of the application. Application developers can also generate their own custom events. Environments are data structures containing keys and associated values. Instances of environments running within different conference processes can communicate, resulting in shared data structures. For example, a user's environment can contain a record of the local user and the remote users. Environments can also act as active values. By combining sharing and event generation, a change in one process environment can change data in others and trigger corresponding actions at all sites. Active values enable quite different views to be generated from the same data abstraction, as in Rendezvous. No concurrency control is applied to environments by default, although an application developer can serialise all changes to an environment or can apply their own concurrency control policies (Greenberg & Marwood, 1994).

Groupware widgets support activities found only in group-work and include: participant status, telepointers and local awareness tools, such as, multi-user scrollbars and gestalt viewers (Gutwin et al., 1996).

Session managers are similar to conference applications. They communicate with a central registrar to manipulate lists of active conferences and their users. GroupKit stores this information in environments and changes to these environments generate events. Developers respond to events by providing their own bindings or by using GroupKit's default handlers.

GroupKit's session manager facilities are based on open protocols, a technique developed for building flexible groupware (Roseman & Greenberg, 1993). A central server, e.g. the registrar, provides a data structure, e.g. an environment, but specifies no policy for how the data structure is to be used. Clients to the server, e.g. the session managers, specify the policy by the selection of operations they perform. Maximum flexibility is achieved by providing open access to the server's data structure via a protocol of operations, such as "add_user" and "add_conference" , refer to figure 2.10. If an operation changes an environment , events are generated, such as, "foundNewUser" and "foundNewConference". Clients employ event handlers in different ways to implement the policy. In an open registration policy, the default event handler is used to respond to the event "foundNewConference", which adds a conference to the local environment and generates a new conference approved event. In a rooms-based policy, developers need to build their own event handlers (Roseman & Greenberg, 1996b). New clients, each with their own unique interface and behaviour, can be added at any time, as long as they use the defined protocol and are well-behaved with respect to each other and to the policy. Thus, the system can be adapted to the changing needs of particular user groups. The main difficulty seen with open protocols is the designer is required at the outset to decide the state changes that are reasonable to embed within the server and the protocol it should obey. The designer must predict the system's future use.

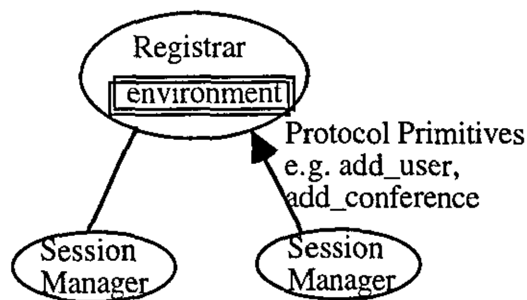


Figure 2.10 Open Protocols for Session Management in GroupKit

GroupKit and its applications run in a UNIX/X11 window environment, and use the Tcl language, Tk interface toolkit and Tcl-DP socket extensions. Application developers build applications using Tcl/Tk and the extensions provided by GroupKit.

2.3.3. Dialogo

Dialogo is a shared window system that enables existing single-user applications to be shared in a real-time conference. The re-use of existing software has two main benefits: firstly, financial, the application functionality has already been coded, and

secondly, personal, users are familiar with and have preferred software tools. The development of distributed window systems enable the connection between the application and the display to be tapped, so that input can be filtered and output multiplexed. A shared window system is distinguished from terminal sharing by supporting the sharing of sub-sets of applications with different users whilst retaining private access to other applications. The precursor to Dialogo is VConf, one of the first shared window systems (Lantz, 1987)(Lantz, 1988). Publications that contribute to this section include: Lauwers (1990), Lauwers and Lantz (1990), and Lauwers et al. (1990).

Many other shared window systems have been constructed, including: Rapport (Ensor et al., 1988)(Ahuja et al., 1990), SharedX (Gust, 1988), MMConf (Crowley et al., 1990), Share (Greenberg, 1990), ShX (Altenhofen et al., 1990) and Matrix (Jeffay et al., 1992). Since Dialogo is the only shared window system to be discussed in detail, this section focuses on the overall features of a shared window system, using Dialogo and other systems to exemplify some of the features. Overall features of shared window systems include: the architecture, multiplexing application output, floor control, session management, workspace management and support for pointing and annotations. Each of these is discussed.

Architecture

There are two main issues: firstly, which part of the window system is best modified, and secondly, to what degree need components in the architecture be replicated. Both are discussed.

There are three possible modifications to a window system. Firstly and most commonly, a pseudo-server is introduced between the applications and the actual display server, as in Dialogo, Rapport and Matrix, and shown in figure 2.11. The pseudo-server maintains the state of the virtual display and filters input and output between the client and multiple displays. Dialogo and Matrix create a virtual X server within the actual X server, so that users may even run and share a window manager in their pseudo-server as well as their normal applications. However, the pseudo-server may be required to store much of the application state and there may not be enough information in the core protocol to map data such as images across heterogeneous display servers. Secondly, display servers are modified to enable them to communicate, as in SharedX. This approach provides full access to the server state and a separate window sharing protocol extension to be used without altering the core graphics protocol. However, a display server is the most device dependent component of a window system and access to its source code may not be available. Thirdly, the

client's toolkit library is modified, as in ShX and MMConf. Client applications must be re-linked and might have to store much of the application state.

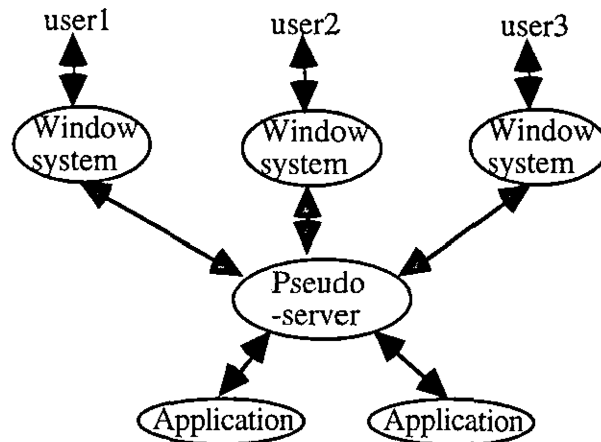


Figure 2.11 A Centralised Architecture with a Pseudo-server

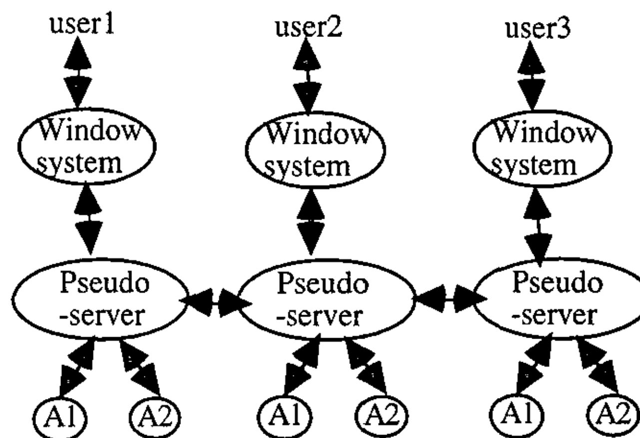


Figure 2.12 A Replicated Architecture

Figure 2.11 also represents a centralised architecture in which there is one instance of each application and one instance of the pseudo-server. Figure 2.12 presents a fully replicated architecture where there are as many instances of the application and the pseudo-server as there are participants in the conference. VConf, Dialogo, MMConf, ShX and Matrix adopt a replicated architecture. The main advantage of replication is improved performance, that is, superior response time and reduced network load. Only user input needs to be distributed; application output, which takes the greater bandwidth, is delivered locally. The disadvantage is that all replicas of the application must be kept in synchronisation. Lauwers et al. (1990) document the synchronisation problems associated with application replication in shared window systems, the aim being to find out how far replication can succeed without modifying existing

software. The most frequent synchronisation problems can be resolved but others require the applications and system servers to be made collaboration-aware.

Dialogo is built in a UNIX/X11 window environment and uses UNIX sockets to communicate.

Multiplexing application output

A primary function of a shared window system is to multiplex output from applications onto users' displays which provides users with shared views (refer to workspace management). Shared window systems are based on the assumption that multi-user interfaces can be modelled as a collection of single-user interfaces and, as a result, do not support different views.

Floor control

Another primary function of a shared window system is demultiplexing input streams from all users into a single input stream directed at the appropriate application which enables all users to interact with it. Floor control is where user input is processed according to whether or not a user is authorised to generate input. Lauwers (1990) identifies three characteristic issues of floor control:

1. the number of floors; one for the entire conference, one per shared application, or one per window.
2. the number of people who can hold a floor at the same time.
3. how the floor is passed between floor holders. This includes the potential use of auxiliary communication channels, e.g. audio.

Shared window systems commonly have at most one floor per shared application and one floor holder at a time. Thus, users are either restricted to following the current view or allowed to do everything. A companion set of mechanisms is needed for changing the floor.

Typical floor policies include:

- *free floor* enables any number of users to enter input at the same time.
- *pre-emptive* allows only one floor holder and any user can be the floor holder at any time.
- *explicit request* allows only one floor holder at a time and any user can request the floor, which is automatically granted.
- *first come* allows only one floor holder at a time, any user can request the floor, the request is queued and the floor passed to the next user in the queue.
- *baton or ring-passing* allows only one floor holder at a time and the floor holder passes the floor onto another user.

- *round-robin* allows only one floor holder at a time and the floor is passed to each user in turn.
- *moderated* allows only one floor holder at a time which is assigned by a moderator.

Some systems have a fixed policy, such as Dialogo, which has one floor for the entire conference (refer to workspace management) and supports the explicit request floor policy. Floor control functionality is distributed between the conference manager and the conferee's agents, refer to figure 2.13. Conferee's agents are equivalent to pseudoservers. The conferee's agents present a user interface that allows conference participants to request the floor. These floor requests are sent to the conference manager which implements the floor policy. Whenever the floor is passed, the conference manager informs the conferees' agents about the new floor holder.

However, the most suitable floor policy may vary with the current activity of the group. MMConf and Share separate the floor control mechanisms from the policy to allow the most suitable policy to be used in a given context and allow the group to change the policy dynamically. MMConf supports the policies: free floor, pre-emptive, explicit request and baton.

Session Management

Dialogo implements session management functions in a separate process, called the conference secretary, which displays a user interface allowing conference initiators to schedule and start conferences. When scheduling a new conference with the secretary, the initiator specifies the prospective participants and optionally lists a set of files that need to be imported into the conference. The conference secretary sends invitations to the prospective participants and keeps track of who accepts and who declines. Participants use the secretary's user interface to start a real-time session. When a session is started, the secretary starts a conference manager process to manage the real-time session and a conferee's agent process on each participant's workstation. Each conferee's agent creates a replica of a shared user interface shell as the first shared application in the conference. Other shared applications are created through the shared user interface shell. The resulting architecture is shown in figure 2.13. It is unclear whether the connection set-up module is a separate process but it creates a well-known socket for the conferee's agent. Conference applications are started with their display set to point to the conferee's agent's port so that communications are directed towards the conferee's agent rather than the display server. The conferee's agent opens a connection to the server, which has a well-known network port.

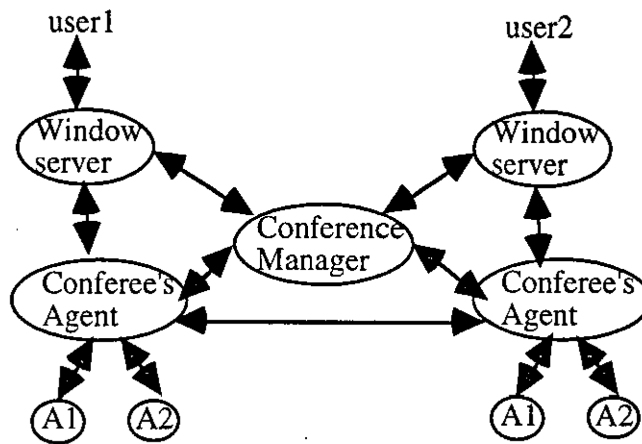


Figure 2.13 Architectural Overview of Dialogo

Dialogo does not support dynamic reconfiguration which is necessary to support the arrival of latecomers, the departure of participants prior to the termination of a conference, and the transfer of shared windows into and out of a conference. Dynamic reconfiguration requires a pseudo-server to be interposed dynamically in the existing communications link between the display and the application. SharedX supports latecomers and enables them to "catch up" by replicating the shared state on a latecomer's display. State transfer is commonly not supported by applications or window systems. Therefore, to implement state transfer, pseudo-servers maintain redundant copies of the shared window system state and the pseudo-server's state vector is projected onto the latecomer's window server. Matrix contains a filter which records every message an application sends to the display server that effects the visual interface.

Workspace management

This is concerned with window layout and size and determines the grouping of shared windows within a shared workspace. It is similar to a window manager for single-user applications but requires additional functionality, such as, distinguishing one conference's windows from another's and from private windows, and being able to manipulate these groups, such as raising to the surface of the display all windows that are associated with a particular conference. Dialogo and Rapport use the rooms metaphor to group shared windows in a conference (Henderson & Card, 1986). Strict WYSIWIS views are guaranteed by running an existing window manager as a shared application. However, this provides only one input focus over the shared workspace, allowing only one floor holder per conference, and therefore, inhibiting user input concurrency

Support for pointing and annotations

The potential to point to a shared object or make graphical annotations can greatly enhance the communication between participants (Tang & Leifer, 1988). Dialogo, SharedX, MMConf and Rapport include a telepointing facility that displays an arrow-shaped pointer on the shared workspace. In Dialogo, the telepointing application draws its pointer directly in the shared conference window by XORing and repeatedly drawing and erasing its tracks across the screen. This method can result in incorrect displays if application output is generated at the same time.

2.3.4. Suite

Suite is a user interface toolkit that supports the development of multi-user, editor-based applications. The novel aspect of Suite is its interaction model which is used for coupling the user interfaces of a multi-user application. Coupling is defined as "informing a user of an input value entered by another user" (Dewan & Choudhary, 1992). In later work, this definition is extended to include sharing the presentation of a data structure (Dewan & Choudhary, 1995) and sharing access to a data structure (Shen & Dewan, 1992). Coupling is considered to be a collaborative function necessary to support a multi-user interface. Editor-based applications include: spreadsheets, language-oriented editors, form editors and command interpreters.

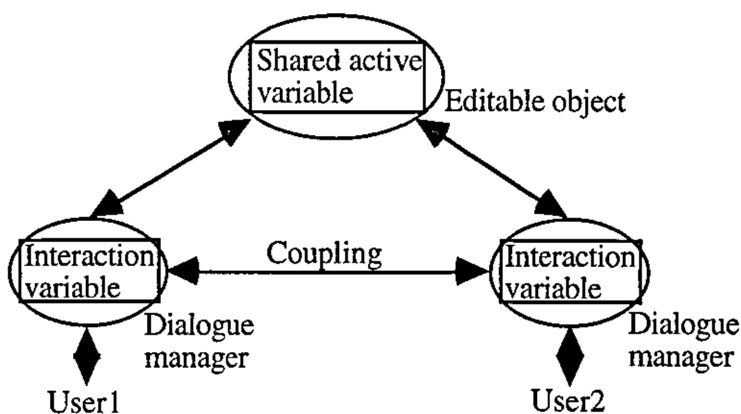


Figure 2.14 The Suite Architecture

Suite supports two main types of objects: shared active variables and interaction variables, refer to figure 2.14. Shared active variables are application variables whose value is displayed to users and can be changed via an interaction variable. Interaction variables are users' local versions of active variables which are created when a user connects to the application. An interaction variable has a set of properties which

determine, for example, its presentation. Another object discussed is a window object which displays interaction variables, menus, errors and attributes.

The Suite coupling model determines, firstly, which properties of interaction variables and window objects created for a user are shared with other users, and secondly, when changes made by a user to an object are communicated to others. The Suite coupling model does not require direct specification of the parameters of coupling between a pair of interaction entities. Instead, it requires the specification of "coupling attributes" which are associated with individual interaction entities. Each coupling attribute indicates a user's preference regarding some parameter between the interaction entities and the interaction entities created for a group of users. For instance, a user might specify the value "True" for the "ValueCoupled" attribute of a particular interaction variable which would indicate that the user would like to share the value of that variable with other users. Suite supports several criteria for choosing when a change made by a user to a shared value is communicated to other users. These criteria include structurally how complete the change is, how correct it is, and the time at which it is made. The coupling model itself can be extended.

The Suite framework provides a user interface for users and a procedural interface for programmers to specify the values of the coupling attributes. In order to make the specification process less laborious, Suite provides mechanisms for grouping the specifications, inheriting attributes, using default values and provides rules to resolve the differences in the specifications between individuals. The coupling attributes can be changed dynamically.

Mechanisms for grouping the specifications include coupling sets, value groups and user groups. A coupling set is a mechanism for grouping the properties of an interaction variable. For example, the coupling set associated with the ValueCoupled attribute contains the value of the variable and the properties, Uninitialised and Erroneous. Other coupling sets are associated with the attributes: ViewCoupled, FormatCoupled, AccessCoupled, SelectionCoupled, ScrollCoupled (whether the scrollbar is shared), ActionCoupled (whether actions performed in a window are shared), PositionCoupled (for windows) and SizeCoupled (for windows). A value group is a group of related interaction variables that stores attributes shared by these variables. A coupling attribute is specified for a particular group of users which can be system or user defined. For example, the value (True, suite_sub_group) can be specified for the ValueCoupled attribute where suite_sub_group is a group of users.

Suite supports an architecture with a central semantic component or editable object containing the shared active variables, and local user interface components or dialogue managers, refer to figure 2.14. A dialogue manager manages the interaction entities for a particular user. It operates in its own environment, is dynamically created, and is connected to and from the editable object by the user. A dialogue manager exchanges coupling and locking information with its peers by sending messages to the editable object which distributes them to other dialogue managers. Library routines linked to the editable object implement multiplexing and the distribution of messages among the dialogue managers. A group manager supports user and program defined groups.

Publications that have contributed to this work include: Dewan (1990), Dewan and Choudhary (1991a), Dewan and Choudhary (1991b), Dewan and Choudhary (1992), Dewan and Choudhary (1995), and Shen and Dewan (1992).

2.3.5. SEPIA

SEPIA aims to support the distributed creation of multi-author hypertext documents, and in particular, SEPIA focuses on supporting the creation of structures of hyperdocuments. SEPIA employs a hypermedia-based organisation model not only for designing the final hyperdocument, but for structuring and relating information as part of the authoring process. Users can assign a division of tasks based on SEPIA's decomposition and aggregation structures. Publications on SEPIA include: Haake & Wilson (1992), Streitz et al. (1992), Streitz (1994) and Streitz (1995).

SEPIA consists of three major parts: an object management system, a multi-user hyperdocument database and multiple activity spaces. The architecture of SEPIA is shown in figure 2.15. The object management system interacts with the multi-user hyperdocument database. Types of objects supported include: atomic nodes, composite nodes, labelled links and container objects. Composite nodes are organisations of atomic nodes and labelled links that enable clustering of related parts of a document. A container object contains the view information of a data object, such as, the position, icon, style and size. A data object can be connected with different container objects, and therefore, displayed differently in different contexts. Each object carries a lock attribute and each composite node carries a list of all users who have opened the node. Changes to objects are broadcast using a broadcast server.

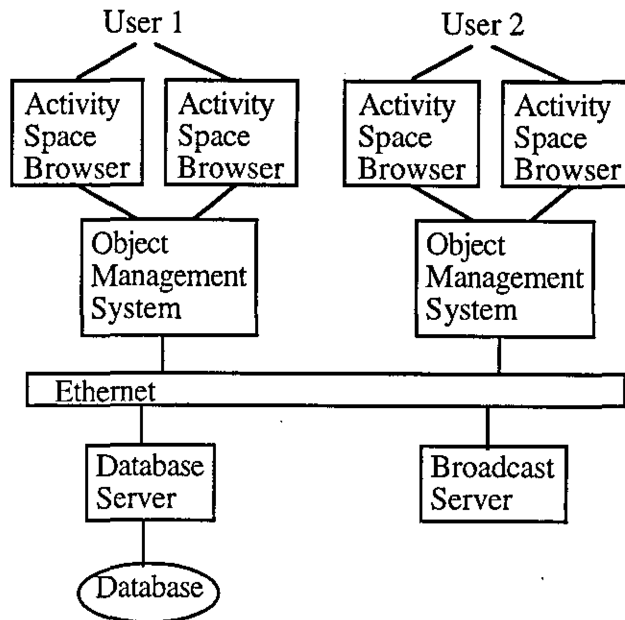


Figure 15 The SEPIA System Architecture

SEPIA supports activity spaces which can be viewed as task-specific browsers and which enable a user to take part in the creation process of hyperdocuments. SEPIA supports one scrollable browser per activity space, which is used to display the contents of a composite node, and since there can be multiple activity spaces, SEPIA supports multiple browsers on different composite nodes.

SEPIA addresses a requirement made by Grudin (1989) that cooperative work should not interfere with the productivity of individual work. This is interpreted to mean cooperative work should not be intrusive and it should be possible to move easily between individual and cooperative work. Three distinct modes of collaboration are identified: individual, loosely-coupled and tightly-coupled work. Individual mode is working on any node not visited by other authors. Loosely-coupled mode is working on the same composite node as another. Tightly-coupled mode is sharing the same view of a composite node as another. Each mode provides different levels of awareness of other users' activities. SEPIA supports all three modes and the easy transition between them. In all modes, authors can access and modify concurrently shared hyperdocuments.

Users are aware of other users working on the same composite node by two mechanisms. Firstly, a status line on a browser indicates other users working on the same composite node and whether they are working in loosely-coupled or tightly-coupled mode. Secondly, an object must be selected before an operation can be

performed on it. On selection, an object is locked and displayed in yellow on the authors display and red on other users' displays.

Tightly-coupled mode adds another three mechanisms for awareness: shared views, telepointing and an audio connection. In shared views, the size of the browser and the scroll position are shared. Each user is given a telepointer which is displayed to others when a mouse button is depressed. Each user is able to perform their own operations independently and concurrently, except for scrolling, re-sizing the browser and selecting objects which are already locked.

SEPIA automatically sets up a loosely-coupled session amongst the users who are working on the same composite node. A user in a loosely-coupled session can request a tightly-coupled session. The user is presented with a menu of names of users in the loosely-coupled session from which users for the tightly-coupled session can be selected and invited to join. Users cannot join a tightly-coupled session later. It is possible to have multiple tightly-coupled sessions on the same composite node by different groups of authors. Each author can participate in at most one tightly-coupled session per composite node. Therefore, it is possible that several tightly-coupled sessions together with some users in loosely-coupled mode are simultaneously working on the same composite node.

WSCRAWL is a shared drawing tool which is integrated with SEPIA to enable authors to share the view and jointly edit the contents of documents.

Hypertext editor rIBIS is similar to SEPIA (Rein & Ellis, 1991). rIBIS users can switch between a loosely-coupled session and a tightly-coupled session in a hypertext network. In a tightly-coupled session, users take turns using a floor-controlled group pointer for all mouse-based activities.

2.3.6. MEAD

MEAD is a multi-user interface prototyping environment which focuses on supporting the development of tailorable cooperative interfaces (Bentley et al., 1992) (Bentley et al., 1994). MEAD is applicable to command and control domains where a shared information space forms the focus of the work taking place, e.g. air traffic control. The shared information space reflects the external processes being controlled, e.g. the aeroplanes. Tailoring the filtering of the information space and the display composition are considered important requirements in these domains. Figure 2.16 shows the MEAD architecture.

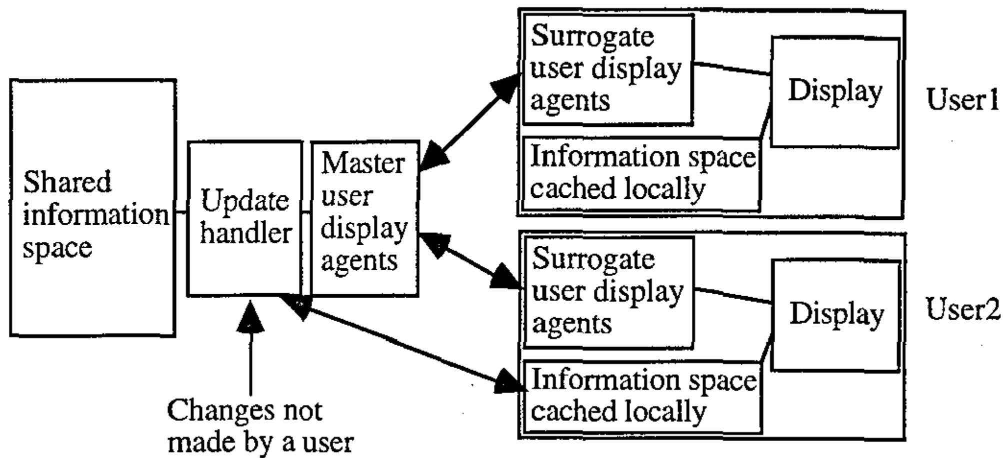


Figure 2.16 The MEAD Architecture

Users' displays are considered as autonomous entities, or user display agents, with properties that can be tailored dynamically by interface developers and users. The states of these agents characterise the information presented to users who interact directly with the agents and update the underlying information. An agent's properties are defined by the following criteria: selection criteria, which are the criteria used to dynamically choose the set of information entities from the information space, e.g. aeroplanes in a particular zone, presentation criteria, which are the views chosen to represent the information entities, and composition criteria, which are the positions of the views on the display, e.g. correlated with the location of the planes.

A user display agent can be a member of more than one user's working set, and thus support WYSIWIS views. Each screen representation managed by such an agent is effected by updates to the shared information in the same way. A surrogate of each user display agent is held locally. A master user display agent uses the definition criteria to compute the effects of relevant updates to information entities and informs each surrogate of the new state. This master/ surrogate arrangement of agents allows local tailoring of information displays.

All updates to shared information objects are delivered to an update handler which forwards updates to the information space, as well as notifying potentially effected user display agents. To minimise communication between the information space and the remote displays, shared information objects are cloned and held in a local cache.

In summary, MEAD supports user display agents which both filter the information space for selected entities and specify the form the entities take when presented to

users. MEAD supports the tailoring of the selection, presentation and composition criteria for each display agent and, thus, supports tailoring for users.

2.3.7. SOL

SOL is a user interface toolkit which can be configured to support multi-user interfaces (Smith & Rodden, 1995). It aims to separate the application semantics from features of cooperative use in order to identify re-usable and generalisable interface components. Figure 2.17 shows the SOL architecture. SOL contains shared user interface objects which are collaboration-aware. The configuration of these objects can be divided into the configuration of their presentation and use, and the configuration of their response to user input.

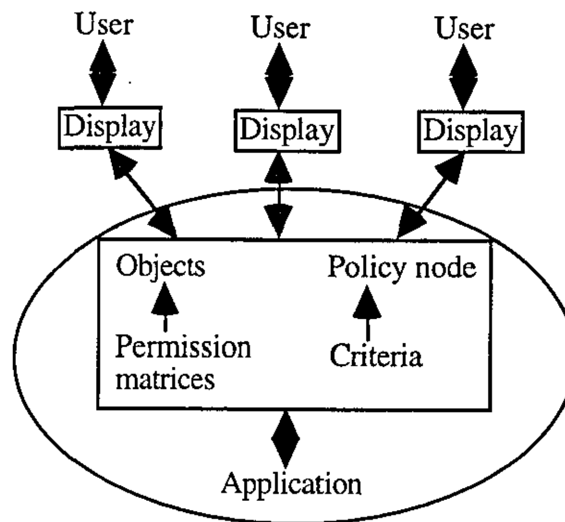


Figure 2.17 The SOL Architecture

The configuration of individual user interfaces within a multi-user application are specified in the SOL environment using an access control mechanism. The shared objects are assigned methods. For example, the methods associated with a push button are visible, usable, movable and resizable. Orthogonal to these methods are domains which define the context in which the method is to act. The domains chosen in SOL are "use", which defines whether a user can invoke a method, and "sharing", which defines whether the result of a user's invocation of a method is shared. A permission matrix is specified for each object and user, and kept with the object. For example, a permission matrix for a push button will contain eight flags: four specifying if a user may see, use, move and resize the button, and four specifying whether the effect of seeing, pressing, moving or resizing the button should be broadcast to interested parties. Two types of flag in the matrix indicate who may edit

the access permissions, an end-user or a system administrator. When a method is specified as "sharing", all users who have configured their interface object to sharing that method, see the effect of that method. If a user has chosen not to share a method, the effect of that method is local to that user's display. This allows users to control the coupling of their interfaces.

Permissions can be grouped in order to reduce the amount of specification required. Roles encompass a number of users. The user interface hierarchy is used to group objects so that child objects can inherit from parent objects. Objects can be given symbolic names and the name used within the access mechanism.

The configuration of an object's response to user input is controlled by the policy node which receives all input events from the different displays of an object and produces an output token. Criteria in the policy node allow objects to react to different patterns of input in different ways. The criteria are in the form of rules: if <input=context> then <token>. A token can be SOL specific, in which case a SOL object is commanded to do something, or application dependent, whereby a token is passed to the callback code and the application decides the nature of the cooperation. For example, a criteria might be: if user=fred then solo_override, where the token solo_override tells a SOL object to ignore all locks. SOL supports two types of context: selective and consensus. A selective context applies for a user interacting with an object. A consensus context applies for a number of users interacting with a specialised button.

SOL applications have access to a user profile for each user permitted to use the system. The policy node may use any of the resources used within a user's profile in the input context of a criteria. For example, if a user is assigned the role of examiner, a rule in the criteria might read: if profile.activity=examiner then examiner. The user profile provides a single point of amendment for the whole system.

In summary, SOL uses an access control mechanism to couple the presentation, use and access to interface objects. It generalises the coupling used in Suite to a wider range of applications but does not couple the presentation of objects in the same detail as Suite.

2.4. Summary

This chapter began with an overview of a range of categorisations that have been used to examine systems in CSCW. It described the limitations of the categorisation approach for the purpose of analysing systems. It suggested developing characteristics of systems and outlined the partial descriptions of collaboration offered by existing tools that take this approach. The Framework aims to fully characterise collaboration.

The second part of the chapter identified seven existing systems that are representative of systems within the scope of this work. The seven systems were selected for analysis by the Framework and were described in detail. The systems are: Rendezvous, GroupKit, Dialogo, Suite, SEPIA, MEAD and SOL.

Chapter 3 describes the Framework. Chapter 4 uses the Framework to characterise the seven systems. Chapter 5 uses the characteristics to compare the systems.

Chapter 3:

A Framework of Collaboration

3.1. Introduction

This chapter describes a framework of collaboration. The description identifies the important entities in group-work support by employing the entity relationship modelling technique (Chen, 1976). It is possible to identify the possible states of the entities described in the entity relationship models and to describe the transitions between the states using state transition diagrams. The valid transitions define a set of primitives which a system providing group-work support might support.

Entities are used first of all to describe the Framework in order to properly describe the components of the Framework. Although the notation for entity relationship modelling has been used to focus attention on the relevant aspects of the Framework, it is acknowledged that these entities have properties that make them equivalent to objects. Each entity used in the Framework may be instantiated in many different ways and, as such, the entities used are the overall classes which would be found in any framework of collaboration. It is not necessary to add the property of encapsulation in the description, however it is acknowledged that any or all of the entities used to describe the Framework will generally exhibit encapsulation.

Entity Relationship modelling is used and enhanced towards Object Oriented Design (Cox, 1986, Booch, 1986) in such a way that the simplicity and clarity associated with entity relationship modelling is preserved. The properties useful to the description of collaboration are introduced as required. Object oriented modelling languages, such as OMT, OOSE, Booch and UML, could have been used. The main disadvantage is having to adopt an associated methodology which is unnecessary and complicates the description.

As a collaboration progresses, the organisation of the entities changes and as such the relationships between the entities change. The overall entity representing the evolving system, primarily comprising users and applications, changes by reorganising the entities that are present in the system and redefining the relationships between the entities. The analysis seeks to present this change in organisation as a primary goal. The system therefore changes state by reorganisation, by introducing new entity instances, such as a new user, deleting entity instances and changing the states and properties of the participating entities. State transition diagrams are a proven useful way of representing changes in objects and so have been used here.

For the purpose of developing a framework of collaboration, a group is defined as several users (one or more) working on a common task. The task might be supported by one or more computer-based tools (or applications), as shown in figure 3.1. In order to use an application in a group, it is necessary to provide the functionality for a group of users to access the application. Such functionality could include the ability for users to view the output of the application and to concurrently interact with the application or to use a different application. This functionality is collectively called group-work support.

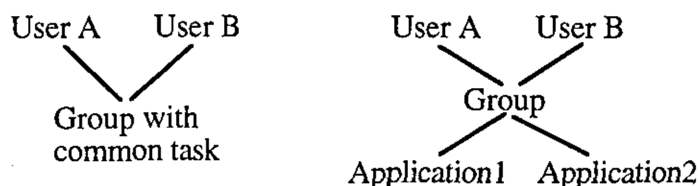


Figure 3.1 A Group

The Framework describes the functionality in group-work support. It does not describe how this functionality is presented to users because this is outside the scope of this work. For example, the Framework identifies a function which enables a user to join a group, but whether this requires a command or the movement of an icon representing a user from one window to another, as in a rooms-based metaphor, is not specified.

Group-work support can be considered separately to application functionality. Application functionality might provide task-related functionality, a shared space, such as audio/ video communications links between users, or a group decision-making tool, such as gIBIS (Conklin & Begeman, 1988). In the latter case, it is not easy to distinguish between group-work support and application functionality. A

group decision-making tool still needs group-work support in order to operate in a group, but its functionality provides additional mechanisms for group-work.

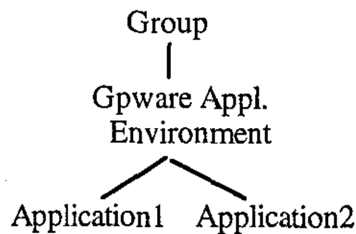


Figure 3.2 A Groupware Application Environment

Group-work support might assume different forms. Group-work support has been embedded within the application itself, such as RTCAL (Sarin & Greif, 1988). More recently, groupware toolkits have been developed which provide the group-work support and from which applications can be built, such toolkits as Rendezvous (Patterson et al., 1990). An application that embeds group-work support is referred to by terms such as "shared application" or "group-aware application" or "collaboration-aware application". Shared window systems enable existing single-user applications to operate in a shared environment by enhancing the window system with group-work support, such as Dialogo (Lauwers, 1990). Groupware application environments have been built that provide the group-work support for multiple single-user applications, such as MUMS (Jones & Edmonds, 1994), as shown in figure 3.2. The term "collaboration-transparent application" is used to refer to an application that is single-user and does not provide group-work support.

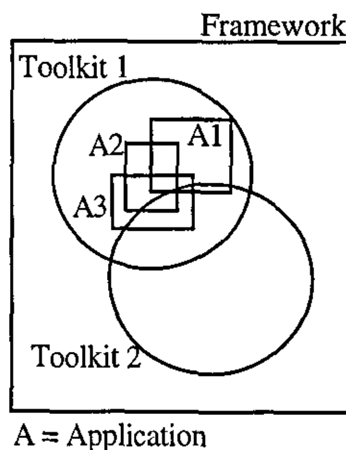


Figure 3.3 Functionality Space

The Framework offers a complete description of the functionality in group-work support within its scope. Commonly, particular systems, be they applications, toolkits or environments, implement a subset of the functionality described by the Framework. Further, applications built using groupware toolkits implement a subset of the functionality offered by the toolkit. Thus, the functionality space can be described as shown in figure 3.3.

Just because a toolkit can offer only a subset of the functionality does not mean it is not sufficient. For example, groupware toolkits might be tailored to encompass only the support needed by a particular category of application or task. A broad category frequently encountered in the literature is the distinction between loosely-coupled and tightly-coupled tasks. A loosely-coupled task is where users are focused on the same aspect of the task at different times or different aspects of the task at the same time. A tightly-coupled task is where users are focused on the same aspect of the task at the same time. Chapter 4 describes particular systems which provide various sub-sets of the functionality.

The term "group-work support" is intended to encompass any functionality needed to use an application in a group. However, to be clear, the term needs to be distinguished from other terms, such as conference support and shared application support. Conference support usually refers to support for real-time application sharing with audio/ video communications links between users, e.g. Rapport (Ensor et al. 1988). In this thesis, group-work support encompasses the real-time application sharing but views the audio/ video links as a separate application. Shared application support usually refers to supporting the sharing of one tightly-coupled application, e.g. a card-game (Patterson, 1991). In this thesis, group-work support encompasses tightly-coupled and loosely-coupled sharing.

This chapter is divided into five sections. The first three sections discuss three broad categories of functionality in group-work support: session management, data sharing and concurrency control. Within each category, a model is developed which includes a description of the entities and primitives. The penultimate section describes the integrated model. The final section draws the conclusions from this chapter.

3.2. Session Management

Session management commonly refers to the management of users and applications involved in a group activity which is carried out in real-time. For example, it includes functionality that enables users to join in on a group activity.

3.2.1. Entity relationship model

Five entities have been identified in session management. Four of these entities are: **user, group, application instance** and **persistent data**. A group is defined as several users working on a common task. The task might be supported by one or more applications. An application instance refers to a particular running application. An application is assumed to encompass both group-work support and application functionality. Applications used by users out of the group influence this work but are not considered part of it, except when they are brought to the task. Persistent data refers to stored data and commonly exists as system files. Persistent data can be used by one or more application instances. Thus, an entity relationship model of session management might be described as shown in figure 3.4.

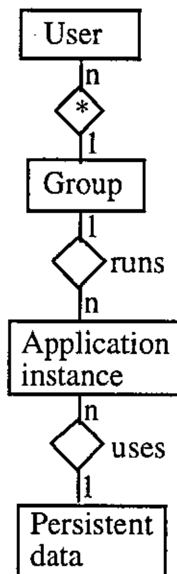


Figure 3.4 Preliminary Session Management Model

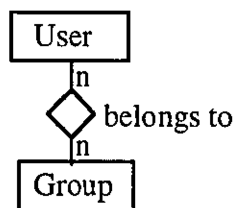


Figure 3.5 Multiple Groups

A user might participate in several tasks at the same time and thus belong to several groups, as shown in figure 3.5. The entity **membership ticket** emerges from the entity relationship modelling technique in order to resolve potential problems as a

result of multiple users belonging to multiple groups. A membership ticket defines a link between a specific user and a specific group. It needs both a user and a group in order to exist. Thus, an entity relationship model of session management is re-described in figure 3.6.

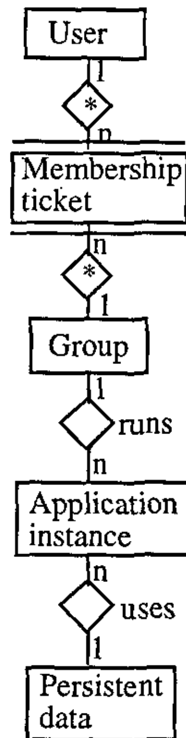


Figure 3.6 Session Management Model

The membership ticket entity enables a group to know which user is interacting with it and enables a user to know which group it is interacting with. A group passes on knowledge of the user to the applications which provides invaluable information if the applications are collaboration-aware and need to discern between different user input and output. More generally, an application needs to know who the users are in a group in order to distribute application output to their displays. A user needs to know which applications belong to a group and, if an application has multiple windows, which windows belong to an application and a group. This brings in issues concerned with workspace management, such as the positioning of windows on the display to bring them together so that applications in a particular group are apparent to users.

Collectively, membership tickets contain knowledge about users belonging to groups. This information can be made available to users in various forms to support user awareness. For example, a group could be informed of other members' presence and

the groups that other members belong to. Alternatively, prospective members could be informed of who is in a group before deciding whether to participate or not.

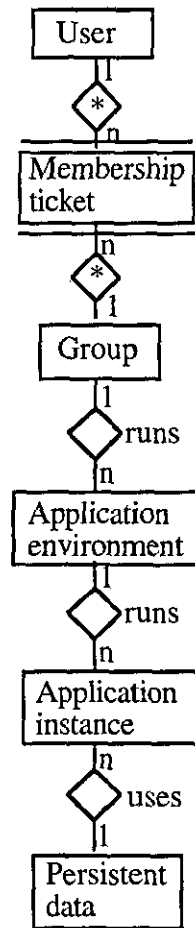


Figure 3.7 Session Management Model with an Application Environment

If the group-work support is embedded in the applications, the model shown in figure 3.6 suffices. However, if the applications are collaboration-transparent and the group-work support is embedded in a separate entity, such as an application environment or a shared window system, the model needs to be extended. The entity **groupware application environment** is introduced. An groupware application environment supports run-time services which provide the group-work support functionality that enables single-user applications to be available to a group. In this case, an application encompasses application functionality but not group-work support functionality. An groupware application environment is used by a group in order to support the common task, and potentially, can run more than one application instance. Since potentially multiple users can belong to multiple groups which are supported by multiple groupware application environments respectively, the membership ticket

entity is still needed. An entity relationship model of session management with a groupware application environment is shown in figure 3.7.

A model can be considered in part. For example, if the user, group and application instance entities are considered important, the model can be described as shown in figure 3.8. Since the membership ticket entity is removed, the relation between the user and the group entities changes. Multiple users can belong to multiple groups.

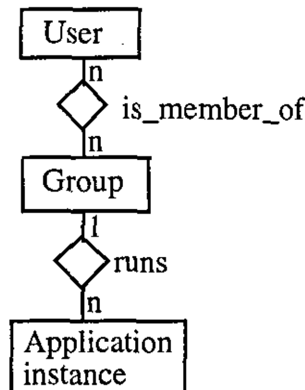


Figure 3.8 Model with Entities: User, Group and Application instance

A single user is a special case of the model in which there is only one member of the group and therefore only one membership ticket exists. The user can run multiple applications.

3.2.2. State transition diagrams

Initially, in order to model session management functions, such as, joining a conference, three entities are considered: user, group and application instance. The entity relationship model for these entities is shown in figure 3.8. Each entity is considered to have two states: either a user (u) exists or not, either a group (g) is formed or not, and either an application instance (ai) is running or not. If two or more entities exist, they are related in some way and considered to constitute one state. The possible states of these three entities are identified below. For example, consider case 7 where a user and a group exist but an application instance does not exist. As a result of them both existing, case 7 identifies the state where a user is a member of a group. This is represented diagramatically as one state by both entities existing and being linked. If they are not linked, a user would exist (as in case 5) and a group would exist (as in case 3) and thus two independent states would be represented.

States

	User	Group	Application Instance	
1.	x	x	x	null case
2.	x	x	√	[ai]
3.	x	√	x	[g]
4.	x	√	√	[g]-[ai]
5.	√	x	x	[u]
6.	√	x	√	[u]-[ai]
7.	√	√	x	[u]-[g]
8.	√	√	√	[u]-[g]-[ai]

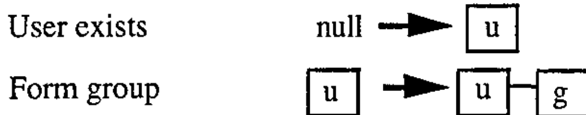
Basic transitions

Start application	null → [ai]
Stop application	null ← [ai]
Form group	null → [g]
Dismantle group	null ← [g]
User exists	null → [u]
User does not exist	null ← [u]
Add user	[u] [g] → [u]-[g] ; [u] [g]-[ai] → [u]-[g]-[ai] ; [u]-[ai] [g] → [u]-[g]-[ai]
Remove user	[u] [g] ← [u]-[g] ; [u] [g]-[ai] ← [u]-[g]-[ai] ; [u]-[ai] [g] ← [u]-[g]-[ai]
Add application	[ai] [g] → [g]-[ai] ; [ai] [u]-[g] → [u]-[g]-[ai]
Remove applic	[ai] [g] ← [g]-[ai] ; [ai] [u]-[g] ← [u]-[g]-[ai]

This section on basic transitions examines the possible transitions between the states and combinations of the states described in the previous section. The transitions listed are the minimum basic set and a sub-set of the possible transitions. The other

transitions have been eliminated because they can be created from a combination of these basic transitions.

It is easy to understand how an application is started, but the formation of a group and the existence of a user is less clear. To reiterate, a group is defined as several users working on a common task. In this thesis, a group entity is considered as having any number of users. Therefore, a group entity can be formed and have no users. In systems, this might be realised by specifying a group of users before any have actually joined the group. An alternative viewpoint would be to consider a group entity as having at least one member. This requires a user to exist and a user to form a group.

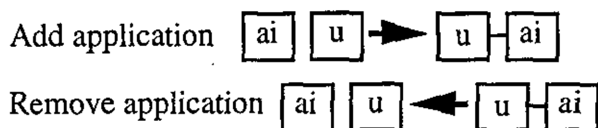


This viewpoint does not distinguish between forming a group and adding a user, and therefore can be described as a combination of transitions of the first case. Further, a group entity containing no users is considered to be able to run an application. In systems, this is realised by an application running in background, waiting for users to join the group. An alternative viewpoint would be to restrict a group to having at least one member in order to run an application. It requires the transition "add user" before "add application", and thus, is a specialisation of the first case.

A user entity that exists in this context is a particular individual who is related to the group, such as, a user who wishes to join the group or leave the group. In the transitions "add user" and "remove user", it is necessary to have a user entity. In the transitions "add application" and "remove application", it is not necessary to have a user entity and if one exists in the group it is of no consequence.

The transitions "add user" and "remove user" encompass the two transitions defined as "create membership ticket" and "delete membership ticket" respectively. When a user is added to a group, a membership ticket is created which specifies the link between the user and the group. When a user is removed from a group, the membership ticket is deleted.

A third and valid case of the transition "add application" is shown below. An individual runs an application but the application is not shared by a group. As mentioned earlier, although this transition influences this work, it is not considered part of it.



The basic transitions have associated parameters, such as, identifiers, actors and states. The complete list of session management transitions with their associated parameters are detailed below but first some explanations are needed.

Each transition affects an entity and therefore has an entity identifier and an actor as parameters associated with it. Possible actors include: **member**, **system** and **user**. A member is a user who has been added to a group. The system is a process or a collection of processes supporting a session management transition (the system only affects applications). A user is someone who uses the system and who has not been added to the group. An actor might adopt a particular role, such as, initiator or terminator. An *initiator* sets up the group arrangement and a *terminator* ends it. Certain actors and roles are appropriate for certain transitions as detailed below.

Certain transitions have entity states as parameters to describe support for both suspended sessions and application transfer between an individual and the group. Support for suspended sessions and application transfer are considered in turn. A session might need to be suspended after running for a long time, or because of an interruption, and the session re-started at a later point. To suspend a session, it is necessary to store the state of a group, which could include storing information such as the group identifier and the connected user identifiers. Further, it is necessary to store the state of any running applications in the group. This requires that the applications incorporate functionality that enables their state to be suspended and re-started which, in practice, very few applications support.

Several of the session management transitions require the transfer of application state between an individual and a group. The transitions are: the second case of "add user", where a user is added to a group which is running an application; the third case of "add user", where a user running an application is added to a group; and the third case of "remove user", where a user takes an application with her when removed from a group. These transitions are only possible if an application incorporates state transfer functionality. These cases are discussed further in the section on data sharing.

The basic transitions listed before have associated parameters, such as identifiers, actors and states. The basic session management transitions and their parameters are listed below:

• Start application <appl id> <actor>, and <appl state> if the application has been suspended, where actor is a member, the system (a process set to start) or a user.

• Stop application <appl id> <actor>, and <appl state> if the application is suspended, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

• Form group <group id> <actor>, and <group state> if the group has been suspended, where actor is a user.

• Dismantle group <group id> <actor>, and <group state> if the group is suspended, where actor is a member or a user.

• User exists <user id> <actor>, where actor is a member or a user.

• User does not exist <user id> <actor>, where actor is a member or a user.

• Add user <user id> <group id> <actor>, and <appl state> if user brings an application instance to the group, where actor is a member, the system or a user. If the actor is a member or a different user to the one being added, the user is said "to connect" to the group. If the actor is the user being added, the user is said "to join".

• Remove user <user id> <group id> <actor>, and <appl state> if user takes an application instance from the group, where actor is a member, the system or a user. If the actor is a member with the same identifier as the user being removed, the user is said "to leave" the group. If the actor is a different member or a user, the user is said to "disconnect". If the actor is the terminator, the group is being dismantled.

• Add application <appl id> <group id> <actor>, where actor is a member, the system or a user.

• Remove application <appl id> <group id> <actor>, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

The basic transitions offer a complete description of session management functionality. However, in a system, a basic transition might be embellished, subsumed in another transition and not supported independently, or supported in a restricted form. For example, the transition "add user" might be embellished such that users first receive invitations to join a group, acceptances and declinations are

tracked, and on an acceptance, a user is added to the group. Alternatively, "add user" might be subsumed in the transition "start application", whereby a user who starts an application automatically becomes a member of the group even if he or she just wants to start the application. Alternatively, "add user" might be restricted so that users can only be "connected", that is, added to the group by other people, and cannot "join" themselves. Chapter 4 details the transitions and the parameters supported by a broad range of systems.

Sequencing transitions

The transitions have a "natural" sequencing which arises as a result of the prerequisites for each transition. For example, "add user" requires "form group" to have been performed in order that a group exists to which a user can be added. Sequencing that is enforced in a system in addition to this "natural" sequencing amounts to a restriction on supporting the complete session management functionality.

In order to illustrate how the session management transitions might be used to manage users and applications in a group activity, consider the formation of a group with two users, supported in their common task by two applications. The "natural" sequencing is indicated.

User exists <A>	null → [uA]	
User exists <B>	null → [uB]	

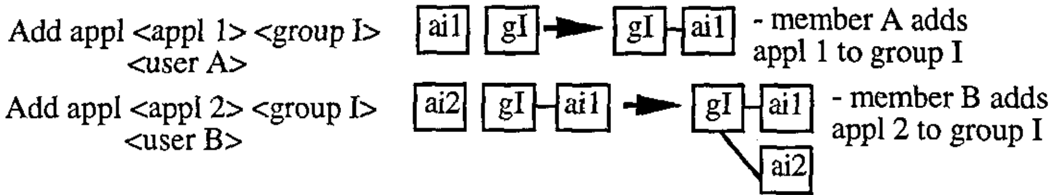
User A must exist before performing the following transitions:

Start application <appl 1> <user A>	null → [ai1]	- user A starts appl 1
Start application <appl 2> <user A>	null → [ai2]	- user A starts appl 2
Form group <group I> <user A>	null → [gI]	- user A forms group I

Group I must exist before performing the "add user" transitions:

Add user <user A> <group I> <user A>	[uA] [gI] → [uA]-[gI] - user A joins group I
Add user <user B> <group I> <user A>	[uB] [gI]-[uA] → [uB]-[gI]-[uA] or [gI] - member A connects user B to group I - members do not need to be considered separately from the group

An application must be started before performing the "add application" transition.



The sequencing of the transitions used in the example is not the only sequencing that might have been used. For example, applications might have been added to the group before the users. If a system does not support this flexibility of sequencing, the system supports a restricted form of the functionality.

Groupware application environment

If the group-work support is embedded in the applications themselves, the above transitions suffice. However, in the case where there is a separate application environment or a shared window system, the entities and their state transitions need to be re-considered. Four entities are taken: user, group, groupware application environment and application instance. The entity relationship model is shown in figure 3.9.

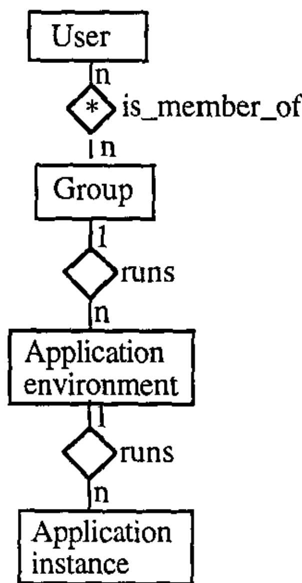


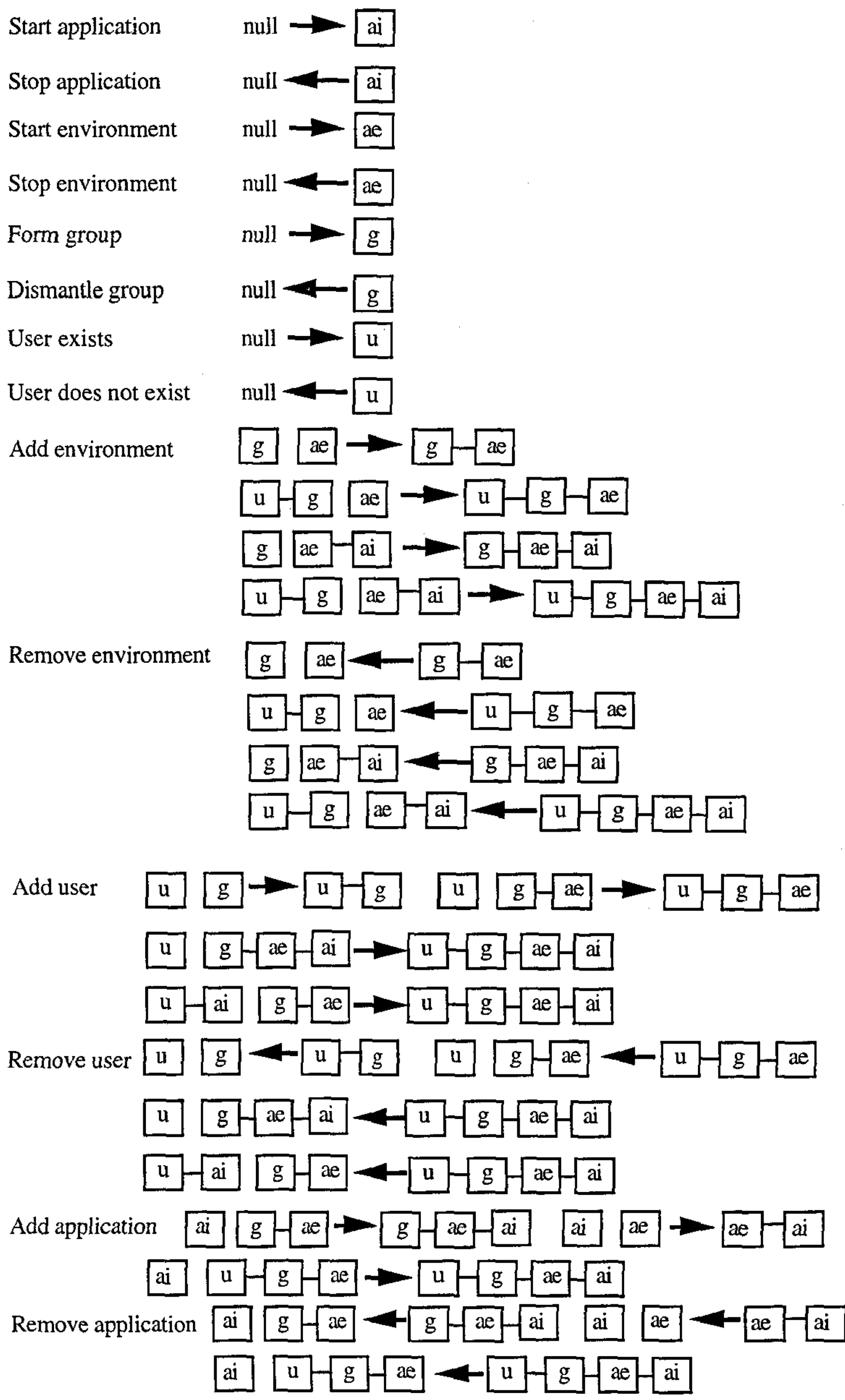
Figure 3.9 Model with Entities: User, Group, Application environment and Application instance

The groupware application environment (ae) has two states: either it is running or not. The combinations of the states of the four entities are identified below. Certain combinations are eliminated for one of two reasons. Firstly, a groupware application

environment can be used by a group and not a user. Secondly, a collaboration-transparent application instance must run in a groupware application environment in order to be accessible to a group.

	User	Group	Appl env	Appl inst	
1.	x	x	x	x	- null case
2.	x	x	x	√	[ai]
3.	x	x	√	x	[ae]
4.	x	x	√	√	[ae]-[ai]
5.	x	√	x	x	[g]
6.	x	√	x	√	X - ai must run in ae to run in g
7.	x	√	√	x	[g]-[ae]
8.	x	√	√	√	[g]-[ae]-[ai]
9.	√	x	x	x	[u]
10.	√	x	x	√	[u]-[ai]
11.	√	x	√	x	X - ae is used by g and not u
12.	√	x	√	√	X - ae is used by g and not u
13.	√	√	x	x	[u]-[g]
14.	√	√	x	√	X - ai must run in ae to run in g
15.	√	√	√	x	[u]-[g]-[ae]
16.	√	√	√	√	[u]-[g]-[ae]-[ai]

The basic transitions between the states of the four entities are listed below. Other transitions have been eliminated because they can be created from a combination of these basic transitions.


Start application
null
ai
Stop application
null
ai
Start environment
null
ae
Stop environment
null
ae
Form group
null
g
Dismantle group
null
g
User exists
null
u
User does not exist
null
u
Add environment
Remove environment
Add user
Remove user
Add application
Remove application

These transitions are similar to the transitions identified when there is not a groupware application environment entity. However, there are four new transitions: "start environment", "stop environment", "add environment" and "remove environment". The parameters for these new transitions are identified below. Further, the group identifier parameter for the transitions "add application" and "remove application" has changed to the environment identifier.

It is possible to support suspended sessions by storing the state of an application environment in addition to storing the states of a group and the running application which have already been mentioned. By storing the state of the application environment when it is stopped, the environment is said to be suspended and can be re-started from that suspended state.

The basic transitions have associated parameters, such as identifiers, actors and states. This is similar to the case analysed before where the groupware application environment entity did not exist. The transitions with their parameters are listed below.

• Start environment <env id> <actor>, and <env state> if the application environment has been suspended, where actor is a member, the system or a user.

• Stop environment <env id> <actor>, and <env state> if the application environment is suspended, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

• Add environment <env id> <group id> <actor>, where actor is a member, the system or a user.

• Remove environment <env id> <group id> <actor>, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

• Add application <appl id> <env id> <actor>, where actor is a member, the system or a user.

• Remove application <appl id> <env id> <actor>, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

The transitions listed above, together with those listed before, offer a complete description of session management functionality when the group-support is embedded in a separate entity to the applications.

3.3. Data Sharing

Data sharing considers the different classes of data which might be supported in an application, the sharing of these classes of data amongst users and the implications of the shared states for users. For example, one type of sharing supports the same view of the application to different users and another type supports different views. Since a running application might be supported by persistent data, data sharing refers to both application data and persistent data sharing.

The first section of this chapter describes the data entities, outlining the different classes of data entity and detailing the relationships between the classes. The second section describes the different states of shared data entities and the third section describes the transitions between the data entities and their shared states.

3.3.1. Entity relationship model

An application instance is considered to contain **abstract data objects.** Abstract data objects are composite data objects. An abstract data object only exists while an application is running and hence, use of the term abstract. The application data model is shown in figure 3.10.

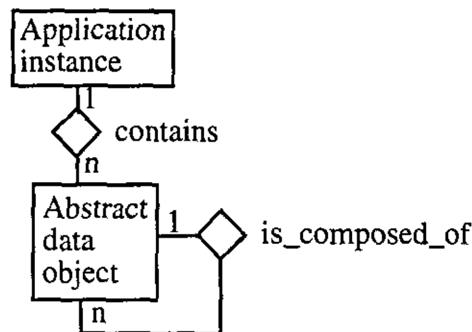


Figure 3.10 Application Data Model

Persistent data is considered to comprise **persistent data objects**. Persistent data refers to the stored data that supports an application instance. It might be distributed over several system files.

Persistent data objects and abstract data objects can be considered as a common entity, namely **data objects**. The combined data model is shown in figure 3.11. Data objects can be grouped together to form composite data objects, as indicated by the recursive link in the data model.

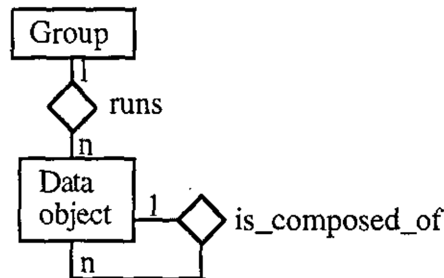


Figure 3.11 Data Model

Each data object belongs to a particular class, namely: physical representation, logical representation, function or persistent. The classes of data object are no different to those that might be found in a single-user application. The reason for examining the data model is to describe the sharing that occurs between the different classes of object in the group context, as described in section 3.3.2. The data object classes are discussed below, beginning with the class that the user interacts with directly.

Physical representation data refers to the class of data that can be perceived by a user at a workstation. The data can take different forms. In a windows environment, the physical representation data is the display objects on the screen. In a command line interface, the physical representation data is the lines of text output on the screen. Physical representation data also includes the meta representations made on the screen, such as the position of the pointer, the position of the cursor and the annotations. Different pieces of software might contribute to the formation of physical representation data in addition to the application, such as, device drivers, display servers, window systems and overlay software. In fact, physical representation data is a composite data object that can be further unwrapped into different types of specification, as described in figure 3.12, but the specification is implementation-dependent.

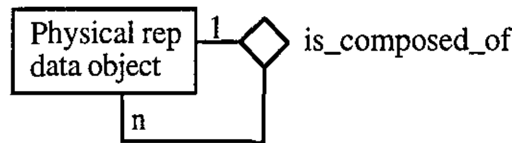


Figure 3.12 Composite Physical Representation Data Model

For example, in a windows environment, a physical representation data object might be described as a dialogue box containing a line of text and two control buttons, one to confirm and the other to cancel. Alternatively, the same physical representation data object might be described as a form widget located at a specific position, of a specific size and with a specific "look and feel", and containing etc..

Logical representation data refers to abstractions of the interaction data as presented to a user at a workstation. Two examples of systems that support this class of data are the Graphical Kernel System (GKS) (Hopgood et al., 1983) and FOCUS (Edmonds & McDaid, 1990). GKS is built upon a logical model of input and output devices, and interaction modes. For example, a logical model of an input device is a choice, which provides a selection of a set of alternatives. A logical model of an output device is fill area, which displays a specified area. An interaction mode is event mode, where input is generated asynchronously by the user; window systems support this mode. FOCUS is built upon abstractions, such as question, query, inform, selection and choice, and supports the interaction modes: event, application request and application sample. For example, in FOCUS, the question abstraction consists of text output and requires text input in response. The visual appearance of the question abstraction depends on its realisation in physical representation data.

Function data refers to the data objects created to support the functionality of the application. Function data is distinguished from the functionality of an application. For example, in a statistics package, part of the functionality might be to calculate the mean of a set of numbers. In calculating the mean, function data objects are produced. Different pieces of software might contribute to the formation of the function data, such as, software that supports a task model and a domain model. Therefore, function data, like physical representation data, is a composite data object that can be further unwrapped into different types of specification, as described by figure 3.13, but where the unwrapping is implementation dependent.

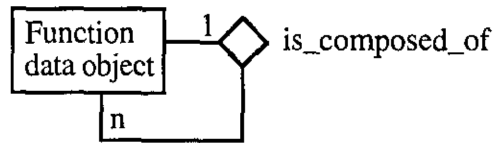


Figure 3.13 Composite Function Data Model

Persistent data refers to the stored data used by an application (as mentioned earlier). Persistent data is commonly stored in system files.

Figure 3.14 shows the relationship between data objects of different classes. It is possible to have different physical realisations of the same logical representation data, to have different logical representation data interact with the same function data, and to have different function data access the same persistent data. Figure 3.14 offers a complete description of the data classes.

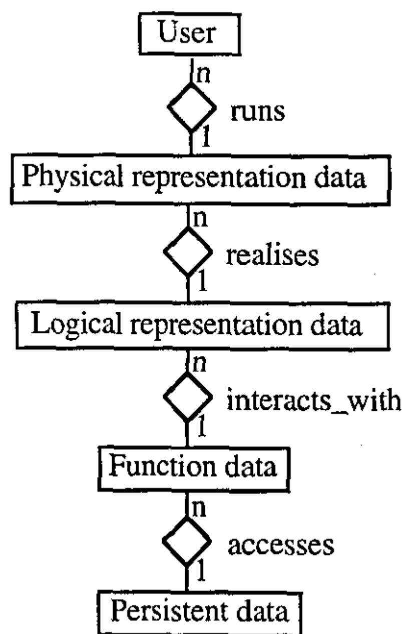


Figure 3.14 Relationships between sets of data objects of different classes

As an application instance comprises multiple data objects of different classes, figure 3.14 can be viewed as an instantiation of a composite data object, where the data objects are grouped into sets of classes. The different classes of data object and the actions carried out on the classes, including interaction between the classes, are supported by an application and its persistent data. An application creates function data whilst performing the application functionality. It supports access to the persistent data and selects the appropriate abstract interaction objects. It forms the

abstract interaction objects which constitute the logical representation data and realises them on a user's display, thus producing the physical representation data perceived by a user.

The focus of control in an application resides commonly in the functionality which handles either the logical representation data or the function data. In the former, the application is referred to as user-driven, and in the latter, the application is referred to as application-driven.

Not all classes of data are necessarily supported by an application. Commonly, the logical representation data is not supported and an application must support interaction directly between the function and physical representation data. As a result, function data becomes more involved with the physical characteristics of the interaction. However, it would be unusual if an application did not support physical representation and function data, and the interactions between them and persistent data.

Some applications separate support for user interaction and support for application functionality. As a result, it is possible to build specific components to support user interaction which are application independent. Generic components, such as some User Interface Management Systems (UIMS), support physical and logical representation data, and the interactions between them. Other components, such as window systems, support physical representation data and its realisation.

The granularity of a data object and the amount of interaction between the different classes of data object are issues central to the field of research known as user interface management. The coarse level of granularity leading to the separation of the user interface sub-system and the application functionality is justified for both its software engineering advantages (separate and parallel development, improved maintenance and portability) and end-user advantages (higher level interfaces, user-centred locus of control and interface consistency). However, when the granularity is too coarse, application designers are tempted to reduce the level of abstraction of the data exchanged between the function and interaction objects. As a result, some interface objects seek to appropriate application functions, such as checks upon user input and the provision of feedback to the user. This leads to problems of duplication, inconsistency and a lack of co-ordination (Dance et al., 1987) (Coutaz, 1989). A finer grain of data exchange is needed to support rapid semantic feedback. Edmonds (1993) reports on research into user interface tools aimed at tackling these issues and achieving the separable user interface.

In applications that support group-work, the level of information sharing between data objects is an issue. Low-level data objects, such as function data, have an impact on the way in which groups interact with an application. If low-level objects are modularised and separated, important properties can be hidden behind abstraction barriers, inaccessible to other data objects, even though the properties can have a significant impact on objects' interaction with users. As a result, it becomes problematic for group interaction to influence these properties. For example, historically, data distribution and concurrency control have been shielded from applications (e.g. distribution transparency) and from users. However, data distribution and concurrency control have significant impact on the ways in which groups interact in terms of responsiveness and support for parallel working. For instance, if data consistency is maintained using a strict locking mechanism in which only one user can access a data object at any one time, then potentially, another user has to wait for the lock on that object to be released before acting. The semantics of an application can be exploited to increase concurrency while maintaining adequate data consistency. For example, a "read" operation might not require an object to be locked and some delays can be avoided. Further, users might be made aware that a conflict has arisen and allowed to mediate their own actions. There is a need for interaction objects to have a handle on the low-level objects in order to control the underlying properties. Dourish (1995b) indicates that objects need to contain an embedded model of their own semantics which is accessible to objects of other classes in order to fully support group-work.

In summary, the primary entity in data sharing is the data object, and four classes of data object and their corresponding relationships are described in figure 3.14.

3.3.2. States

When multiple users run an application, four possible shared data states can be identified, corresponding to the class of data object that is shared. The four shared data states are: physical representation data sharing, logical representation data sharing, function data sharing and persistent data sharing. The sharing can take place by multiple users accessing a single copy of a class of data object or by synchronising replicated classes of data object, depending on the architecture employed by the application. A centralised architecture supports the former and the shared states are shown in figure 3.15. A replicated architecture supports the latter and the shared states are shown in figure 3.16. Shared access and synchronisation are discussed in

section 3.4 on concurrency control. Figures in this section indicate only two users but the description is easily generalised to any number of users.

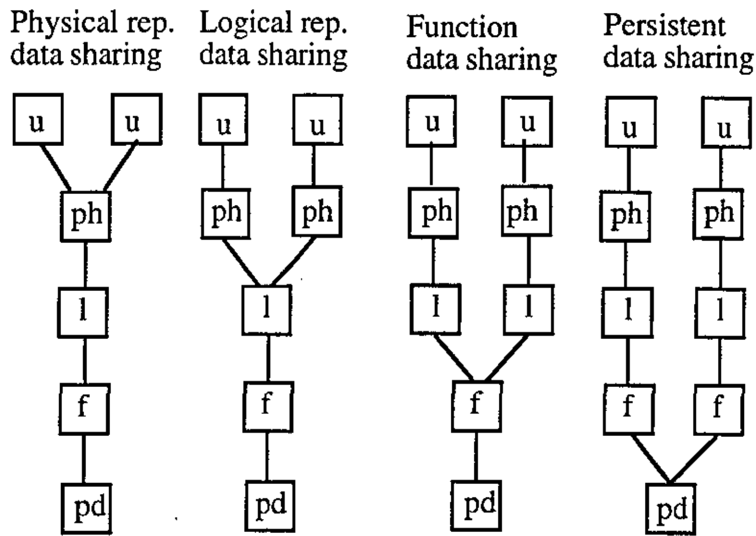


Figure 3.15 The Shared States in a Centralised Architecture

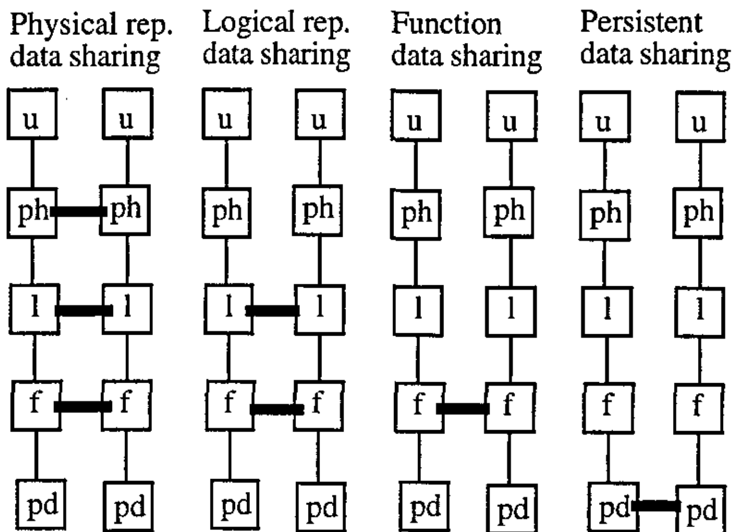


Figure 3.16 The Shared States in a Replicated Architecture

A hybrid of these architectures is also possible. For example, the architecture of an application could support physical representation sharing by having a single copy of function and persistent data objects which are shared and replicated copies of logical and physical representation data objects which are synchronised, as shown in figure 3.17.

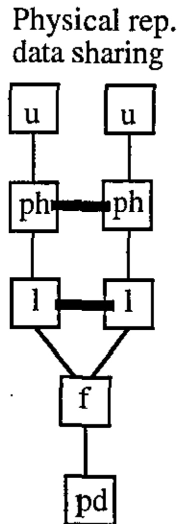


Figure 3.17 Physical Representation Sharing in a Hybrid Architecture

Users within the same group might have different shared states. Figure 3.18 shows two users sharing the same physical representation data and all three users sharing the same persistent data. Users with the same shared state are considered to constitute a user sub-group. In figure 3.18, one user sub-group comprises two users physical representation data sharing, and one user sub-group comprises three users persistent data sharing.

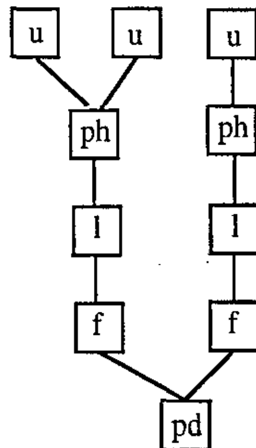


Figure 3.18 Different Shared States in a Group

A shared state has a specific scope, which might be for one particular object, for all objects that comprise an application instance, for all objects that comprise an application environment, for all objects that comprise a group, or for all objects that comprise all groups. If the scope is an object, then the shared state is applied to the

corresponding data objects in the different classes. As a result, different views of the same application are supported; users active on the same application instance could have different objects in different shared states. If the scope is an application instance, then the shared state is applied to all objects that comprise the application instance. As a result, users active on the same application instance would have all objects in that application in the same shared state but could have objects in other applications in different states. If the scope is an application environment, then the shared state is applied to all objects that comprise the application environment. As a result, users would have all objects in the application environment in the same shared state. If the scope is a group, then the shared state is applied to all objects that comprise the group. As a result, users would have all objects in the group in the same shared state. If the scope is all objects that comprise all groups, the shared state is applied to all objects. The latter is sometimes extended to workstation scope, and the shared state is applied to all objects that comprise applications including those run independently of the group.

It is possible to have partial sharing of data classes whereby some aspects of a class are shared and others are not. This is particularly applicable to physical representation data sharing (refer to that section).

Although data sharing refers to the functionality that enables users to view application output on their displays, it is also necessary to consider user input, and in particular, to maintain the data consistency. For example, in physical representation data sharing, to keep physical representation data the same for each user, changes made to the data for one user need to be reflected to other users. Essentially, there are two mechanisms to maintain data consistency, access control and synchronisation, which are discussed in section 3.4. The following discussion indicates to which class of data object the mechanisms must be applied to support a particular shared state.

Each shared state is discussed in detail in the following.

Physical representation data sharing

Physical representation data sharing results in multiple users perceiving the same data on their displays, commonly called WYSIWIS (What You See Is What I See), as shown in figure 3.19. Complete physical representation data sharing includes sharing the following aspects: application output, user input, window management functions (including scroll position, size, placement, icon state, stack position and "look and feel"), pointer position, cursor position and annotations. In a centralised architecture, users perceive one copy of the physical representation data and therefore must be

situated at the same workstation. In a replicated architecture, complete physical representation data sharing is not generally feasible, although it is possible. Commonly, partial physical representation data sharing is supported and the aspects shared depend on the implementation.

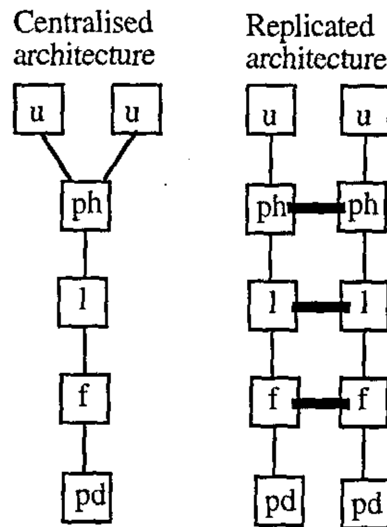


Figure 3.19 Physical Representation Data Sharing

The physical representation class of data objects must remain consistent on user input. In a centralised architecture, access to the shared physical representation class of data objects must be controlled, and as a result, the other classes remain consistent. In a replicated architecture, physical representation, logical representation and function data classes of data objects must be synchronised. If the function class of data object is synchronised, then the persistent data class is synchronised. Therefore, in a replicated architecture, each class of data is shared. For example, in the physical representation data class, any change in a window management function on one user's display requires the same change in window management function on other users' displays. In the logical representation data class, any value input entered by one user is shared with other users. In the function and persistent data classes, any function or persistent data changed by one user is shared with other users.

To illustrate the difficulties in achieving complete physical representation data sharing, take the example of a windows application employing a replicated architecture. The widgets, the window management functionality, the pointer and cursor positions, must be shared. Users essentially must share the same window system, the window managers need to be coordinated and there needs to be a shared pointer and cursor facility. Commonly, a window manager is single-user and does not

publicise its activities or provide a handle on its functionality. For example, if a user re-positions a window, it is necessary for physical representation sharing that other users have the same window re-positioned. However, generally, a window manager does not provide information about the new position of a window, nor is it possible to tell a window manager to re-position a window. Therefore, it is necessary to develop group-aware window managers. Some shared window systems overcome this by sharing the window manager as an application, e.g. Dialogo (Lauwers, 1990), but this has other drawbacks, such as restricting input to one user at a time. Dialogo supports a group pointer (also known as a telepointer) which displays an arrow-shaped image on the display and is accessible by all users.

Another approach has been to consider what aspects of physical representation sharing need to be shared in order for users to have a common understanding. Crowley et al. (1990) allow users of MMConf to have different window placements but not different icon states. However, the Cognoter team found that an extremely flexible layout of shared windows made sharing common references in discussion difficult (Tatar et al., 1991).

Logical representation data sharing

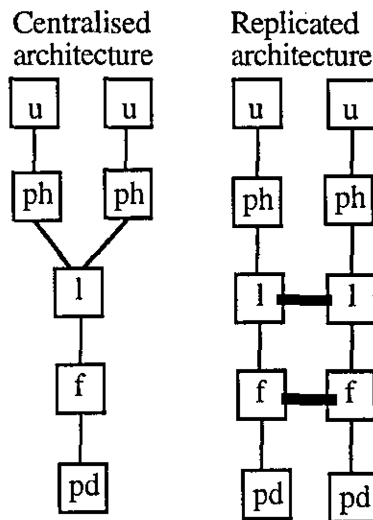


Figure 3.20 Logical Representation Data Sharing

Logical representation data sharing results in multiple users having the same interaction abstractions, but different physical realisations of those abstractions on their displays, as shown in figure 3.20. For example, a question abstraction object might be realised on displays running different window systems and thus take on the "look and feel" of each window system.

The logical representation class of data objects must remain consistent on user input. In a centralised architecture, access to the shared logical representation class of data objects must be controlled, and as a result, the function and persistent data classes remain consistent. In a replicated architecture, logical representation and function data classes must be synchronised. (If the function class of data object is synchronised, then the persistent data class is also synchronised.)

Since very few systems support logical representation data, this type of sharing is uncommon. MUMS is an example of a system that does support logical representation sharing (Jones & Edmonds, 1994).

Function data sharing

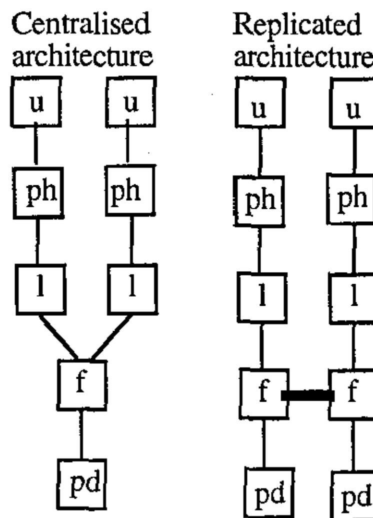


Figure 3.21 Function Data Sharing

Function data sharing results in multiple users sharing the same function data but having different interactions with the application, as shown in figure 3.21. For example, imagine two types of user using a statistics package, an expert and a learner. An expert would know which statistics function to use and the conditions for its use, and would benefit from easy access to the function. A beginner would need additional help selecting and accessing the statistics function. The interactions with the application would be different for the two types of user.

The function class of data objects must remain consistent on user input. In a centralised architecture, access to the shared function class of data object must be controlled, and as a result, the persistent data class remains consistent. In a replicated

architecture, the function class must be synchronised. (If the function class of data object is synchronised, then the persistent data class is also synchronised.)

Very few systems, if any, consider a group of users having different interactions with an application. MUMS is an example of a system that could support this functionality (Jones & Edmonds, 1994). Many systems support different views of the same application data, that is, different physical representations of the same function data, such as Rendezvous (Patterson et al., 1990). These systems do not support logical representation data as a distinct class of data. However they are considered to support function data sharing where the different interactions with an application adopt a limited form, such as selecting different aspects of the application output and tailoring its presentation to each user. For example, one Rendezvous application is a card game where each user views his or her own hand.

Persistent data sharing

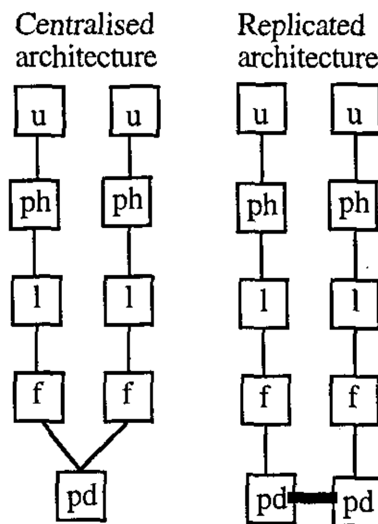


Figure 3.22 Persistent Data Sharing

Persistent data sharing results in multiple users sharing the same persistent data but running different applications or different instances of the same application, as shown in figure 3.22. For example, a file of ASCII text could be shared by one user running electronic mail and another user running a word processor.

The persistent data class of data objects must remain consistent when accessed by applications. In a centralised architecture, access to the persistent data class of data object must be controlled. In a replicated architecture, the persistent data class of data objects must be synchronised. Both access control and synchronisation are supported by a special type of application, commonly called a database. Databases interact with

the users themselves to indicate that the persistent data has changed and what changes have been made, as supported in some version control systems.

Persistent data sharing is also referred to as loosely-coupled sharing. Loosely-coupled sharing is defined as users focused on different aspects of the task at the same time or users focused on the same aspect of the task at different times. Up to now, persistent data sharing has referred to users focused on different aspects of the task at the same time. In this context, different aspects of the task refers to different application functionality since application functionality is there to support a task. In fact, all the other shared states have also considered sharing at the same time, i.e. physical representation, logical representation and function data sharing. They are sometimes referred to as tightly-coupled sharing which is defined as users focused on the same aspect of the task at the same time. None of these other types of sharing can be considered as sharing at different times.

Loosely-coupled sharing in which users focus on the same aspect of a task at different times is also described by figure 3.22. Each user is considered to have his or her own instance of an application which run at different times but which share the same persistent data. For example, one user might construct and send an e-mail and another user might receive and read the e-mail. The persistent data remains consistent because it is accessed by users at different times.

Systems have been developed to control access to the persistent data based on activity and process models, e.g. workflow systems. Examinations of these systems is outside the scope of this work.

3.3.3. State transitions

State transitions in data sharing can be divided into three categories: data object transitions, shared data transitions and data state transitions. Data object transitions are supported by the application. Shared data transitions are supported by group-work support functionality. Data state transitions are supported by application state transfer functionality.

Data object transitions

The primary entity in data sharing is the data object and a data object belongs to a particular class of data. When an application runs, data objects are created and deleted. Two transitions are defined:

• Create data object <data object id> <class>
where <class> is physical representation, logical representation, function or persistent data.

• Delete data object <data object id>

The application is responsible for interaction with the user via physical representation data. The actual visual appearance of a data object is defined by:

• Realise data object <physical representation data object id>

Shared data transitions

The group-work support functionality in an application or an application environment supports data sharing. The type of sharing and the scope of sharing are defined when a user is added to a group. Therefore, the session management transition "add user" is enhanced to the following specification:

• Add user <user id> <group id> <actor>
    (<shared state> <sharing scope> <user sub-group> <when shared>)*
    and <appl state> if the user brings an application instance to the group.

<actor> is either a member or a user. <shared state> has the value: physical representation, logical representation, function or persistent data sharing. The shared state also indicates what aspects of the data class are shared which is necessary particularly for physical representation data sharing. <sharing scope> is defined in terms of data objects and has the value: data object, application instance, application environment, group, all groups or a specified group of data objects. <user sub-group> is a specified sub-group of users with the same shared state and scope, within a group. <when shared> indicates at what time data objects are shared to this specification. The attributes <shared state>, <sharing scope>, <user sub-group> and <when shared> define the type of sharing between specified objects, by specified users, at a specified time. These attributes are repeated accordingly.

The sharing attributes, such as the shared state, are changed by using the transitions "remove user", followed by "add user" with new attributes.

Although the sharing attributes are associated with a user, the attributes refer to data objects and thus, do not come into effect until an application is added to the group.

The group-work support functionality that supports data sharing includes data distribution mechanisms and data consistency mechanisms. Data distribution mechanisms handle the replication and location of data objects. These mechanisms require the addition of a <location> attribute to the transition "create data object" and the specification of a new transition:

• Create data object <data object id> <class> <location>

• Copy data object <data object id> <location> <new data object id> <new location>

A combination of the transitions "copy data object" and "delete data object" support the relocation of a data object (or the move transition).

The data consistency mechanisms are described in detail in section 3.4 on concurrency control. A consistency ticket entity is introduced and two new transitions defined, "create consistency ticket" and "delete consistency ticket". By creating a consistency ticket, a link is defined between a user and a data object which contains appropriate mechanisms to direct the user input.

Data state transitions

In order to support suspended sessions, application transfer and data repair, it is necessary for an application to store the state of a data object and be able to transfer the state. Therefore, the transition "create data object" must be enhanced:

• Create data object <data object id> <class> <location> <data object state>

A session is suspended and re-started by storing and transferring the state of a group, an application environment and any applications. Since an application contains data objects, storing and transferring application states requires storing and transferring data object states. This is seldom supported by applications.

In the two cases of the transition "add user" that involve a user or group running an application, it is necessary to transfer the application state in order to ensure users are in the specified shared state. In the first case, a group is running an application when a user is added and by transferring the application state to the new user, the latecomer is brought up to date. In the second case, a user running an application is added to a group and by transferring the application state to the group, the group is brought up to date.

Data repair requires a data object to revert to a previously stored state and is discussed in section 3.4 on concurrency control.

3.4. Concurrency control

Concurrency control refers to the functionality that enables multiple users access to the same application and to the same data at the same time. The focus of this section is on maintaining data consistency. Some forms of concurrency control enable the data to become inconsistent and depend on mechanisms to reduce potential conflict and repair the data. These mechanisms are discussed later in this section.

3.4.1. Entity relationship model

Different data consistency mechanisms need to be applied to the application and data depending on the degree of replication adopted by the application and the data, and the type of task supported. Three degrees of replication are considered: centralised, replicated and hybrid. The centralised approach involves having one copy of the application instance and the persistent data. The replicated approach involves having multiple copies of the application instance and the persistent data, one copy for each user, that are distributed across the different user sites. The hybrid approach involves parts of the application instance and the persistent data being centralised and parts of them being replicated. If users are located at different geographical sites, it is beneficial for performance reasons to adopt some degree of replication.

Tasks are broadly categorised into two types: tightly-coupled and loosely-coupled. If parts of the application instance are replicated and tightly-coupled tasks are supported, the replicated parts need to be kept in synchronisation (figure 3.23a). It is assumed that if the application instances are synchronised, then multiple copies of permanent data are kept consistent. If parts of the application are replicated and loosely-coupled tasks are supported, version control and merging techniques need to be applied to multiple copies of the persistent data (figure 3.23b). If parts of the application are centralised and tightly-coupled tasks are supported then, potentially, multiple users can access one copy of the application. As a result, access to the application needs to be controlled (figure 3.23c). If parts of the application are centralised and loosely-coupled tasks are supported then, potentially, multiple users can access one copy of the persistent data. As a result, access to the persistent data needs to be controlled (figure 3.23d).

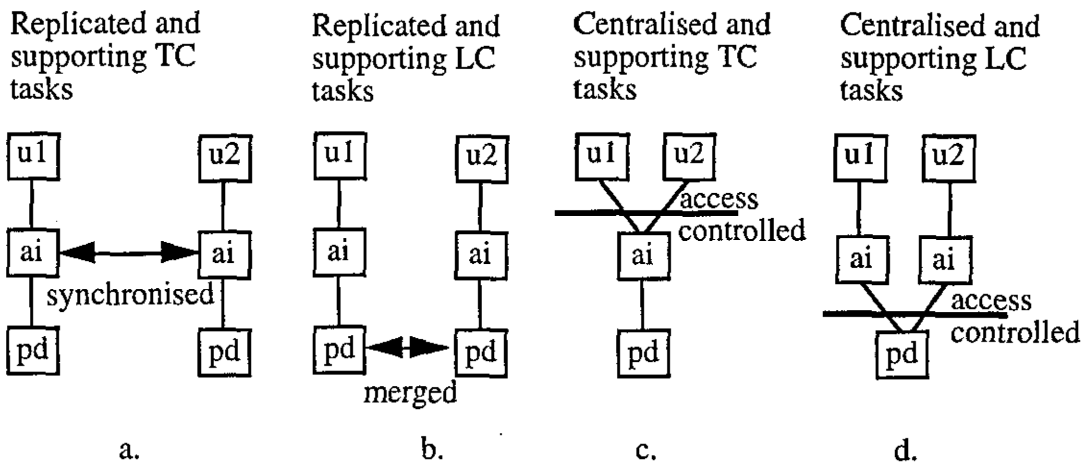


Figure 3.23 Four cases where concurrency control is needed in tightly-coupled (TC) and loosely-coupled (LC) tasks

Broadly, there are two mechanisms for maintaining data consistency, serialisation and locking. Serialisation is used to ensure the consistency of a sequence of events or messages and to maintain synchronisation. It can be used to support the case shown in figure 3.23a. Locking can be used to ensure the consistency of a data object. It supports the cases shown in figures 3.23c and 3.23d. Locking can also be used to ensure the global serialisation by controlling the order that sites obtain and release locks. As a result, it can also be used to support the case shown in figure 3.23a. Both locking and serialisation can support different levels of data consistency which can be categorised as pessimistic, optimistic and semi-optimistic.

Pessimistic consistency maintenance employs a more traditional mechanism used for managing conflicts by avoiding them. It ensures strict serialisation whereby events are executed in the correct order at all sites. It ensures strict locking whereby a user is not allowed to manipulate an object until the lock for that object is approved. An example of a system that supports pessimistic consistency maintenance is the source code control program RCS (Tichy, 1982). RCS provides coarse grain locking whereby parts of documents are locked out for lengthy periods of time. Pessimistic consistency maintenance is highly secure but leads to time delays. Delays can have a significant impact on the way in which groups interact and, therefore, there has been a growing interest in more optimistic forms of consistency maintenance to improve concurrency.

Optimistic consistency maintenance is based on the assumption that conflicts are rare and that it is more efficient to proceed with execution and then repair or resolve the conflict, than to guarantee data consistency at all times (Dourish, 1995a). Optimistic

serialisation does not require the user to wait for synchronisation before events are transmitted and, potentially, events can arrive in a different order at different sites. Optimistic locking allows a tentative lock to be granted before approval is gained, and potentially, multiple users can access an object at the same time. However, if a conflict does occur, then repair can be difficult, if not impossible, and incurs high implementation costs. Semi-optimistic consistency maintenance aims to limit the amount of repair needed. For example, semi-optimistic locking ensures that the tentative lock is approved before a user moves onto manipulating other objects.

In some situations, such as that shown in figure 3.23b, data inconsistencies are allowable. The problem changes from one of maintaining consistency to one of carrying out repair and resolving inconsistencies. In other situations, inconsistencies do not matter because it is apparent to users that an inconsistency has arisen. For example, in a sketchpad, a user might draw-over, modify or rub-out another user's part of a sketch, but it is apparent to the original sketcher. Users might find this acceptable and even beneficial in the context. As a result, in some situations, data consistency does not need to be maintained.

Consistency guarantees is a mechanism for describing the level of data consistency offered by a system (Dourish, 1996). The amount of detail in which the level of data consistency is specified can be varied and the level of data consistency can be negotiated.

The Framework describes serialisation and locking.

Serialisation model

To describe how serialisation works, consider an event-based application that is replicated at each user site. A user interacts with the application at one site, which causes an event to be passed to a data object. In order for other sites to be kept in synchronisation, the event is distributed to the replicated application at other sites. Potentially, each user can interact with the application at the same time and inconsistencies can occur, refer to figure 3.24. Concurrency control is the activity of coordinating the potentially interfering actions of processes that operate in parallel. Algorithms can synchronise event interleaving to ensure the correct ordering of events at each site and thus identical behaviour at each site. One of the simplest algorithms involves time-stamping events. This global serialisation supports pessimistic data consistency. Algorithms for synchronising distributed objects can be described using petri-nets for example, but are beyond the scope of this work. As mentioned earlier, optimistic consistency maintenance is obtained by allowing events

to be received out of order, in which case inconsistencies can arise and repair might be needed.

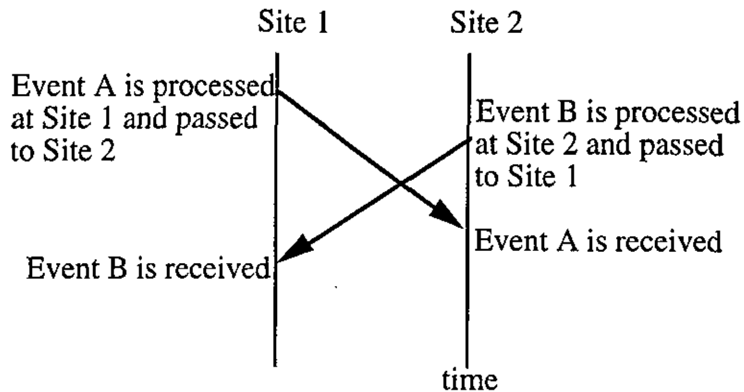


Figure 3.24 Potential conflict

Serialisation can be applied to the data model in two different ways, model A is shown in figure 3.25 and model B is shown in figure 3.27.

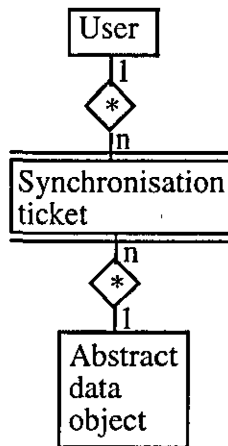


Figure 3.25 Serialisation model A

In model A, a **synchronisation ticket** defines the link between a particular user and an abstract data object and contains the serialisation mechanisms which synchronise the replicated objects, as shown in figure 3.25. A synchronisation ticket needs both a user and an abstract data object in order to exist. The serialisation mechanisms within a synchronisation ticket support different levels of consistency maintenance. Because a user may have access to many abstract data objects at many levels and any abstract data object may be shared by many users, entity relationship modelling allows us to denote a many to many relationship and discover a new entity which is denoted by a

synchronisation ticket. A user can have multiple synchronisation tickets and, as a result, multiple objects of different classes can be synchronised.

The distribution of events and the concurrency control are handled by mechanisms within the synchronisation ticket. Every replicated object has a synchronisation ticket and these communicate with one another, as shown in figure 3.26.

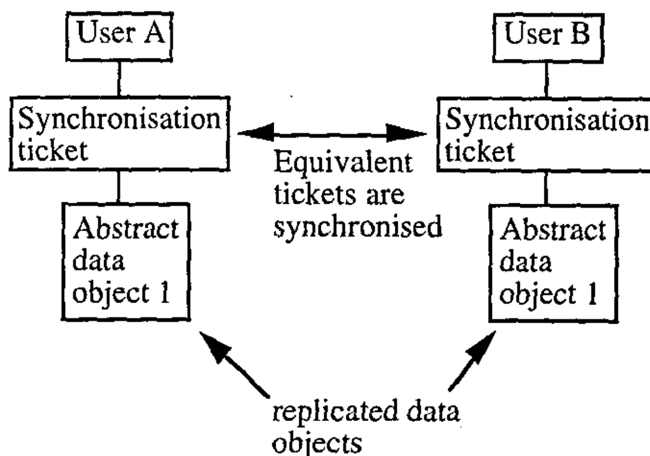


Figure 3.26 Synchronisation model A

In model B, a **synchronisation ticket** defines the link between replicated abstract data objects and contains the serialisation mechanisms which synchronise the replicated objects, as shown in figure 3.27. Any data objects which are identical can be synchronised. The serialisation mechanisms within a synchronisation ticket support different levels of data consistency.

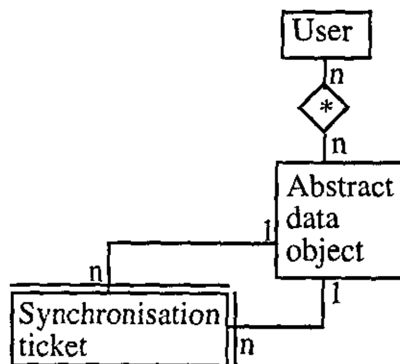


Figure 3.27 Serialisation model B

As in model A, the distribution of events and the concurrency control are handled by mechanisms within the synchronisation ticket. However, instead of there being

multiple synchronisation tickets, one per object, that communicate with one another, here there is logically only one (although it might be distributed) which coordinates communication between identical data objects, as shown in figure 3.28.

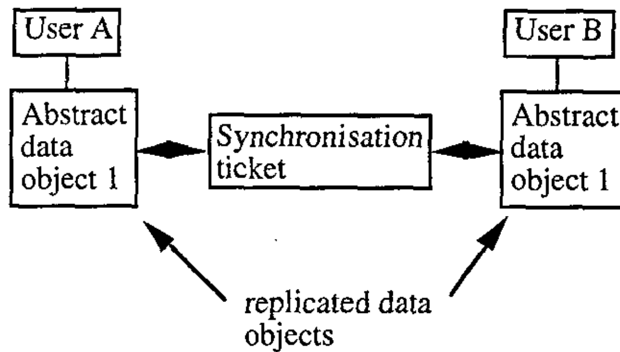


Figure 3.28 Synchronisation model B

The synchronisation ticket might itself comprise data objects of different classes which collectively operate to serialise application data objects. It is considered to be a meta-object.

In practice, serialisation model A supports event synchronisation and serialisation model B supports object synchronisation, such as state transfer.

Locking model

Locking is applied to the data model as shown in figure 3.29. An **access ticket** defines the link between a user and a data object and contains the mechanisms that enable the user to access the data object. A user has an access ticket for each data object. An access ticket needs both a user and a data object to exist.

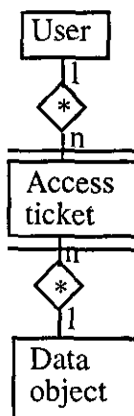


Figure 3.29 Locking Model

Mechanisms in an access ticket support different levels of data consistency. Pessimistic locking ensures that only one user has access to a data object at any one time. Optimistic locking enables multiple users access to a data object at any one time. Mechanisms in an access ticket support serialisation by controlling the order of access to data objects.

Data consistency model

The data consistency model is a combination of the serialisation model A and locking model and is shown in figure 3.30. A **consistency ticket** defines the link between a user and a data object. It contains mechanisms to handle serialisation and locking, as detailed for the synchronisation ticket (serialisation model A) and the access ticket.

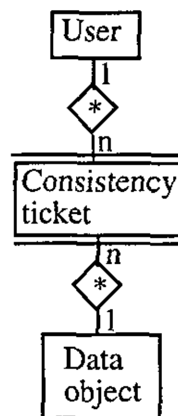


Figure 3.30 Consistency model

Collectively, consistency tickets contain information about which users have access to which data objects and the mechanisms that might be employed to gain access to data objects. This information can be made available to members of a group to inform members of the availability of access, the access facilities and the other members' activities.

3.4.2. State transition diagrams

Two basic transitions are needed in order to support the data consistency mechanisms and are defined as:

- Create consistency ticket <user id> <consistency scope> <access right>

- Delete consistency ticket <user id> <consistency scope>

<consistency scope> is defined in terms of data objects and has the value: object, application instance, application environment, group, all groups or a specified group of data objects. <access right> gives an indication of the level of data consistency and has the value: read, write or none. For example, if only one user has write access to a data object, the level of data consistency is pessimistic.

To change the consistency state, use the transition "delete consistency ticket" followed by the transition "create consistency ticket".

The consistency ticket transitions can be used to form access policies. For example, take the policy explicit request which requires a user actively to request access and only allows this user access to the data object. The transitions are:

• Create consistency ticket (user, data object, write)
• Create consistency ticket (all_other_users, data object, read)

Further, policies are set and changed dynamically. Thus, higher level primitives can be defined which represent the policies and manipulations of the policies. The primitives use the basic consistency ticket transitions but instantiate policy decisions. These higher level primitives are not considered part of the Framework.

3.4.3. Conflict reduction

Four different mechanisms for reducing conflict are discussed: affordances, making recourse to application semantics, fine granularity of data objects and lock release. Each of these mechanisms effects the data consistency mechanism.

People are very capable of mediating their own actions using familiar social protocols. By providing feedback and affordances of shared objects to support people's natural abilities, conflict can be reduced. This behaviour was first noticed in CoLab, a face-to-face meeting environment, in which objects were greyed-out when selected to indicate they were busy (Stefik et al., 1987a). Consistency tickets contain the relevant information for providing feedback.

Secondly, the data consistency mechanism can make recourse to application semantics to focus on operations where conflict can occur, rather than consider conflict to occur in all operations. For example, in a text editor, certain operations, such as the "read" operation, do not create conflict, whereas other operations, such as the "write" operation, can potentially cause conflict. Therefore, data consistency mechanisms needs to be applied to "write" but not "read" operations. GroupDesign uses the application semantics to define commutative actions and thus specify events

which can be out of order (Karsenty & Beaudouin-Lafon, 1993). Conflict can only occur when ordering matters. The transition "create consistency ticket" needs to be enhanced to include a parameter that specifies the application operation.

Thirdly, by keeping the data objects at a fine granularity, there is greater opportunity for concurrency. If the granularity is fine, there are more data objects and less opportunity that a data consistency mechanism denies access to a user and less delays. However, the trade-off is that more data consistency mechanisms are needed, which increases the overhead in the system of maintaining data consistency. Take for example a text editor, locking a document might be too coarse, whereas locking a character might be too fine. An approach which has been adopted to find the appropriate balance is to provide users with the facilities to select the granularity of data to which the data consistency mechanisms are applied. In MACE, a user specifies a pair of locks that bound the top and bottom of a text region (Newman-Wolfe & Pelimuhandiram, 1991). Since locked regions cannot overlap, users work on different parts of the text. The SASSE text editor locks selected text (Baeker et al., 1993). DistEdit also provides region locks on text which are set explicitly by the user or by the system, which tries to provide the finest grain lock possible (Prakash & Knister, 1992). One problem with locking regions is carrying out global operations. DistEdit tries to obtain all the locks needed for a global operation and only then carries out the operation. Therefore, the data consistency mechanism needs to enable users to establish consistency tickets which apply to multiple data objects which are specified by the users themselves.

Fourthly, with a locking mechanism, to increase the probability that a lock is granted and thus reduce some delay, locks can be released if the current holder is inactive. One such lock, known as a "tickle lock", is described by Greif and Sarin (1988). In other words, the data consistency mechanism needs to enable consistency tickets to be removed (e.g. time-out) if the user is not interacting with the data object to which it is linked.

3.4.4. Repair

Repair is needed when the data becomes inconsistent. Inconsistencies arise either because an optimistic data consistency mechanism is used or because no data consistency mechanism is used, as in the case shown in figure 23b. In the latter, two parallel versions of the data are maintained which later need to be merged. Systems such as CoVer are designed to support a group create and manage parallel versions of an object (Haake & Haake, 1993).

Firstly, consider repair that takes place in applications that support tightly-coupled tasks. Repair can take place by reverting to a particular application state. It is possible either to revert to a previously stored state, as in TimeWarp (Jefferson, 1985), or to apply undo functions. In the Framework, it is possible for data objects to revert to a particular state using the transition "delete object" followed by the transition "create object" with a previously stored state. DistEdit and Gina (Berlage & Genau, 1993) investigate the implications of group undo and discuss selective undo which involves dialogue with users. The efficiency of using undo functions can be improved by combining the approach with an approach that employs application semantics or transforms events. An example of a system that employs application semantics is GroupDesign, which only uses undo functions when event ordering matters. An example of a system that transforms events is Grove, which uses a set of rules to transform a pair of operations so that the effect is the same at both sites regardless of order (Ellis & Gibbs, 1989). The final outcome might be quite different to performing the operations in order, but the goal is to produce a consistent result. A consistency ticket needs to encompass a mechanism that enables users to invoke undo functions for application operations.

Building a sensible interface for optimistic consistency in applications that support tightly-coupled tasks is difficult because repair can result in something quite different to what is intended. (Greenberg & Marwood, 1994) consider the impact of consistency maintenance on the interface and how transactions are displayed and perceived by a group. They conclude that there is no recipe for presenting lock denial or data repair that minimises user confusion.

Secondly, consider repair that takes place on the persistent data, also considered as "late concurrency". One approach is to detect inconsistencies and inform users so that they may carry-out the repair. A consistency ticket needs to employ a mechanism for comparing data objects. Flexible Diff reports differences in text documents (Neuwirth et al., 1992). It allows users to choose the granularity of the differences found, and thus, how they are reported, e.g. in sentences. Gina maintains a command history which it uses when the application is running to perform undo/ redo functions. If two versions of an object are created, the command history associated with one is applied to the other, in effect, redoing one set of changes on the other version. Only if there is a conflict are users informed. The other approach attempts to repair the inconsistencies. Munson and Dewan (1994) enhance Suite, a group interface toolkit, with a merge matrix which allows automatic and interactive merge policies. Suite requires the user to specify the merge requirements for each operation on an object.

As a result, it provides a general framework that employs the application semantics. A consistency ticket needs to employ a mechanism which allows the merge requirements for application operations to be specified.

Conflict might be resolved by priority rather than repair. Specific application operations or specific users might take a priority over other operations or other users, respectively. Suite assigns a set of collaboration rights to data objects (Shen & Dewan, 1992). When conflict occurs, conflict resolution rules examine the access rights to determine who gets the object. It requires the data consistency mechanism to contain user priorities.

3.5. Integrated Framework

The integrated Framework consists of an entity relationship model, which identifies the important entities in collaboration, and a set of primitives, which describe the transitions between states of the entities. The Framework offers a complete description of the functionality in group-work support. The Framework is expected to apply to all group contexts, although only a sub-set of the functionality might be appropriate to a particular task and encompassed in a particular system. For example, the Framework describes all different types of sharing, as detailed earlier in this chapter, whereas a system might support just one type of sharing.

User interaction takes two forms: interaction with the application and interaction with the group support functionality. The Framework describes how users interact with an application, but there has been no discussion so far as to how users interact with the group support functionality. The group support functionality can be viewed as a special application with application functionality defined by the transitions. Status information is provided by membership tickets and consistency tickets. As mentioned in section 3.2 on session management, membership tickets contain information about users membership of groups and this information can be used to provide users with some indication of who the other members of a group are. As mentioned in section 3.4 on concurrency control, consistency tickets contain information about users access to data objects and the mechanisms that might be employed to gain access to data objects. This information can be used to indicate the availability of access, the access facilities and provide an awareness of other members' activities. The Framework focuses on describing the functionality involved in collaboration and not on how this functionality might be presented to users, which is considered beyond the scope of this work. Therefore, the Framework gives no indication of how the transitions and the status information are presented to users.

3.5.1. Entity relationship model

The integrated entity relationship model is described in figure 3.31. The integrated model is a combination of the session management model, shown in figure 3.6, the data model, shown in figure 3.11, and the consistency model, shown in figure 3.30.

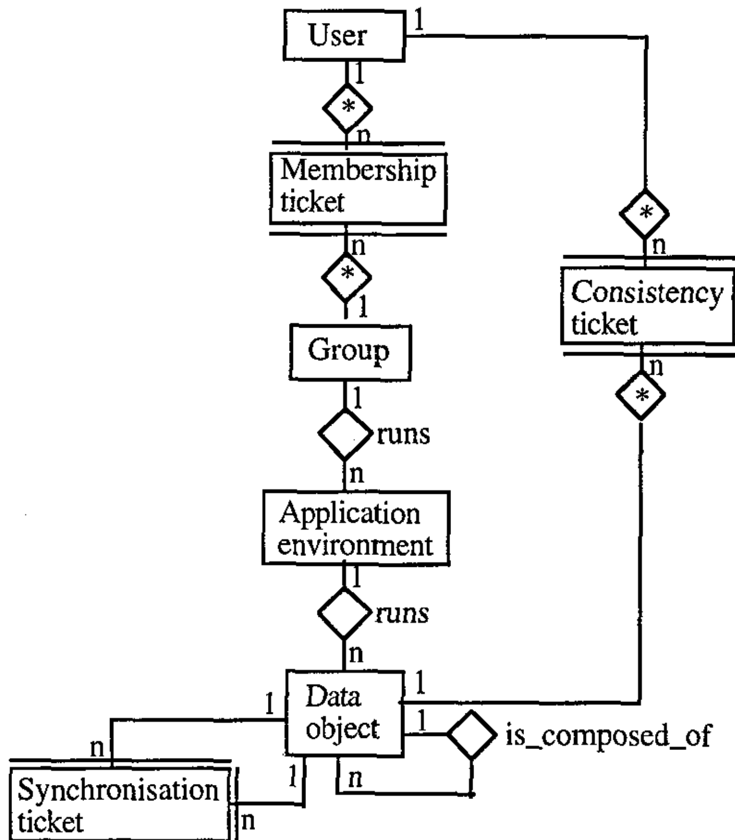


Figure 3.31 An Integrated Model of Collaboration

3.5.2. State transitions

A summary of the basic transitions is given below:

• Start application <appl id> <actor>, and <appl state> if the application has been suspended, where actor is a member, the system (a process set to start) or a user.

• Stop application <appl id> <actor>, and <appl state> if the application is suspended, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

• Start environment <env id> <actor>, and <env state> if the groupware application environment has been suspended, where actor is a member, the system or a user.

• Stop environment <env id> <actor>, and <env state> if the groupware application environment is suspended, where actor is a member, the system or a user.

• Form group <group id> <actor>, and <group state> if the group has been suspended, where actor is a user.

• Dismantle group <group id> <actor>, and <group state> if the group is suspended, where actor is a member or a user.

• User exists <user id> <actor>, where actor is a member or a user.

• User does not exist <user id> <actor>, where actor is a member or a user.

• Add user <user id> <group id> <actor>

(<shared state> <sharing scope> <user sub-group> <when shared>)*

and <appl state> if the user brings and application instance to the group, where <actor> is either a member or a user. <shared state> has the value: physical representation, logical representation, function or persistent data sharing. The shared state also indicates what aspects of the data class are shared, which is necessary particularly for physical representation data sharing. <sharing scope> is defined in terms of data objects and has the value: data object, application instance, application environment, group, all groups or a specified group of data objects. <user sub-group> is a specified sub-group of users with the same shared state and scope, within a group. <when shared> indicates at what time data objects are shared to this specification. The attributes <shared state>, <sharing scope>, <user group> and <when shared> define the type of sharing between specified objects, by specified users, at a specified time. These attributes are repeated accordingly.

• Remove user <user id> <group id> <actor>, and <appl state> if user takes an application instance from the group, where actor is a member or a user.

• Add application <appl id> <group id> <actor>, where actor is a member, the system or a user.

• Remove application <appl id> <group id> <actor>, where actor is a member, the system or a user. If the actor is the terminator, the group is being dismantled.

• Add environment <env id> <group id> <actor>, where actor is a member, the system or a user.

• Remove environment <env id> <group id> <actor>, where actor is a member, the system or a user.

• Create data object <data object id> <class> <location> <data object state>
<class> is physical representation, logical representation, function or persistent data.

• Delete data object <data object id>

• Realise data object <physical representation data object id> <logical representation data object id>

• Copy data object <data object id> <location> <new data object id> <new location>

• Create consistency ticket <user id> <consistency scope> <access right>
<consistency scope> is defined in terms of data objects and has the value: object, application instance, application environment, group, all groups or a specified group of data objects. <access right> gives an indication of the level of data consistency and has the value: read, write or none.

• Delete consistency ticket <user id> <consistency scope>

3.6. Conclusions

The entity relationship modelling and state transition techniques employed were found to be effective tools in helping to develop a complete and consistent Framework. For example, the membership ticket entity was derived using the entity relationship modelling technique.

The Framework treats group-work support as separate to application functionality without any problems. In addition, the Framework is able to support different forms of group-work support, that is, group-work support embedded in applications, application toolkits and application environments.

The Framework provides a complete description of the functionality in group-work support within the scope. In addition, it provides a framework with which to compare

group-work support systems. The functionality supported by a particular system can be represented by entities, transitions with specific parameter values and possibly a particular sequence of transitions specific to the system. Systems that support collaboration might not make explicit the functionality they support in terms of entities and transitions. The transitions might be embellished and hidden behind a more elaborate interaction. The following chapter describes some well-known systems in terms of the entities and transitions they support.

Chapter 4:

Characteristics of Existing Systems

4.1. Introduction

Chapter 3 developed a Framework that is a tool with which to compare systems that support group-work. To test whether the Framework meets its objective, the Framework is used to characterise a representative set of systems and then the characteristics are compared. The representative set of systems were introduced in Chapter 2. This chapter uses the Framework to characterise this set of systems. The systems include: Rendezvous, GroupKit, Dialogo, Suite, SEPIA, MEAD and SOL of which the first five systems are analysed in detail. The analysis for each system is divided into three broad categories of functionality in group-work support outlined by the Framework: session management, data sharing and concurrency control. MEAD and SOL are characterised in summary.

4.2. Rendezvous

Rendezvous is a groupware toolkit for building multi-user, real-time applications with graphical direct manipulation interfaces. It is the most extensive toolkit developed to date. It is used as a major system for analysis by the Framework.

4.2.1. Session Management

Figure 4.1 illustrates the entities and the entity relationships supported by Rendezvous as described in the Framework.

A Rendezvous session is a running Rendezvous application. The group entity does not exist explicitly in Rendezvous. A group constitutes those users who are in a session and, therefore, those users running a common application. Therefore, a group and an application are considered as one entity. It is not possible to have a group

running multiple applications. A user can belong to multiple sessions, and thus potentially, run multiple applications. Figure 4.2 illustrates a user involved in multiple sessions.

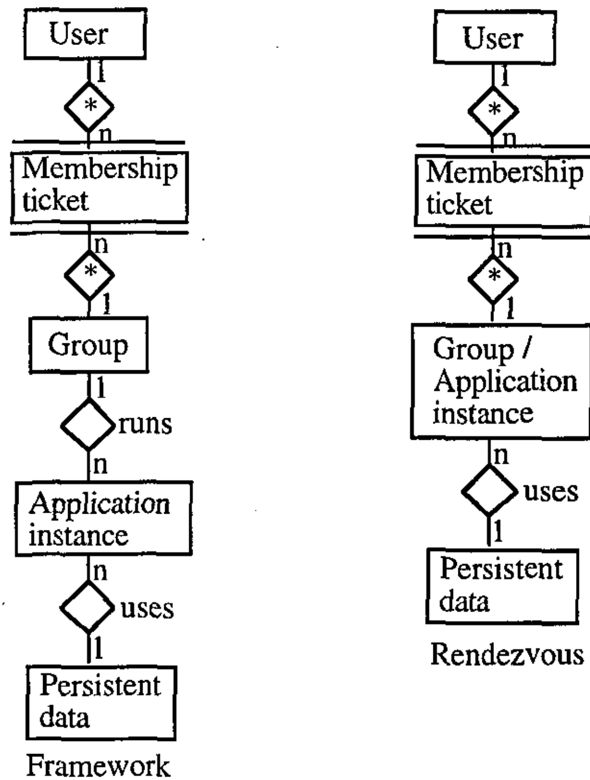


Figure 4.1 Rendezvous Session Management

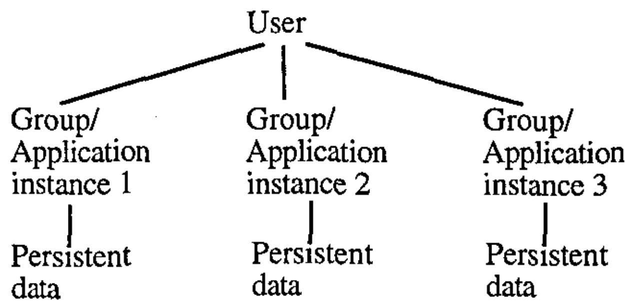
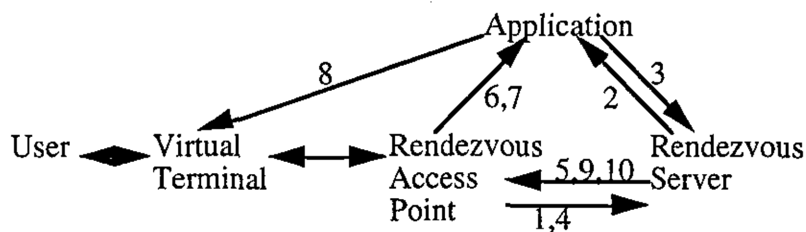


Figure 4.2 Multiple Rendezvous Sessions

The membership ticket entity must exist in Rendezvous, although it is not made explicit. The membership ticket entity enables a group, or application in this case, to know which user is interacting with it and enables a user to know which group, or application in this case, it is interacting with. An X windows virtual terminal, such as xterm, supports the first of these in Rendezvous. Most of the Rendezvous applications

only support one window on a display, so there is no confusion as to which application it belongs. Brinck and Hill (1993) developed a graphical editor which has a drawing board in one window and a set of tools in another. Information for discerning the windows that belong to a particular application is not available. Rendezvous makes a list of active applications available on request but maintains no knowledge about applications' constituent members, information that is commonly used to support group awareness.

To discuss the session management transitions, it is necessary to analyse the communication that occurs in Rendezvous at start-up, as described in section 2.3.1. Figure 4.3 illustrates the communication that takes place between the user, application, virtual terminal, Rendezvous access point and Rendezvous server.



1. Request to start application
2. Start application
3. Register channel address
4. Request channel address
5. Send channel address
6. Call application
7. Virtual terminal address
8. Connect
9. List of active sessions
10. Invitations

Figure 4.3 Rendezvous Start-up Communication

The communication supports a subset of the session management transitions described by the Framework. Each transition described in the Framework is examined in turn for Rendezvous. This is followed by a list of the transitions supported by Rendezvous. Some transitions are assumed to exist in Rendezvous even though they are not mentioned in the publications; it is noted where this is the case.

The session management transitions described in the Framework are:

• The "User exists" and "User does not exist" transitions are not made explicit in Rendezvous. They might be considered as the starting and stopping of a Rendezvous access point respectively.

• The "Form group" and "Dismantle group" transitions are not distinct from the "Start application" and "Stop application" transitions respectively.
• The "Start application" transition involves stages 1, 2 and 3 detailed in figure 4.3. The transition is initiated by a user who automatically joins the session. The "Stop application" transition is expected to follow similar stages to "Start application", such as: request to stop application, stop application and unregister channel address. An application cannot be suspended and so there is no state option.
• The "Add user" transition involves a changed form of stage 1, that is, request to join application, and stages 4, 5, 6, 7 and 8. The transition is initiated by a user and, therefore, a user can join a session but cannot be connected by a member. The "Remove user" transition is expected to perform the reverse of stage 8, that is, disconnect. Since a group and a running application are indistinguishable, the group identifier is filled by the application identifier. A user cannot bring an application to the group or take one from the group and so there is no application state option.
• The "Add application" and "Remove application" transitions are not supported since a group is not distinct from a running application.

As a result, the session management transitions supported at start-up are:
• User exists <user id> <user>
• Start application <appl id> <user>
    The transition subsumed is:
    • Add (first) user <user id> <appl id> <user>

During the running of a session, the transition is:
• Add user <user id> <appl id> <user>

The transitions for leaving and closing down the system are not specified in the publications. After stage 3 in the communication, the Rendezvous server registers the channel address for the application. This information is retained and enables the server to provide a list of running applications (or active sessions) on request, performed in stage 9. This information is part of the information that a membership ticket entity supports. Further, given the well-known address of each user's access point, users can receive invitations to join running applications, performed in stage 10. This is an embellished form of the "add user" transition.

Rendezvous supports a more limited sequence of transitions than that described in the Framework. For example, Rendezvous requires an application to be started before users can be added. The "natural" sequencing described in the Framework also applies. For example, a user must exist before he or she is added to the application.

Rendezvous supports dynamic binding whereby users can join and leave the application when they wish. Thus, the transition sequencing in Rendezvous is specified as:

1. User exists
2. Start application
3. Add user
4. Remove user
5. Stop application
6. User does not exist

4.2.2. Data Sharing

Patterson (1994) offers a taxonomy for characterising various architectures of real-time groupware applications. Patterson is one of the main developers of Rendezvous and, although Rendezvous is not explicitly mentioned in the taxonomy, the taxonomy characterises the Rendezvous architecture. The taxonomy is relevant to analysing the data sharing functionality of Rendezvous. This section discusses the data entities, the shared states and, finally, the data sharing transitions.

4.2.2.1. Data sharing entities and states

Patterson suggests that applications may be divided into four levels of state: display, view, model and file, as depicted in figure 4.4. The four levels of states are compared to the different classes of data described in the Framework.

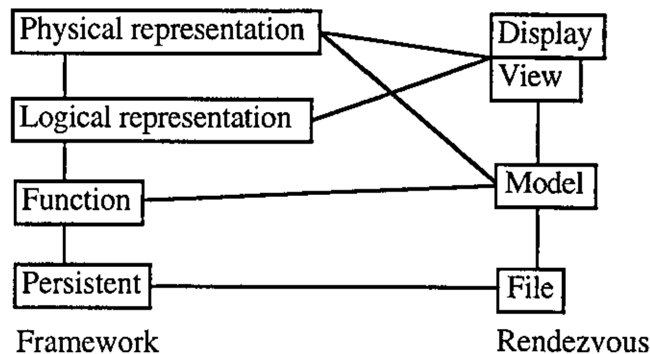


Figure 4.4 Rendezvous Data Classes

A display state is defined as information that drives the user's display. A view state is what a user sees on a display and through which a user interacts with the application. A view consists of a description of the interaction in terms of its presentation in graphics primitives, e.g. a rectangle, and user input in event handlers, e.g. a mouse

button event. The graphics primitives have attributes which include: position, offset, colour and visibility. Both a display and a view are described by physical representation data in the Framework since together they constitute the realisation of objects on the display and support for user interaction.

A logical representation class of data is not described explicitly in Rendezvous, although it must exist and is located implicitly in the view. For example, the Tic-Tac-Toe application has a logical cell array output and a choice input, which is supported by the Tic-Tac-Toe view but not described. In addition to defining a logical model of input and output devices, the logical representation class defines interaction modes, such as event, application request and application sample. Rendezvous supports user-driven event interaction but does not support application request or sample. As a result, an application cannot obtain the state of the user interface.

A model state is defined as the underlying model of the application. In Rendezvous, a *model state refers to a shared abstraction and can also contain any information, such* as display details, that are common to all users in order that the information can be shared across multiple views. A model state is described by function data in the Framework. In containing common display information, a model is described by physical representation data in the Framework. For example, the abstraction for the Tic-Tac-Toe application contains the state of each of the nine cells, which is considered to be function data, and the colour of the cells, which is considered to be physical representation data.

A file state is defined as the persistent representation of the underlying information. The application Conversation Board enables users to save and retrieve the contents of the whiteboard, and therefore, must support the notion of persistent data. A file state is described by persistent data in the Framework.

Rendezvous runs as a single process but objects within this process can be replicated to support different views. Three shared states can be derived from the four levels of state supported by Rendezvous: shared view, shared abstraction and shared file, as shown in figures 4.5, 4.6 and 4.7. These can be described by physical representation data sharing, function data sharing and persistent data sharing respectively, in the Framework. As the logical representation class of data is not supported, logical representation data sharing cannot be supported by Rendezvous. The sharing is performed on objects and, as a result, a Rendezvous application can support both physical representation and function data sharing on different objects displayed by the same application. For example, the CardTable application presents the same cards on

the table for all users to see and different card-hands to each player which only that player can see. Each shared state is discussed below.

Shared view

A shared view state is defined as one in which users see the same thing at the same time. It is described by physical representation data sharing in the Framework and shown in figure 4.5. Patterson (1994) assumes that the display state must be individualised for each user, and that the same view information drives the displays of multiple users. Therefore, partial physical representation data sharing is supported by Rendezvous.

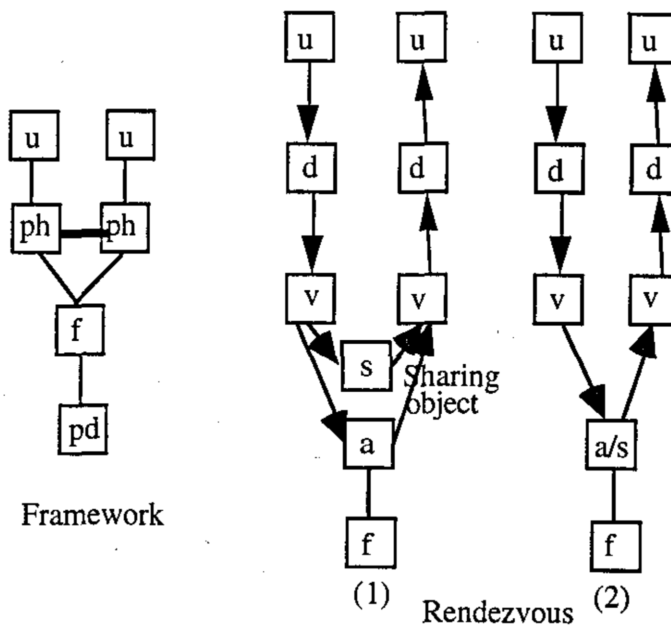


Figure 4.5 Physical Representation Data Sharing in Rendezvous

Rendezvous requires each user to have a display object and a view object, although logically there is one view object which is replicated for each user. The view objects are kept identical by placing common information either in a sharing object, as shown in figure 4.5 (1), or in the abstraction, as shown in figure 4.5 (2). The sharing object or abstraction object possess slots that mimic those of the view objects requiring WYSIWIS consistency. The slots in the view objects are constrained to be equal to those in the sharing or abstraction object. Any of the graphics primitives attributes can be put in the slots and shared, such as, position, offset, colour and visibility. For example, in Conversation Board, the position attribute is shared so that if one user moves an object on the screen, the object is re-positioned on other users' screens.

A telepointer facility is provided in Conversation Board which enables users to reference objects using individual pointers. The pointers are displayed with their owners' names on other users' displays. The location of each user's pointer must be stored in the abstraction in this application in order for it to be shared.

Multiple users can interact with replicated view objects at the same time. Because the abstraction object and the sharing object process events serially, users do not know in which order the interactions are processed, which can lead to conflicts. For example, in CardTable, if multiple players put a card on the shared pile at the same time, the order of the pack is unknown to the players, assuming that the order of the shared pile is maintained by the abstraction. The multiple actions are serialised but the users are unaware of the enforced order!

User input can be controlled in order to implement the rules of an application. The abstraction object maintains the user input policy. The Tic-Tac-Toe policy is for each user to take it in turns to play. A guard variable is set in each view object to control the user input.

Shared abstraction

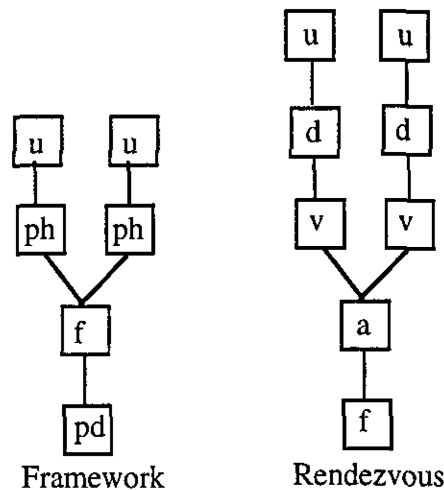


Figure 4.6 Function Data Sharing in Rendezvous

A shared abstraction state is defined as one in which users examine the same underlying information but have different, possibly customised views. It is described by function data sharing in the Framework and shown in figure 4.6. Replicated view objects can have different physical attributes. For example, CardTable displays each players own cards in the south position of the table on the screen. Concurrency control and user input control are the same as for shared view. For example, guard variables on each player's hand in CardTable prohibit opponents from manipulating a

player's hand. Although Rendezvous supports different views it does not support different interactions with the application, as described by function data sharing in the Framework.

Shared file

A shared file state is defined as one in which users may examine and manipulate the same persistent information but only after it is committed to the file or database. It is described by persistent data sharing in the Framework and shown in figure 4.7. Conversation Board is the only application that makes a reference to persistent data and it supports saving and retrieving the contents of the whiteboard. Rendezvous supports real-time application sharing and does not focus on sharing at different times.

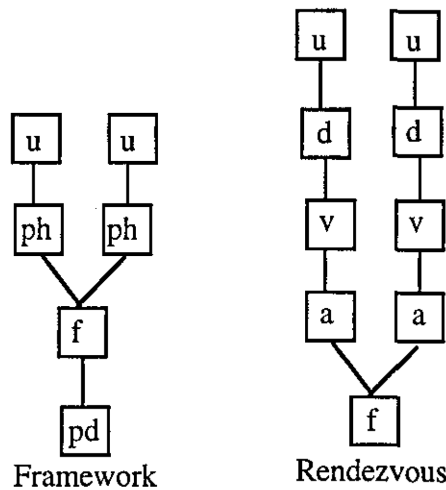


Figure 4.7 Persistent Data Sharing in Rendezvous

4.2.2.2. Data sharing transitions

Rendezvous applications support physical representation, function and persistent data classes of object, and therefore, must support the data object transitions:

- Create data object <data object id > <display object | view object | abstraction object | file object | sharing object>
- Delete data object <data object id>
- Realise data object <data object id>

Rendezvous supports aspects of physical representation, function and persistent data sharing. Therefore the "add user" transition supported by Rendezvous is:

• Add user <user id> <application id> <user>

(<physical representation | function | persistent data sharing> <object> <user sub-group>)*

The <shared state> parameter indicates which attributes are shared in addition to the type of sharing. For example, in physical representation data sharing, the following attributes might be shared: position, offset, colour, visibility and telepointer. The <sharing scope> parameter is defined per object, indicating that Rendezvous supports sharing some objects and not others. The <user sub-group> parameter is specified by roles and indicates users in the same shared state. There is no notion of a time element in Rendezvous.

In order for users to move between the different shared states, Rendezvous would have to dynamically create and remove links to a sharing object or dynamically create and remove shared values in an abstraction object, neither of which are supported. Therefore, Rendezvous does not support the dynamic change of shared state.

Rendezvous runs as a single process and is not concerned with supporting data distribution transitions. It seems possible for users to join a session once a session has begun, but it is not clear how latecomers "catch up", i.e. whether another view object is created and how that is linked to a shared abstraction. Rendezvous does not support suspended sessions.

4.2.3. Concurrency Control

Data consistency maintenance in Rendezvous is optimistic with the possibility of repairing conflict with undo operations. In addition, Rendezvous supports the control of user input in order to implement rules inherent in applications such as Tic-Tac-Toe. User input control takes precedence over the concurrency control mechanisms when it is used.

Rendezvous controls concurrency by a mixture of pre-emptive scheduling between objects and non-pre-emptive scheduling within objects. Each view and abstraction object in Rendezvous can process events pre-emptively with respect to one another or, in other words, input to multiple objects is processed concurrently. Within each object, the events are handled by a non-pre-emptive regime, that is, the processing is not interrupted. Since there are multiple view objects, multiple users can interact with their respective views at the same time. For example, players can reposition cards on the card table independently and at the same time. Since there is one abstraction object, it must process events serially.

Potentially, two types of problems can arise because users are unaware of the ordering enforced on operations carried out by an abstraction object. Firstly, the ordering of operations might be important in the application context, as in a card game. Secondly, users commonly mediate their interactions by the operations of others, and in so doing, tend to avoid conflict. However, if one user drags an object whilst another deletes it, conflict arises. Brinck and Hill (1993) indicate that, instead of locking interactions, the tools in Conversation Board attempt to do their own actions. A tool could verify that an object still exists before acting, but instead, it acts even if the object has been deleted. Therefore, the object is repositioned but does not exist. Deletion removes an object from the graphical tree but the object is stored until all pointers to it are removed. Apparently, most manipulations remain valid. Each tool implements a method for undoing its actions but it must check that the state necessary to carry out the undo is still valid. For the drag to be undone, it checks the object exists. If it does not exist, the user is presented with the option of undoing the deletion operation. Developers are required to anticipate all potential conflicts.

Patterson et al. (1990) introduces a sharing object to maintain the consistency between view objects. Since there is one sharing object, it must process events serially. Problems arising as a result of users being unaware of the ordering of operations on a sharing object are apparent to users because the operations are concerned with the physical presentation of objects. Users can correct the operations as necessary.

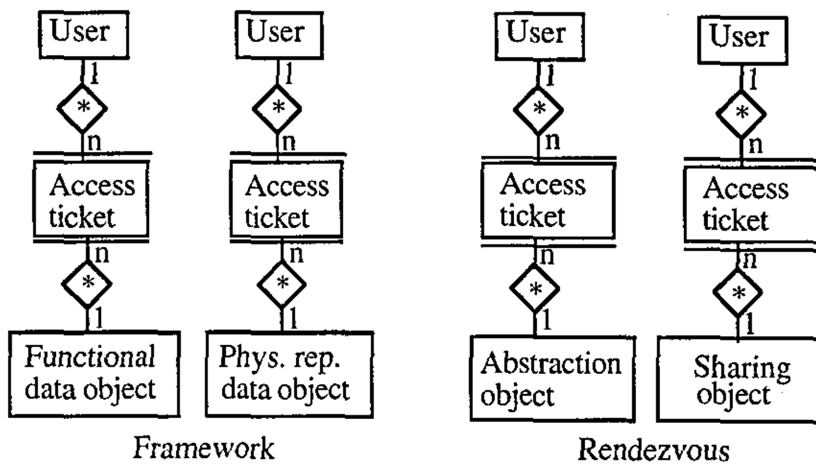


Figure 4.8 Rendezvous Data Consistency

Effectively, data consistency maintenance in Rendezvous is represented by the locking model, shown in figure 4.8. Each user has an access ticket for an abstraction

object and an access ticket for a sharing object, if one exists. The access tickets serialise the input so that each operation is dealt with in turn. Conflicts are reduced by providing affordances of shared objects to users, as in Conversation Board. When conflict occurs, Brinck and Hill (1993) indicate that each tool in Conversation Board can perform a local undo but checks on pre-conditions before doing so.

User input is controlled not to maintain data consistency but is controlled to meet the demands of the application, e.g. Tic-Tac-Toe requires each user to take it in turns to play. User input control is implemented by having a guard variable in each user's view object which is checked on receiving input. By setting the guard variable to be equal to the value of some other variable, such as the active user's view process, input from the active user is accepted and input from other users is denied. User input control is represented by the locking model, as shown in figure 4.9. A guard variable is represented by an access ticket. A view object is represented by a physical representation data object. Each user has an access ticket for a view object which allows only one user to enter input to that object.

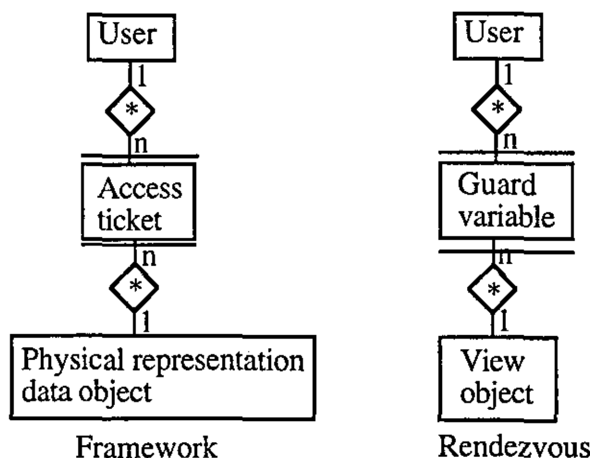


Figure 4.9 Rendezvous User Input Control

The "create consistency ticket" transition creates an access ticket for a data object. In data consistency maintenance, an access ticket is set up for each abstraction object and each sharing object and cannot be removed. Access is not denied. As a result, Rendezvous supports the transition:

- Create consistency ticket <user id> <function | physical representation data object id> <read, write>

In user input control, an access ticket is set up for each view object but the value of the guard variable will change as each user is allowed to enter input. To change the

guard variable, the access ticket is deleted and then added with the new value. As a result, Rendezvous supports the following transitions:

- Create consistency ticket <user id> <physical representation data object id> <read, write | none>
- Delete consistency ticket <user id> <physical representation data object id>

4.2.4. Comments

Section 4.2 has detailed the functionality supported by Rendezvous as expressed by the Framework. It is found that Rendezvous supports a sub-set of the functionality described in the Framework. More interestingly perhaps, particular aspects of Rendezvous and the Framework are highlighted by the analysis which are discussed here.

Rendezvous does not support the concept of a group. As a result, a group cannot support a common task by running multiple applications. Multiple applications can run but for each application the set of users must be specified.

Rendezvous makes a distinction between a display, which is defined as software that drives the workstation, and a view, which is composed of interaction objects. The software that drives a workstation is the window server. Interaction objects are built from graphics primitives which are built in X windows but without using a higher-level X toolkit or window manager. Therefore, the distinction between a display and a view seems to be based on window system software. The Framework expresses such a distinction as implementation-dependent physical representation data.

Hill et al. (1994) suggest that Rendezvous "extends the notions of dialogue independence", the separation of the user interface from the application functionality, "with one shared abstraction (the application semantics) connected to several views (user interfaces)". However, they also indicate that "the shared abstraction stores and provides access to all information that is common to all views". The abstraction represents a model of common, or shared, interaction. In Rendezvous applications, the abstraction contains both function data and shared physical representation data. Therefore, the notion of dialogue independence is not implemented in Rendezvous architecture. Patterson et al. (1990) indicate the requirement for a sharing object to support identical views which stores common display information. As a result, the abstraction would need only contain function data and the notion of dialogue independence would still apply. However, a sharing object is not implemented in any of the Rendezvous applications.

It is not immediately obvious in Rendezvous how concurrency control is supported. It might seem as if user input control is the main mechanism. However, Conversation Board does not have any user input control and is able to illustrate the underlying consistency mechanism. The consistency mechanism in Rendezvous is optimistic with the possibility of repairing conflict with undo operations. Parallel user input into replicated views is serialised at the shared abstraction. Tic-Tac-Toe and CardTable make use of user input control predominantly. User input control restricts user input and overrides the consistency mechanism.

Performance issues are not expressed in the Framework. In Rendezvous, the abstraction and all the views run as a single process. Rendezvous is referred to as having a centralised architecture but it has multiple view objects, and therefore, is more like a hybrid architecture that is centrally implemented. Users running a shared application commonly require simultaneous service. For example, if one user's display needs updating, so do others' displays. Running only one process leads to time delays in user feedback.

Rendezvous allocates users into roles, which results in users in a particular role having the same customised view. For example, CardTable has players, who can only see and manipulate the cards on the table and in their hand, and Kibitzers or onlookers, who can see all players' hands but cannot manipulate any of the cards. The Framework describes the mechanisms to support roles but does not make the support explicit. Roles are supported in data sharing by defining a common shared state for a sub-group of users (by using the <user sub-group> parameter in the "add user" transition). User interface control is supported by defining consistency tickets for each user. However, the Framework should be able to describe a sub-group of users with the same access permissions. This can done by adding a <user sub-group> parameter to the "Create consistency ticket" transition.

4.3. GroupKit

GroupKit is a groupware toolkit which is used for building multi-user, real-time applications. It focuses on supporting group awareness and uses an approach called "open protocols", which is applied to supporting session management. It is used as a major system for analysis by the Framework.

4.3.1. Session Management

Figure 4.10 illustrates the entities and their relationships supported by GroupKit as described in the Framework. A GroupKit conference is a running application. A group of users run the application but the group is not considered a distinct entity from the application. Therefore, a group and an application are seen as one entity. It is not possible for a group to run multiple applications.

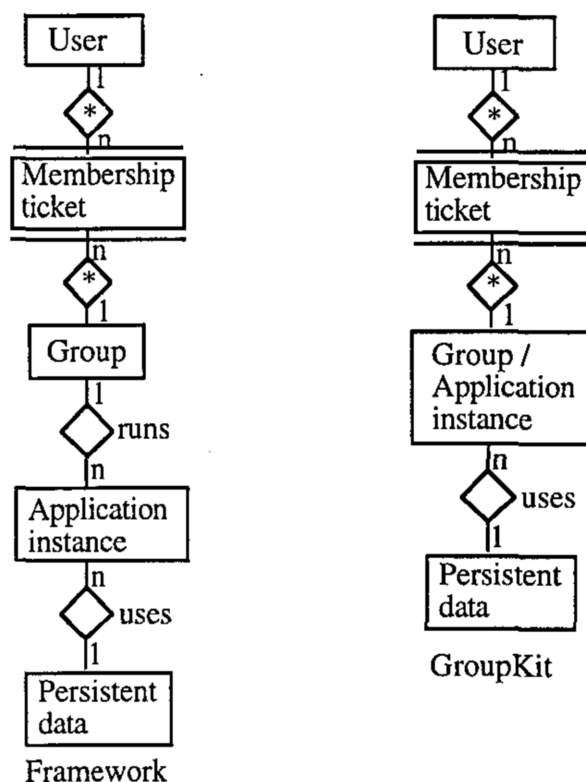


Figure 4.10 GroupKit Session Management

A user can be involved in multiple conferences. A centralised process called the Registrar maintains a list of all conferences and the users in each conference. This information is described by the membership ticket entity in the Framework. It provides participant status information. In the Framework, the membership ticket entity also enables a group, or application, in this case, to know which user is interacting with it. In GroupKit, this is supported by an X windows virtual terminal. The membership ticket entity also enables a user to know which application he or she is interacting with, but since GroupKit applications support only one window on a display, there is no need to discern groups of windows and this information need not be supported by GroupKit.

In order to discuss the session management transitions in GroupKit, it is necessary to analyse the conference registration infrastructure and protocol, as shown in figure 4.11. A box in the figure represents an individual process. A user interacts with the system via a Session Manager. Session Managers implement particular registration policies and provide user interfaces which can be tailored to each user. A Registrar process is the first process created in a session and there is typically one for a community of users. The Session Manager creates and deletes Conference objects. A Conference object actually runs the specific groupware application. It locates other users via the Registrar and opens communication with other Conference objects.

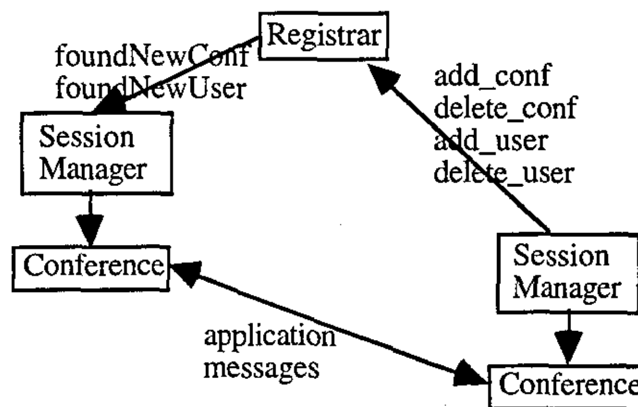


Figure 4.11 Conference Registration Infrastructure in GroupKit

To illustrate how the protocol is used to support an open door registration policy, consider starting and joining an existing session. A user starts a new conference via the local Session Manager's user interface. The local Session Manager sends an "add_conf" message to the Registrar. This updates a GroupKit environment (or shared value) in the Registrar which generates events "foundNewConf" to the other Session Managers. The local Session Manager creates a new conference object. A user can initiate joining an existing conference via the local Session Manager's user interface. The Session Manager sends an "add_user" message to the Registrar. The Registrar updates a GroupKit environment and generates events "foundNewUser" to the other Session Managers. The Session Manager also creates a new Conference object which opens communication with other Conference objects. The protocol can be used to support other types of registration policy.

The session management transitions identified in the Framework are compared with the primitives supported by GroupKit:

• The "User exists" and "User does not exist" might be considered as the starting and stopping of a Session Manager respectively.

• The "Form group" and "Dismantle group" transitions are not distinct from the "Start application" and "Stop application" transitions respectively.
• The "Start application" transition is supported by the "add_conf" primitive. The transition is initiated by a user who automatically joins the conference. The "Stop application" transition is supported by the "delete_conf" primitive. A conference cannot be suspended and so there is no state option.
• The "Add user" transition is supported by the "add_user" primitive. It is not used for the user who invokes a conference. The transition is initiated by the user, and therefore, a user joins a conference and is not connected. The "Remove user" transition is supported by the "delete_user" primitive, and again initiated by the user. A user cannot bring an application to the group or take one from the group and so there no application state option.
• The "Add application" and Remove application" transitions are not supported since a group is not distinct from a running application.

The session management transitions supported by GroupKit are:
• User exists <user id> <user>
• Start application <appl id> <user>
    The transition subsumed is:
    • Add (first) user <user id> <appl id> <user>
• Add user <user id> <appl id> <user>
• Remove user <user id> <appl id> <member>
• Stop application < appl id> <member>
• User does not exist <user id> <user>

GroupKit supports a sequence of transitions which restricts the sequence described in the Framework. For example, GroupKit requires an application to be started before users can be added. The "natural" sequencing described in the Framework also applies. For example, a user must be exist before he or she is added to the application. GroupKit supports dynamic binding whereby users can join and leave the application when they wish. Thus, the transition sequencing in GroupKit is specified as:

1. User exists
2. Start application
3. Add user
4. Remove user
5. Stop application
6. User does not exist

4.3.2. Data Sharing

Data objects, other than overlays, are scarcely mentioned by the GroupKit publications. However, they are discussed in relation to GroupDraw, a precursor to GroupKit (Greenberg et al., 1992). It is assumed that GroupKit considers data sharing issues in a similar way to GroupDraw. Firstly, the data classes are discussed, then the shared states, and finally the data sharing state transitions.

The data classes represented in GroupKit are compared to the Framework and shown in figure 4.12. GroupKit considers meta representation data, which includes telepointers and graphical annotation, to be distinct from application data. This separation has the advantage that the meta representation data does not interfere with the underlying application graphics and the disadvantage that the meta representation data is only used temporarily to inform the work surface. GroupKit provides each user with a telepointer which can be represented by a variety of different icons and customised for each user. The Framework considers meta representation data to be part of the physical representation data.

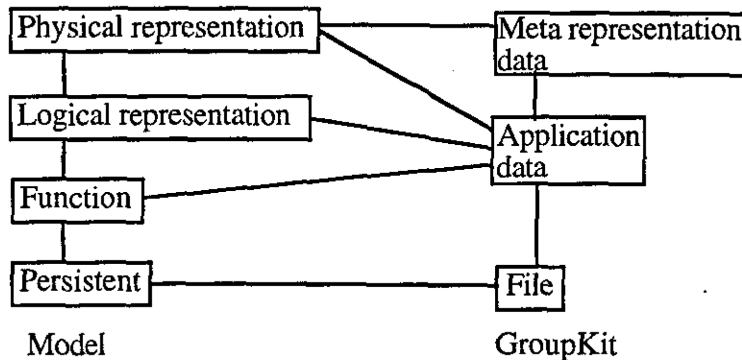


Figure 4.12 GroupKit Data Classes

GroupDraw specifies an abstract graphical object that can be subclassed into concrete objects such as shared lines. A concrete object is defined, along with methods for graphically manipulating and drawing the object, and methods for changing the object attributes. Instances of concrete objects are considered to be application data objects. Therefore, an application data object incorporates logical representation data, function data and aspects of physical representation data, and no distinction is drawn between them. GroupDraw enables an image of the work surface to be stored as a file and retrieved. The file data is equivalent to persistent data in the Framework.

GroupKit supports a replicated architecture in which each user has a local copy of the application running as a separate process. The processes communicate with one another using multicast remote procedure calls and events. GroupKit supports one shared state, physical representation data sharing, as shown in figure 4.13. As GroupKit does not distinguish between physical representation data, logical representation data or function data, logical representation data sharing and function data sharing cannot be supported. Further, although persistent data exists in GroupDraw, it is stored locally and not shared, and therefore, persistent data sharing is not supported. The state applies per data object and, as a result, some data objects are shared whilst others are not.

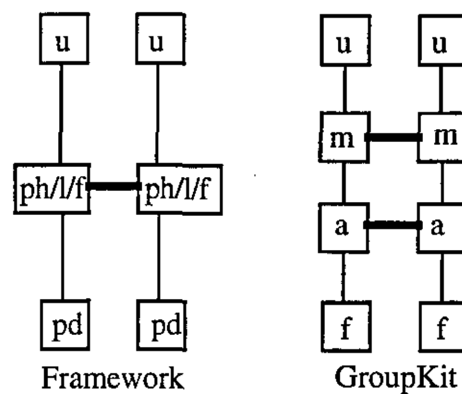


Figure 4.13 Physical Representation Data Sharing in GroupKit

GroupKit supports WYSIWIS views within a conference window, which includes: the same application graphics, the same user input and telepointers for all users. Since GroupKit is implemented in the X window system and there is no mention in the publications of sharing window management functionality, it is assumed that window management functions are not shared, e.g. window size. Data consistency is not maintained in GroupKit, refer to the next section for details.

In order to support private drawings, GroupDraw provides a scrollable drawing surface; a user can scroll and work in their own area of the screen, and then move the image to the main view. Each user's display has the same view but a possibly different window on that view, refer to figure 4.14. An attribute of physical representation data sharing is users with the same scroll position so when the scroll position is not shared, physical representation data sharing is relaxed.

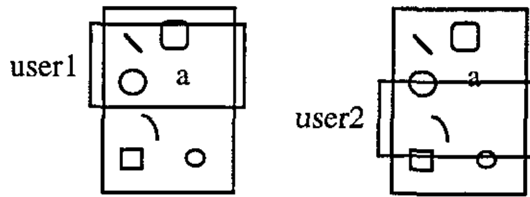


Figure 4.14 Different windows on the same view

Each object in GroupDraw has a "coupling status". A user can indicate whether an object is shared, in which case it can be manipulated by all others ("all of us can see and touch it"), public, in which case it can only be viewed by others ("I can touch it but you cannot"), or private ("only I can see it"). If an object is shared, then it is in a physical representation data shared state. If an object is public, it is in a physical representation data state but with restricted user access. If an object is private, then it is said to be in a private working state. The private working state allows a user to construct a private drawing.

GroupKit applications support the following data sharing transitions:

- Create data object <data object id><meta representation data object | application data object | file>
- Delete data object<data object id>
- Realise data object <data object id>

GroupKit supports aspects of physical representation data sharing, and therefore, the "add user" transition supported by GroupKit is:

- Add user <user id> <appl id> <user>
  (<physical representation data sharing><object>)*

The <shared state> parameter indicates what attributes are shared in addition to the type of sharing. In GroupKit, these include application output, user input and telepointers. The <sharing scope> parameter is defined per object. There is no notion of a user sub-group or a time element in GroupKit. Since GroupKit supports only one shared state, users remain in the same shared state.

Since GroupKit supports a replicated architecture, it must support data distribution mechanisms. Therefore, it supports the transitions:

- Create data object <data object id> <class> <location>
- Copy data object <data object id> <location> <new data object id> <new location>

GroupKit does not support suspended sessions but does support latecomers. GroupKit's run-time infrastructure automatically generates an "updateEntrant" event when a latecomer joins a conference that is already in progress. The event is sent to only one application process in the session which sends its existing state to the new application process. For example, in the TextChat application, the text contents of all existing chat windows are passed to the new process. Therefore, the transition "Create data object" must be enhanced to:

- Create data object <data object id> <appl data object> <location>
  <data object state>

The <data object state> parameter contains the contents of the data object to be passed on.

4.3.3. Concurrency Control

GroupKit aims to support real-time applications in a replicated architecture. However, GroupKit provides no concurrency control mechanisms to synchronise the replicated objects and allows the replicas to get out of step with one another. It is believed that concurrency control needs are highly application dependent and no one mechanism suffices. For example, an undo mechanism is often combined with knowledge of the application semantics which cannot easily be supported in a generic toolkit. Therefore, GroupKit supports optimistic data consistency, as described in the Framework. GroupKit's applications depend on social protocols to minimise conflict and do not undermine an application's responsiveness by paying the performance penalty of concurrency control.

GroupKit must support the consistency ticket entity because information about users' access to data objects is used by location awareness widgets. Therefore, the consistency ticket entity exists but does not contain mechanisms to control user access. Concurrency control in GroupKit is represented in figure 4.15. A consistency ticket is created for each data object and cannot be removed. Access to a data object is not denied.

GroupKit must support the transition:

- Create consistency ticket <user id> <data object> <read, write>

As an option, GroupKit application developers can serialise input by directing changes through a single application process or through a separate, dedicated process.

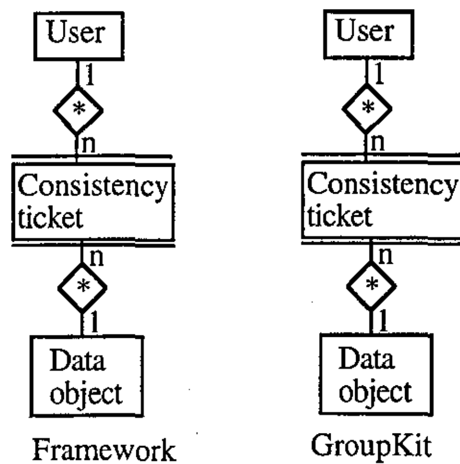


Figure 4.15 GroupKit Data Consistency

4.3.4. Comments

Section 4.3 has detailed the functionality supported by GroupKit as expressed in the Framework. It is found that GroupKit supports a sub-set of the functionality described in the Framework. The following discusses aspects of GroupKit highlighted by the analysis and reflections on the Framework.

The "open protocols" approach offers support for tailoring applications to groups and to individual users. Their main limitation is the need for a designer of an open protocol to decide at the start the state changes and the protocol that initiates them. Without a consistent and comprehensive description of user functionality, the designer is seen to be predicting the system's future use. However, the Framework offers such a description and some of the protocols map to transitions described by the Framework. Therefore, the Framework could provide the functional base on which the state changes and protocols are developed in GroupKit. In addition, open protocols illustrate how transitions described by the Framework might be applied to support the construction of a groupware toolkit.

GroupKit is noted for its support for user awareness. Principally, awareness is supported by three classes of groupware widget: participant status, telepointers and location awareness. The Framework's description of membership tickets indicates that they contain information that could be used to create a participant status widget. Telepointers enable users to share individual pointer positions, and thus, support gesturing. They are described by physical representational data sharing in the Framework. Location awareness widgets tell participants where others are working on a shared work surface and include multi-user scroll bars and gestalt viewers.

Information in the Framework's consistency ticket indicates users' access to objects and could be used to form location awareness widgets. Therefore, awareness information is described in the Framework but the Framework does not describe how the information is presented and made available to users.

4.4. Dialogo

Dialogo exemplifies a shared window system. A shared window system enables existing single-user applications to be shared in a group context. It provides a collaboration-aware environment in which collaboration-transparent applications are run and made available to the group. A shared window system is quite different to groupware toolkits, such as Rendezvous and GroupKit, which support the construction of collaboration-aware applications. Once a shared window system is built, it is intended to support most existing single-user applications. Dialogo is analysed in terms of the Framework.

4.4.1. Session Management

When applications are collaboration-transparent and the group-work support is embedded in a separate entity, the Framework takes its extended form. Thus, the description of the Framework containing a groupware application environment is appropriate in the case of a shared window system. In Dialogo, the conferees' agents, the conference manager, the conference secretary, the telepointing application and the shared window manager together constitute a groupware application environment. Figure 4.16 illustrates the entities and their relationships supported by Dialogo as described by the Framework. Dialogo is shown to support exactly the same entities and nearly the same relationships as the session management model described in the Framework.

A Dialogo conference is effectively the same as a groupware application environment described in the Framework. Each user may participate in any number of conferences simultaneously and each conference may run any number of applications. Therefore, a user can run any number of application environments and an application environment can run any number of collaborative-transparent applications. A group constitutes those users who are in a conference or an application environment. A group of users are specified using the conference secretary before a conference is started and therefore a group is distinct from a conference. A user could be in any number of groups but a group runs only one conference. Therefore, there is a one to one relationship between the group and application environment entities.

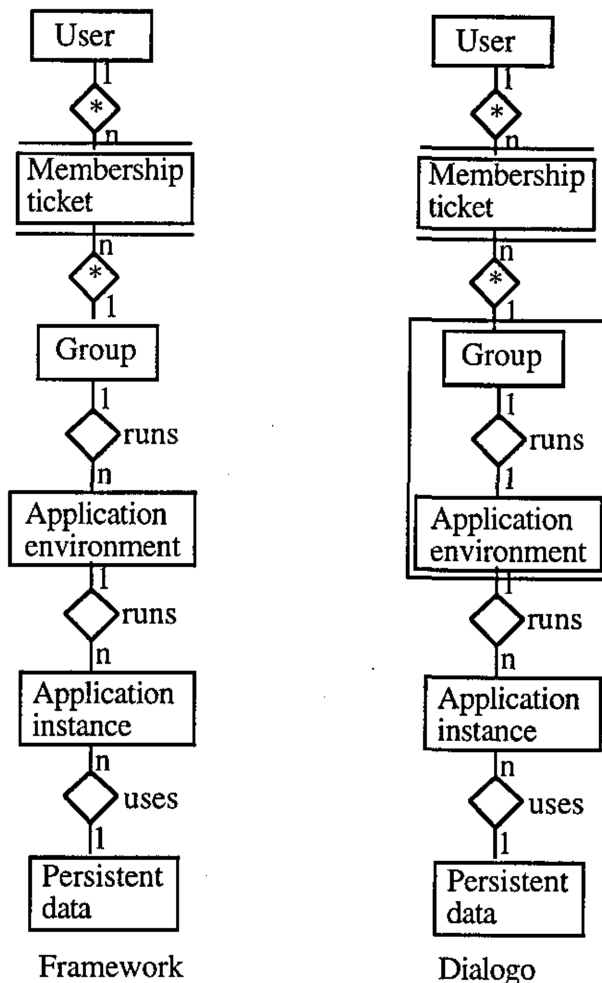


Figure 4.16 Dialogo Session Management

The membership ticket entity must exist in Dialogo, although it is not made explicit. The membership ticket entity enables a conference, or application environment, to know which user is interacting with it and enables a user to know which conference it is interacting with. The virtual terminals supported by X windows enable an application, and thereby a conference, to know which user is interacting with one of its windows. Distinguishing between windows in different conferences and public and private windows are considered features of workspace management by shared windows systems. Dialogo adopts the rooms metaphor to group together windows in a conference and implements it by running an existing window manager as a shared application. Therefore, Dialogo uses the window system to support aspects of the membership ticket entity. The conference secretary maintains a record of which user is in which conference but although this could be used to provide status information, it is not used by the conferences or applications.

The session management transitions described in the Framework are compared with the functionality supported by Dialogo:

• The "Form group" transition occurs when the initiator registers a new conference with the conference secretary. The transition might include a list of files that need importing into the conference. Dialogo does not support suspended conferences and so no group state parameter is specified.

• The "User exists" transition occurs when the initiator specifies a prospective participant with the conference secretary.

• The "Add user" transition occurs for each user when they accept the invitation from the conference secretary to be connected to the conference. This transition involves the conference secretary sending out invitations, tracking the acceptances, and creating a directory and replicating files for users that do accept. Users cannot bring a running application into the conference and so no application state parameter is specified,

• The "Start environment" and "Add environment" transitions occur when a real-time session is started by a conference member. The first application, a shared user interface shell, is also created. Dialogo does not support suspended conferences and so no environment state parameter is specified.

• The "Start application" and "Add application" transitions occur when an application is started through the shared user interface shell by a conference member. Dialogo does not support suspended conferences and so no application state parameter is specified.

• Presumably there is a way of ending a conference and stopping an application, though this is not specified in the publications.

The session management transitions supported by Dialogo are:

• Form group <group id> <user>

• User exists <user id> <user>

• Add user <user id> <group id> <user>

• Start environment <env id> <member>

    The transitions subsumed are:

    Add environment <env id> <group id> <member>

    Start (first) application <appl id> <member>

    Add (first) application <appl id> <env id> <member>

• Start (next) application <appl id> <member>

    The transition subsumed is:

    Add (next) application <appl id> <env id> <member>

The conference secretary supports the transitions: "User exists", "Form group", "Add user" and "Start environment". The first application, a shared user interface shell, supports the "Start (next) application" transition.

In order to fully describe Dialogo, it is necessary to also examine the sequencing of transitions. Most of the sequences are apparent, for example, the "Start application" transition must come after the "Start environment" transition in order that an environment is active before any applications are running. However, some of the sequences permitted by Dialogo are less apparent, for example, the "Add user" transition must come before the "Start environment" transition which indicates that Dialogo does not support latecomers. The transition sequencing in Dialogo is specified as:

1. Form group
2. User exists
3. Add user
4. Start environment
5. Start (next) application

The limitations of Dialogo in terms of session management are apparent from the transitions that are subsumed, the parameters that are not supported and the restrictions on sequencing. These range from broader limitations, such as, not supporting suspended conferences or latecomers, to more detailed limitations, such as, only enabling a member of the group to start a conference.

4.4.2. Data Sharing

Dialogo supports single-user applications, which provide the application functionality, a window system, which supports the user displays, and an application environment which provides the group-work support functionality and consists of the conferees' agents, a conference manager, a conference secretary, a telepointing application and a shared window manager. This section discusses the data classes, the shared states, and finally the data sharing transitions.

The data classes represented in Dialogo are compared to the Framework and shown in figure 4.17. Data objects within the single-user applications, the window system and the application environment, support all classes of data object described by the Framework. However, there is no clear distinction between the different classes of objects. Because Dialogo is implemented in X windows, the physical representation

data class can be divided into data supported by the X display server, the X window manager and X client data, but it is an implementation-based distinction.

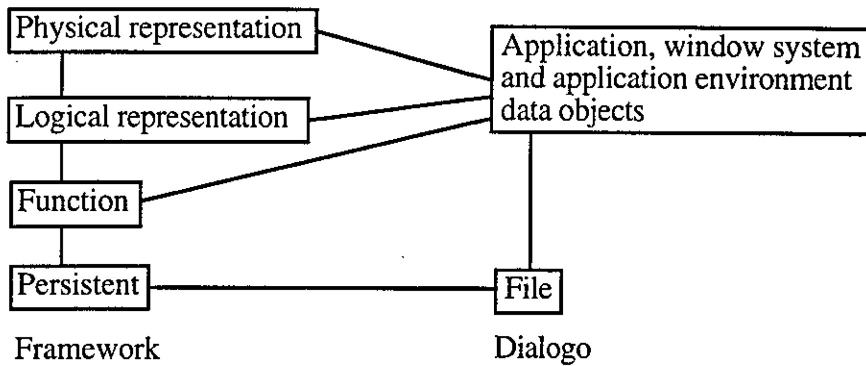


Figure 4.17 Data Classes in Dialogo

Dialogo supports a replicated architecture and WYSIWIS views. Effectively, this is described as physical representation data sharing in the Framework and represented in figure 4.18. User input is distributed by the conferees' agents and application output is delivered by the local replicated application. Strict WYSIWIS views are supported by sharing an existing window manager as a single-user application, which ensures that window management functions are shared. A telepointing application supports a shared telepointer. The only aspect of Dialogo which is not shared is user input at the physical representation data sharing level. For example, if one user enters a command in the form of a text string, other users see the effect of that command but do not see the text string on their displays.

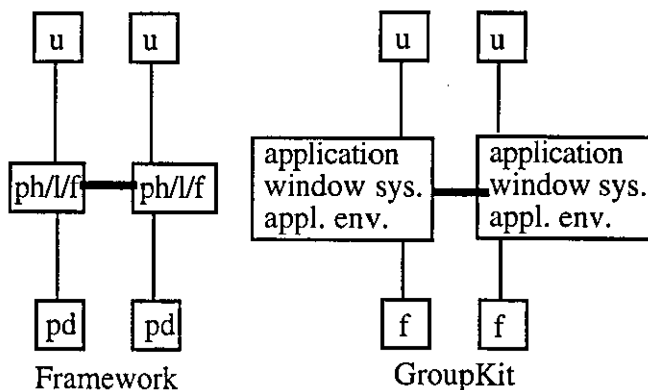


Figure 4.18 Physical Representation Data Sharing in Dialogo

Replicated shared window systems require that files are synchronised before the commencement of a conference. The conference secretary in Dialogo takes this action. In effect, this is a very limited form of asynchronous persistent data sharing.

Other shared window systems support a more relaxed form of physical representation sharing. For example, MMConf (Crowley et al., 1990) supports different users with shared windows which maintain the same size but can take different positions on the display.

Dialogo must support the data object transitions:

- Create data object <data object id> < appl object | window object | appl env object | file object>
- Delete data object <data object id>
- Realise data object <data object id>

Shared window systems do not support anything other than physical representation data sharing. This is described by the "add user" transition:

- Add user <user id> <group id> <user> <physical representation data sharing> <appl env>

The <shared state> parameter also indicates what attributes are shared, which in Dialogo include: application output, window management functions and a telepointer. The <sharing scope> in Dialogo is applied to all data objects in a conference, which is equivalent to all data objects in an application environment. This results from sharing an existing window manager with a single input focus in a conference. Shared window systems that do not share an existing window manager will have a different sharing scope, e.g. all data objects in an application. There is no notion of a user sub-group with the same shared state or a time element.

Since Dialogo only distributes user input, and application output is delivered by the local copy of the application, data distribution mechanisms which handle the replication and location of data objects are not required.

Dialogo cannot assume that single-user applications store their state and are able to transfer their state. Therefore, Dialogo cannot support: suspended conferences, latecomers "catch up", application transfer from private to public workspace and vice versa, and the repair of an inconsistent data object.

4.4.3. Concurrency Control

Physical representation data sharing requires the physical representation data, the logical representation data and the function data to be synchronised in order to maintain data consistency. In other words, replicated copies of the application, window system and application environment software that provides the group support functionality, need to be synchronised. Since neither the window manager or the single-user applications are collaboration aware, there is no means of repairing the data inconsistencies that might arise, and so a pessimistic level of data consistency is preferred. Shared window systems ensure this level of data consistency by controlling user input so that only one user is allowed to enter input at any one time.

Dialogo limits input to one user per application environment because the window manager is shared and, as a result, only allows one input focus for all applications running in the shared environment. Shared window systems that do not share a window manager limit input to one user per application, e.g. MMConf (Crowley et al., 1990). A finer grain of locking is not possible unless the single-user applications can be decomposed into constituent data objects and mechanisms employed to ensure the data consistency of these objects.

Concurrency control in Dialogo is represented by the locking model shown in figure 4.19. The locking model indicates that a user can have access to all the data objects at any one time. Pessimistic locking ensures that only one user has access at any one time.

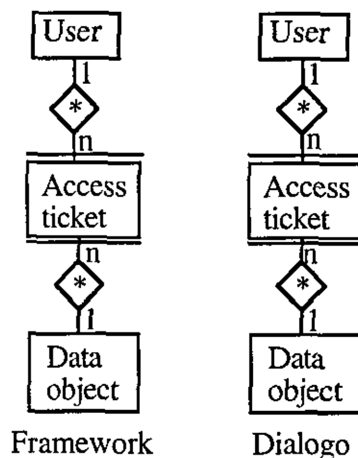


Figure 4.19 Dialogo Data Consistency

Dialogo supports the transitions:

• Create consistency ticket <user id> <appl env> <read, write I none>

• Delete consistency ticket <user id> <appl env>

An "active" user has a consistency ticket with read and write access rights. A "passive" user has a consistency ticket with no access rights.

In Dialogo, the conference manager uses these transitions to implement the explicit request floor policy. A user requests the floor through the user interface presented by the conferee's agent and the request is sent to the conference manager which performs the transitions. The consistency ticket for the currently "active" user is deleted and replaced with a "passive" consistency ticket. The consistency ticket for the user who is making the request is deleted and replaced with an "active" consistency ticket. As a result, the conferees' agents enable or disable input as necessary.

4.4.4. Comments

Section 4.4 has detailed the functionality supported by Dialogo and found it supports a sub-set of the functionality described in the Framework. In the analysis, particular aspects of Dialogo and the Framework are highlighted. These are discussed below.

The functionality of a shared window system is apparent from the analysis and includes:

• Dialogo supports a separate entity to provide the group-work support functionality.

• Dialogo supports physical representation data sharing where only the user input is not shared.

• Only one user has input access to an application at a time. Therefore, concurrency in the shared space cannot be supported.

• Individual views are not supported.

• Transfer of application data between shared and private applications is not supported.

• Latecomers and suspended sessions are not supported.

The Framework gives no indication of how group-work support might be provided. Dialogo provides group-work support by a number of different processes: conferees' agents, a conference manager, a conference secretary, a telepointer application and a shared window manager. Each process provides a particular aspect of the support. For example, the shared window manager provides shared window management functions. However, each aspect of support provided by Dialogo is implementation

based and does not easily correspond to the classes of abstract data object described in the Framework. The Framework does not determine how support is implemented.

4.5. Suite

Suite is a user interface toolkit that supports the development of multi-user, editor-based applications. The novel aspect of Suite is its coupling model which is the main aspect of Suite analysed by the Framework.

4.5.1. Session Management

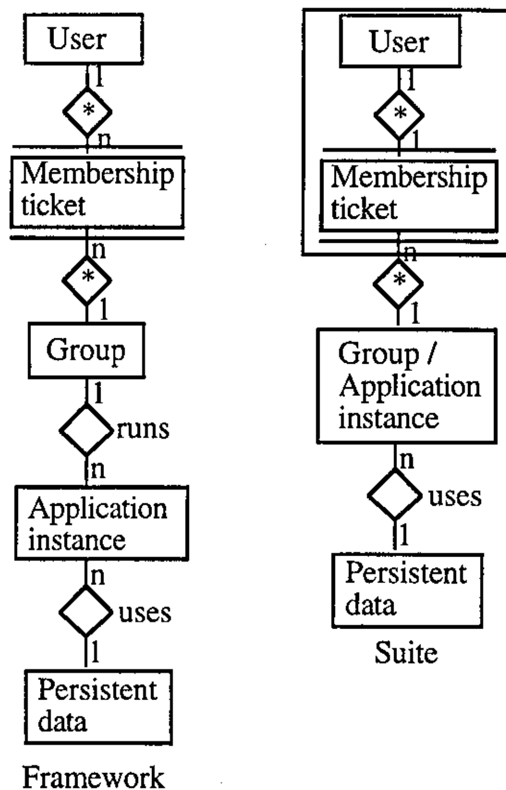


Figure 4.20 Suite Session Management

Suite provides limited support for session management. The session management entities and the entity relationships supported by Suite are compared to the Framework and shown in figure 4.20. The group entity does not exist explicitly in Suite in the same sense as in the Framework. In the Framework, a group is defined as users working on a common task, which is equivalent in Suite to users running a common application. Therefore, a group and an application are considered to be one entity. A group cannot run multiple applications. Suite supports groups but users in these groups can potentially be coupled and have interaction variables which share

common properties. Therefore, a group in Suite is equivalent to a user sub-group described in data sharing in the Framework.

The membership ticket entity is not made explicit in Suite although it must exist. The membership ticket entity enables a group, or application in this case, to know which user is interacting with it. This is provided by the window system underlying Suite. A user does not need to know which application it is interacting with since there is no indication that Suite supports multiple applications. Therefore, each user has at most one membership ticket. Membership ticket information is not used to provide group awareness amongst users.

There is no explicit discussion of persistent data in Suite although persistent data must exist.

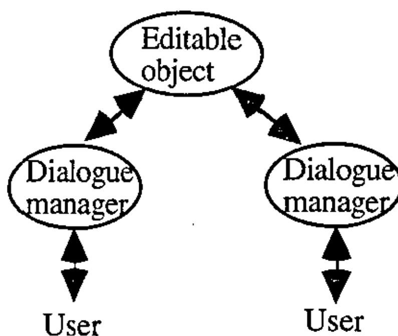


Figure 4.21 Suite Components

The session management transitions described in the Framework are:
• The "User exists" and the "User does not exist" transitions are not made explicit in Suite. They might be considered as starting and stopping a dialogue manager respectively, refer to figure 4.21.
• The "Form group" and "Dismantle Group" transitions are not distinct from the "Start application" and "Stop application" transitions respectively.
• The "Start application" transition is initiated by a user. An instance of the semantic component of an application, or the editable object, is created. There is no support for suspending an application, and thus, there is no application state parameter. It is assumed that the "Stop application" transition deletes the semantic component of an application.
• The "Add user" transition is initiated by a user. Therefore a user is said to join a session but cannot be connected by a member. A user starts a interactive session with the application whereby a dialogue manager is connected to the semantic component.

Since a group and a running application are indistinguishable, the group identifier is filled by the application identifier. A user cannot bring an application to the group, and so, there is no application state parameter. It is assumed that the "Remove user" transition disconnects the dialogue manager from the semantic component.
• The "Add application" and "Remove application" transitions are not supported since a group is not distinct from a running application.

Therefore, the state transitions supported by Suite are:
• User exists <user id> <user>
• Start application <appl id> <user>
• Add user <user id> <appl id> <user>
• Remove user <user id> <appl id> <user>
• Stop application <appl id> <user>
• User does not exist <user id> <user>

Suite supports a sequence of transitions which restricts the sequence described in the Framework. For example, Suite requires an application to be started before users can be added. The "natural" sequencing described in the Framework also applies. For example, a user must exist before he or she is added to the application. Suite supports dynamic binding whereby users can join and leave the application when they wish. Thus, the transition sequencing in Suite is specified as:

1. User exists
2. Start application
3. Add user
4. Remove user
5. Stop application
6. User does not exist

4.5.2. Data Sharing

This section discusses the data classes, the shared states and the data sharing transitions. The data classes supported by Suite are shown in figure 4.22. A shared active variable is an application variable whose value is displayed to users and changed via an interaction variable. A shared active variable is described by function data in the Framework. An interaction variable is a user's local version of an active variable and has a set of properties which determine its presentation. An interaction variable is described by physical representation data in the Framework. A window object displays interaction variables, menus and errors, and is also described by physical representation data in the Framework. Logical representation data is not

explicit in Suite, although it must exist. It is located implicitly in interaction variables. Suite must support the equivalent of persistent data in the Framework.

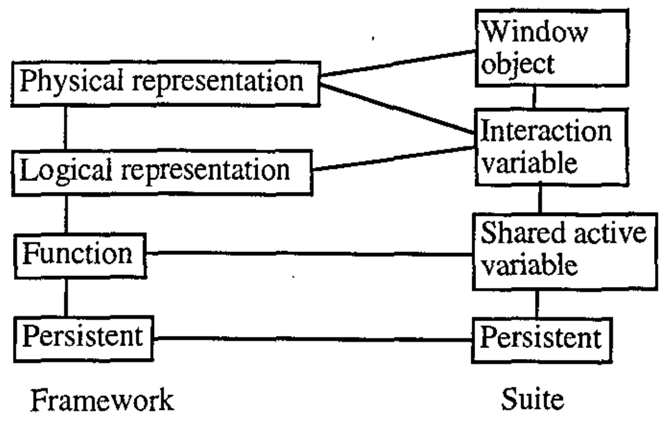


Figure 4.22 Suite Data Classes

Suite supports physical representation data sharing and function data sharing, refer to figures 4.23 and 4.24 respectively. Suite's coupling model determines the properties of physical representation data that are shared. The interaction variables and window objects are replicated for each user, and a user or a program specifies the properties that are synchronised and when they are synchronised. The interaction variables and window objects are supported by dialogue managers which exchange coupling information via the shared active variables. If none of the properties are synchronised, Suite supports function data sharing.

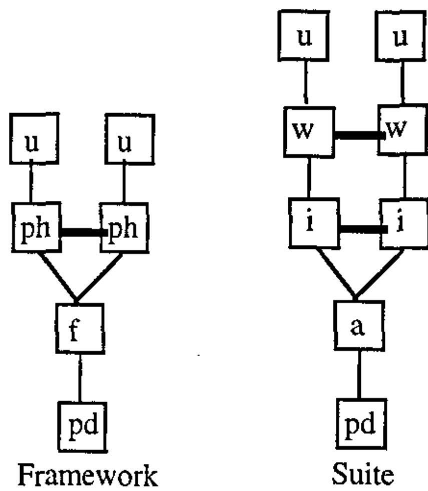


Figure 4.23 Physical Representation Data Sharing in Suite

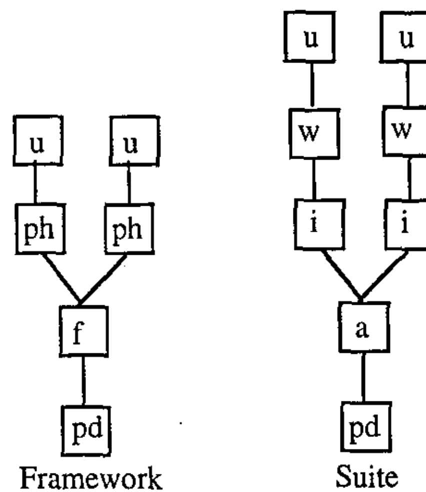


Figure 4.24 Function Data Sharing in Suite

Suite categorises the properties of an interaction variable and a window object into coupling sets and each coupling set is associated with a coupling attribute. The properties are aspects of physical representation data. Examples of coupling attributes with associated properties are given. The ValueCoupled attribute has the associated property of the value of the interaction variable, amongst others. The ViewCoupled attribute has the associated property of the visibility of the interaction variable. The FormatCoupled attribute has associated properties concerned with the presentation of the interaction variable, which include the colour of an interaction variable and an indication of whether it is represented by a text string or a filled bar. Refer to Dewan and Choudhary (1995) for a more complete description of the coupling attributes and their associated properties.

A property is shared with another user if both users have the associated coupling attribute set to true. For example, figure 4.25 shows two users sharing the properties associated the ScrollCoupled, SizeCoupled, ValueCoupled and FormatCoupled attributes. If the ValueCoupled, ViewCoupled, FormatCoupled, SelectionCoupled attributes of each interaction variable and the ScrollCoupled, ActionCoupled, PositionCoupled and SizeCoupled attributes of each object window are set to true, strict WYSIWIS physical representation data sharing is supported. If some attributes are set to false, a relaxed form of physical representation data sharing is supported. If all attributes are set to false, function data sharing is supported.

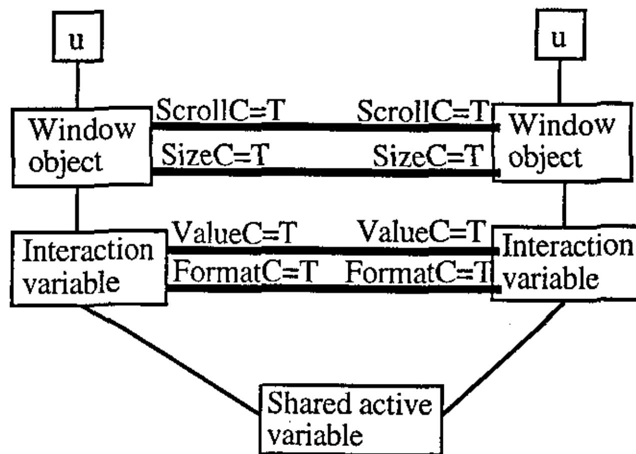


Figure 4.25 Coupling Attributes in Suite

Suite must support the data object transitions:

- Create data object <data object id> <shared active variable | interaction variable | window object>
- Delete data object <data object id>
- Realise data object <data object id>

The type of sharing and the scope of the sharing between the corresponding interaction entities of different users are specified for each user and differences between the users resolved by the system. The "Add user" transition supported by Suite is:

- Add user <user id> <appl id> <user>
  (<shared state> <value group> <user group> <when shared>)*

The <shared state> parameter specifies the value of the coupling attributes. It is described as physical representation or function data sharing in the Framework. In physical representation data sharing, aspects that might be shared include: user input, visibility, colour, format, scroll position, window size and placement. The <sharing scope> parameter is a value group, which is a group of interaction variables that stores shared attributes. The <user sub-group> parameter is a Suite user group, which specifies a particular group of users. The <when shared> parameter indicates when a change to a shared property is communicated to others.

Suite is the only system known to support the specification of sharing in such detail. It needs to be noted that the shared state specified in Suite is commonly physical representation data sharing rather than function data sharing. The specification is dynamically changeable.

Suite must support data distribution mechanisms, and therefore, must support the transitions:

- Create data object <data object id> <interaction variable | window object>
  <location>
- Copy data object <data object id> <location> <new data object id> <new location>

Suite does not support the storage and transfer of object state. Thus, it does not tackle the issues of helping latecomers "catch up" with the current state of the application, suspending an application and bringing a running application to the group.

4.5.3. Concurrency Control

Suite enforces the following rule in order to maintain data consistency (Dewan & Choudhary, 1992). A shared active variable is locked when a user starts modifying its corresponding local interaction variable and unlocked when the user commits its value. When an active variable is locked by a user, other users are not allowed to change their corresponding local interaction variables. This rule can be represented in part by the locking model shown in figure 4.26. The data object is an interaction variable. The locking model indicates that any user has access to any data object. Suite supports pessimistic data consistency whereby only one user can access a particular interaction variable at any one time. In the Suite implementation, a dialogue manager locks and unlocks interaction variables, and exchanges locking and unlocking information with its peers.

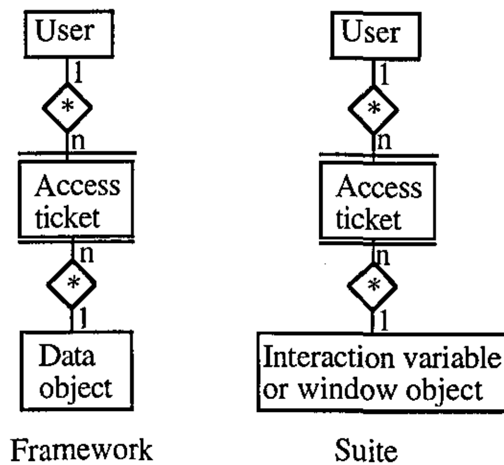


Figure 4.26 Suite Data Consistency

Therefore, Suite supports the transitions:

• Create consistency ticket <user id> <interaction variable> <read, write | none>

• Delete consistency ticket <user id> <interaction variable>

Locking can easily be confused with some aspects of coupling but they are in fact distinct. When a user changes an interaction variable, it is locked so that other users cannot change the same variable. However, the user input might still be coupled with another user so that value changes can be communicated for consultation without the value being committed. For example, if two users have the ValueCoupled attribute set to true for a particular interaction variable, then when one user changes the value of the variable, the other user can see the new value even when the change is not committed. Locking restricts both users committing a changed variable at the same time.

In addition to the policy that only one person can have access to a particular data object at a time, Suite associates each data object with a set of collaboration rights that indicate which users or operations can access a data object and assigns a priority (Shen & Dewan, 1992). The rights include traditional read and write rights and several new rights such as viewing rights and coupling rights. Suite provides programmers and users with a multi-dimensional, inheritance-based scheme for specifying and grouping collaboration rights. The AccessCoupled attribute of an interaction variable determines whether the access attributes of the variable are shared with corresponding interaction variables. If conflict occurs, conflict resolution rules examine the collaboration rights to determine who can access the object. Conflict is resolved by priority rather than repair.

4.5.4. Comments

Section 5 has detailed the functionality supported by Suite as described in the Framework. Suite supports a sub-set of this functionality. Particular aspects of Suite and the Framework are highlighted by the analysis, as discussed below.

Suite provides a near complete description of physical representation data sharing, particularly the aspects that can be shared. It also demonstrates how this sharing can be implemented, and further, illustrates how aspects can be dynamically changed. It provides a far more detailed description of physical representation data sharing than the Framework. It is not that the Framework precludes any functionality but that the functionality is not described in the same detail.

4.6. SEPIA

SEPIA is a hypertext authoring package that supports cooperation among multiple authors. Two aspects of SEPIA are of particular interest: firstly, the employment of a hypermedia-based model for structuring the cooperation amongst users, and secondly, the use of browsers and an object management system to display and manage parts of the hypermedia-based model. SEPIA is analysed with respect to the Framework.

4.6.1. Session Management

SEPIA is a groupware application rather then a toolkit or an application environment. SEPIA provides activity spaces which are supported by task-specific browsers that enable users to take part in the creation of hyperdocuments. SEPIA supports one browser per activity space per user. A browser displays the contents of a composite node. A composite node is a clustering of atomic nodes and labelled links, and is supported by an object management system. It is an organised collection of persistent data. A hyperdocument database supports the persistent data. A persistent data object can be used by several composite nodes. Since SEPIA supports multiple activity spaces, each user can have multiple browsers on different composite nodes. Figure 4.27 shows SEPIA in its conceptual and structural forms.

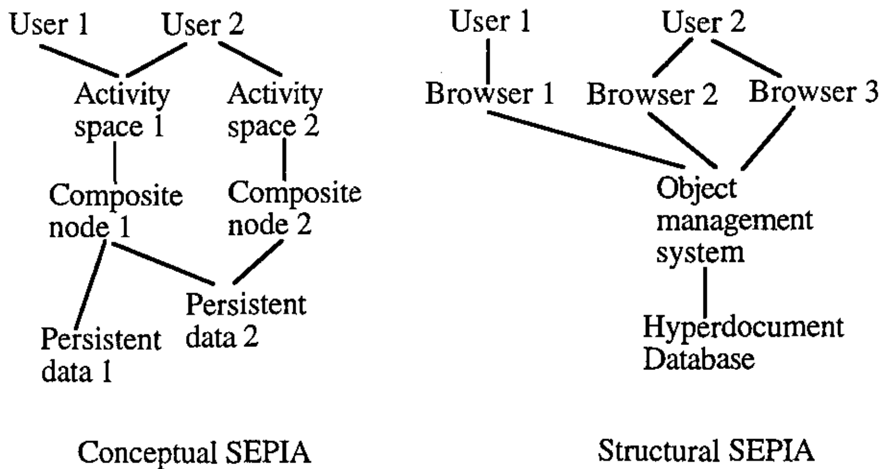


Figure 4.27 Conceptual and Structural SEPIA

SEPIA could be expressed as the application entity in the Framework and a group as constituting those users who are running SEPIA as a common application. However, in the Framework, a group is defined as several users working on a common task, and in SEPIA, an activity space is defined as task-specific. Therefore, it is appropriate to consider a group as those users working in the same activity space. In the Framework,

it was assumed that a task might be supported by multiple applications. It was not considered that a task might be supported by part of an application.

The session management entities and the entity relationships supported by SEPIA are shown in figure 4.28. An activity space is considered to be an application entity. A group is defined as those users working in the same activity space and is not distinct from an activity space. Therefore, a group and an activity space are viewed as one entity. A group cannot run multiple activity spaces, and therefore, have multiple tasks by definition. An activity space is supported by multiple browsers, one per user, which displays the contents of a composite node.

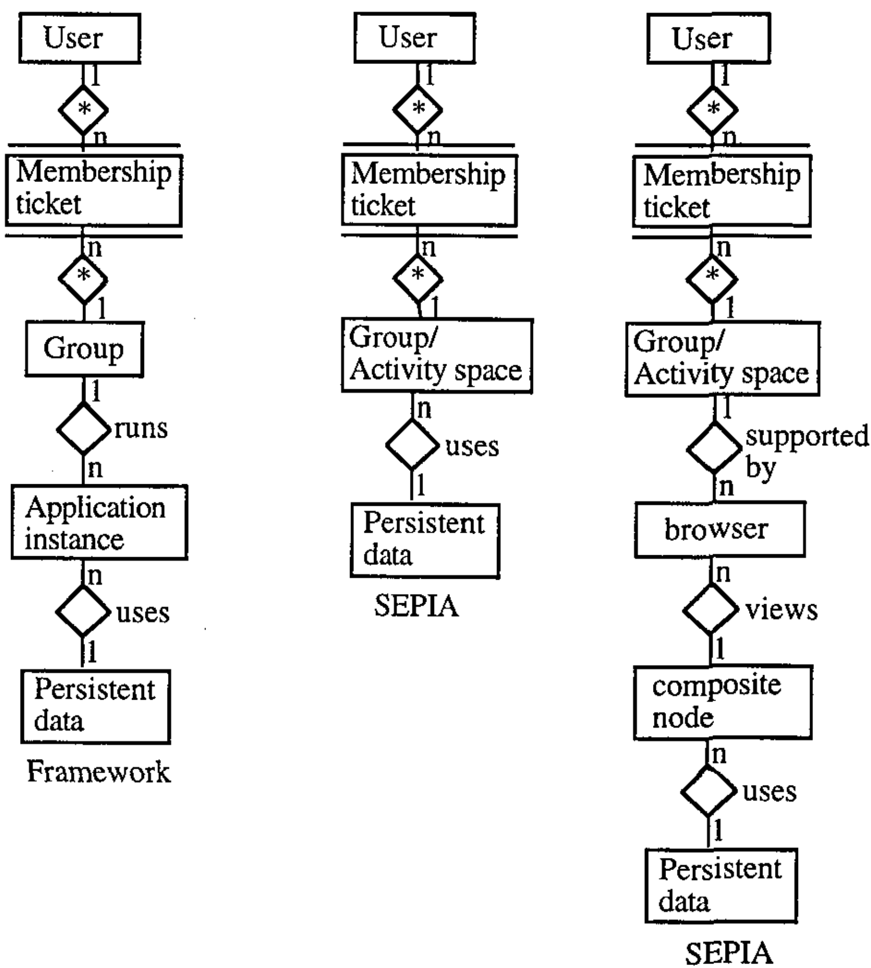


Figure 4.28 SEPIA Session Management

A user can run multiple activity spaces, one activity space per composite node. The membership ticket entity is supported by a browser. A browser enables the object management system to know that a particular user is accessing the composite node. This information is stored in the database. Further, a user is informed who else is

interacting with the node via a status line on the browser. If the users accessing a composite node change, other users' browsers are informed. Since there is only one browser per activity space, a user knows which application he or she is interacting with.

SEPIA distinguishes between three types of sharing: an individual session, a loosely-coupled session and a tightly-coupled session, as expressed in figure 4.29. The term loosely-coupled is used differently to that described in section 3.3.2 on persistent data sharing in the Framework. The relationship between a session, a browser and a composite node changes according to the type of sharing. An individual session is defined as one user working on a composite node. A loosely-coupled session is defined as multiple users working on the same composite node. A tightly-coupled session is defined as multiple users working on the same composite node and sharing the same view of the composite node. There can be multiple tightly-coupled sessions on the same composite node by different groups of authors. In addition and not expressed in figure 4.29, each user can participate in at most one tightly-coupled session per composite node at any one time. A user can have multiple activity spaces by displaying multiple browsers on different composite nodes, and as a result, might be a member of different types of session at the same time. The three types of sharing represent persistent data sharing, function data sharing and physical representation data sharing, as discussed in the section on data sharing.

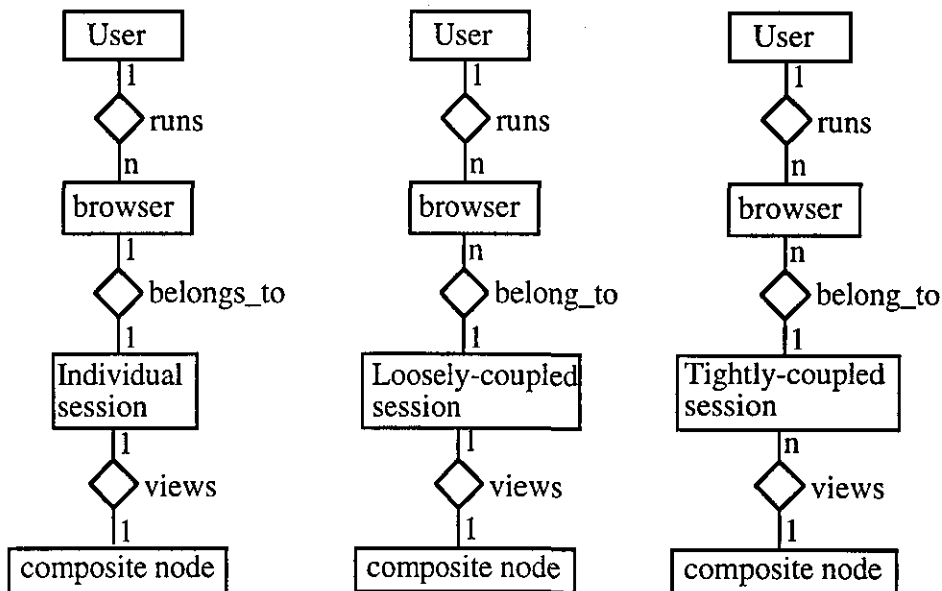


Figure 4.29 SEPIA Session Types

SEPIA's session management functionality can be expressed by the session management transitions described in the Framework. An activity space is started by invoking a browser and opening a composite node. This is expressed by the transition:

• Start application <appl id> <user>

The <appl id> parameter refers to a browser identifier and a composite node identifier. An application state parameter is not supported since an activity space cannot be suspended. The transition subsumed is:

• User exists <user id.><user>

The "Form group" and "Dismantle group" transitions are not distinct from the "Start application" and "Stop application" transitions respectively. Similarly, the "Add application" and "Remove application" transitions are not supported since a group is not distinct from a common activity space.

If a user opens the same composite node as another, a loosely-coupled session is created. This is described by the transition:

• Add user <user id> <appl id> <system> <shared state=loosely-coupled session>

Since a group and an activity space are indistinguishable, the group identifier is filled by the application identifier, which consists of a browser identifier and a composite node identifier. A loosely-coupled session is automatically set up by the system. The application state does not change when a user is added to a loosely-coupled session and so there is no application state parameter. This transition is extended to indicate the type of sharing supported is a loosely-coupled session, refer to section 4.6.2 on data sharing.

When equivalent composite nodes are opened by users, the replicated object management systems ensure synchronisation and data consistency, refer to section 4.6.2 on data sharing and section 4.6.3 on concurrency control for more detail.

Users can leave a loosely-coupled session when they wish by leaving the composite node they are working on with others. This is described by the transition:

• Remove user <user id> <appl id> <member>

The <appl id> parameter refers to a browser identifier and a composite node identifier. The application state does not change when a user leaves a loosely-coupled session and so there is no application state parameter.

A user ends working in an activity space by closing the browser. This is expressed by the transition:

• Stop application <appl id> <user>

The <appl id> parameter refers to a browser identifier and a composite node identifier. An application state parameter is not supported since an application cannot be suspended. The transition subsumed is:

    • User does not exist <user id> <user>

A user joins a tightly-coupled session from a loosely-coupled session, and so users in a tightly-coupled session are considered to be a subset of users in a loosely-coupled session. However, users cannot return to a loosely-coupled session from a tightly-coupled session and, therefore, are considered to leave a loosely-coupled session to join a tightly-coupled session. The move from a loosely-coupled session to a tightly-coupled session is expressed by the transitions:

• Remove user <user id> <appl id.> <member>

• Add user <user id> <appl id> <user> <appl state>
    <shared state=tightly-coupled session>

The <appl id> parameter refers to a browser identifier and a composite node identifier. This transition indicates that the type sharing supported is a tightly-coupled session, refer to the section on data sharing. A user might initiate a tightly-coupled session by joining it and inviting others to join, or a user might be connected to a session on accepting an invitation from the initiator. The application state of the initiator, that is the browser state, is shared amongst the members of the tightly-coupled session.

A user leaves a tightly-coupled session by closing the browser. This is expressed by the transition:

• Stop application <appl id> <member>

The <appl id> parameter refers to a browser identifier and a composite node identifier. The application state does not change when a user leaves a tightly-coupled session and so there is no application state parameter. The transitions subsumed are:

    • Remove user <user id.> <appl id.> <member>

    • User does not exist <user id> <member>

To fully describe SEPIA, it is necessary to examine the sequencing of transitions. Restrictions imposed by SEPIA in addition to the "natural" sequencing described in the Framework are highlighted below:

• A user must share a loosely-coupled session before joining a tightly-coupled session. Therefore,

Add user <u1> <a1> <system> <loosely-coupled session>

must come before

Add user <u1> <a1> <user> <appl state> <tightly-coupled session>

• All users must join a tightly-coupled session at the same time. A tightly-coupled session does not support latecomers. Therefore,

Add user <u1> <a1> <user> <appl state> <tightly-coupled session>

Add user <u2> <a1> <user> <appl state> <tightly-coupled session>

Add user <u3> <a1> <user> <appl state> <tightly-coupled session> etc.

must occur at the same time.

4.6.2. Data Sharing

SEPIA consists of three major parts: an object management system, a multi-user hyperdocument database and multiple activity space browsers. An object management system supports the data classes: atomic nodes, composite nodes, labelled links and container objects. A composite node is an organisation of atomic nodes and labelled links. A container object contains the view of the data object, such as its position, icon, style and size. Atomic nodes, composite nodes and labelled links are described by function data in the Framework. A container object is described by physical representation data in the Framework. The hyperdocument database supports persistent data. A browser provides a view of a composite node and is also considered to be physical representation data in the Framework. Logical representation data class is not made explicit in SEPIA, although it must exist. It is located implicitly in the browser and the container objects. The data classes supported by SEPIA are compared to those described in the Framework and shown in figure 4.30.

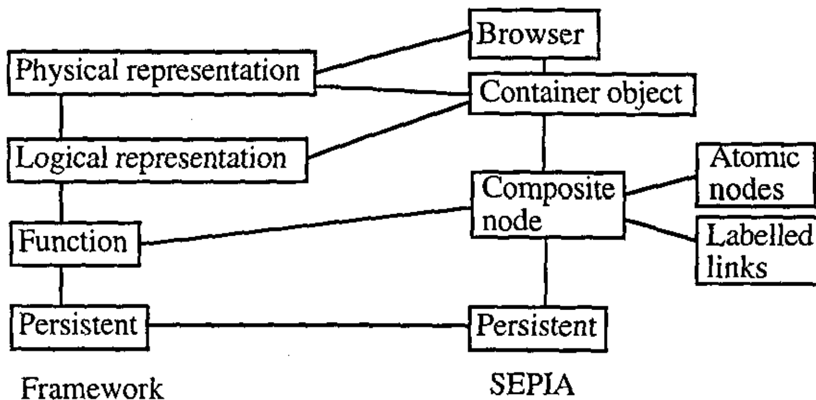


Figure 4.30 SEPIA Data Classes

SEPIA's shared states are expressed by session types. SEPIA distinguishes between three types of session: an individual session, a loosely-coupled session and a tightly-coupled session. The session types are represented by the entities: user, browser, composite node and persistent data. Each user has their own browser and own replicated copy of the composite node.

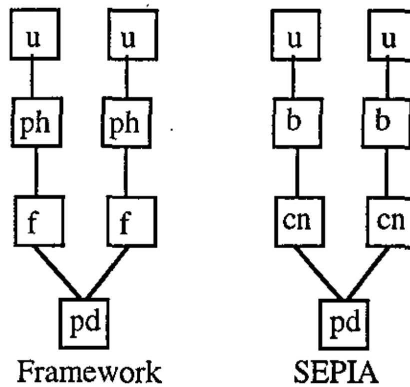


Figure 4.31 Persistent Data Sharing in SEPIA

An individual session is described by persistent data sharing in the Framework, refer to figure 4.31. Users share the same persistent data but run browsers on different composite nodes at the same time. An activity space is defined as task-specific and therefore defines a group, as discussed earlier. A composite node defines an activity space. Therefore, users running on different composite nodes are not considered part of the same group. SEPIA refers to this type of sharing as an individual session and that is exactly what it is.

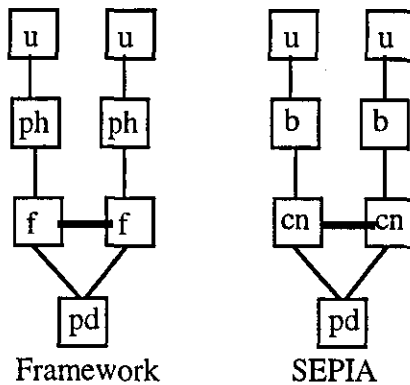


Figure 4.32 Function Data Sharing in SEPIA

A loosely-coupled session is described by function data sharing in the Framework, refer to figure 4.32. Users share the same function data but have different physical

representations, that is, different views of the same composite node. For example, users' browsers might have different scroll positions. In this state, users are kept aware of others' activities by colour-coding the objects on which users are working.

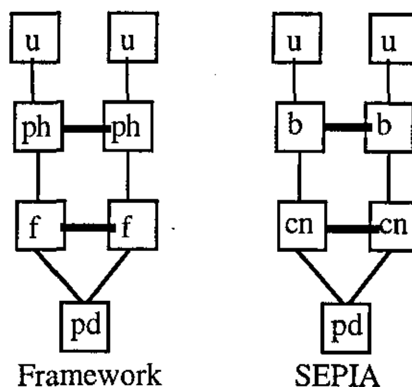


Figure 4.33 Physical Representation Data Sharing in SEPIA

A tightly-coupled session is described by physical representation data sharing in the Framework, refer to figure 4.33. In a tightly-coupled session in SEPIA, in addition to the colour-coding, visual aspects of the browser are shared. For example, users' browser windows are maintained the same size, with the same scroll position. Further, a shared telepointing facility and an audio communications facility are added on demand.

The session type is expressed in the Framework by the "Add user" transition, which is described by:

- Add user <user id> <appl id> <user | system> <appl state> <physical representation data object | function data object | persistent data object> <appl>

The <shared state> parameter also indicates what attributes are shared in addition to the type of sharing. For example, in physical representation data sharing these attributes are: application output, user input, browser window size, browser scroll position and telepointer. The <sharing scope> parameter is defined per application which is considered to be a composite node since this is what a browser views in an activity space. There is no notion of a user sub-group with the same shared state or a time element.

Users move between session types by leaving one session and joining another, as specified by the "Remove user" and "Add user" transitions and detailed in the section on session management. SEPIA focuses on supporting users performing these transitions as easily as possible.

SEPIA must support the data object transitions:

• Create data object <data object id> <atomic node | labelled link | composite node | container object | browser | persistent data>

• Delete data object <data object id>

• Realise data object <data object id>

A broadcast server handles changes to data objects, and therefore, must support the data distribution mechanisms:

• Create data object <data object id> <data object class> <location>

• Copy data object <data object id> <location> <new data object id> <new location>

When a tightly-coupled session is begun, the state of the browser is transferred from the initiator of the session to other users. This is expressed by the transition:

• Create data object <data object id> <browser> <location> <browser state>

The <browser state> parameter includes size and scroll position.

SEPIA does not support suspended sessions or latecomers to a tightly-coupled session.

4.6.3. Concurrency Control

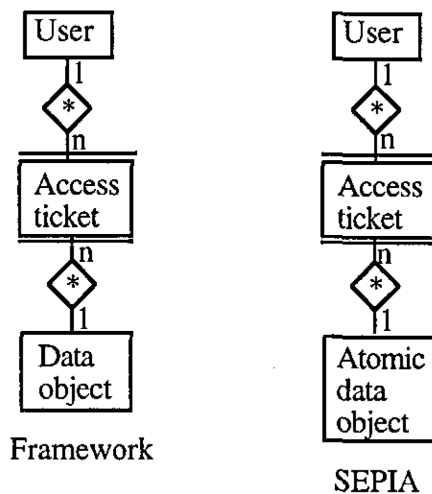


Figure 4.34 SEPIA Data Consistency

The object management system provides each atomic data object with a lock attribute which is used to synchronise access to objects in both loosely-coupled and tightly-coupled sessions. This data consistency mechanism is described by the locking model in the Framework and shown in figure 4.34. The locking model indicates that any

user has access to any atomic data object. SEPIA supports pessimistic data consistency whereby only one user can write to an atomic data object at any one time. SEPIA indicates that an object is in use by another user by colouring it red. This provides users with an awareness of other users' activities and is used as a mechanism to reduce conflict.

Locking is performed on atomic objects, and so it is possible to support concurrency on composite nodes, and thus, support parallel activities in loosely-coupled and tightly-coupled sessions. In other words, multiple users might work on different atomic objects which are organised in the same composite node.

SEPIA supports the transitions:
• Create consistency ticket <user id> <atomic object id> <read | write>
• Delete consistency ticket <user id> <atomic object id>
In order to change the consistency state, the consistency ticket is deleted and re-created.

4.6.4. Comments

Section 4.6 has detailed the functionality supported by SEPIA as expressed in the Framework. It is found that SEPIA supports a sub-set of the functionality described in the Framework. Particular aspects of SEPIA and the Framework are highlighted as a result of the analysis, as discussed below.

SEPIA is shown to support two types of sharing: function data and physical representation data sharing. It focuses on supporting users moving between these shared states in as easy and flexible way as possible. It also demonstrates how these transitions can be implemented.

It is difficult to characterise SEPIA using the Framework. This is partly because several approaches to the analysis might be adopted and none seem to fully express the functionality. The approach adopted in the analysis is to consider an activity space as an application, which has the following limitations. Firstly, the Framework considers a task to be supported by multiple applications but not by an application in part. This can result in some confusion. For example, the application identifier is the activity space identifier. The Framework needs to consider a group supported by part of an application. Secondly, it requires each user to start their own copy of the application which then performs the necessary synchronisation. Although the Framework can express this functionality, it needs to be made clearer in the

description of the Framework. Thirdly, restrictions on multiple activity spaces and intermixed session types cannot be expressed by the Framework as it is. For example, multiple tightly-coupled sessions can be run on the same composite node, providing the sessions involve different users. Because only one activity space is considered, its relation to other activity spaces cannot be expressed. The membership ticket entity in the Framework needs to be expanded to support the restrictions. An alternative approach is to consider SEPIA as an application, but then it is impossible to express a shared activity space, its session type and its constituent users in the Framework.

It is believed the difficulties with characterising SEPIA arise because of the structure of SEPIA. SEPIA behaves as if it is composed of multiple applications, or activity spaces, which are supported by three types of replicated component, with functionality between the components, the activity spaces and the encompassing SEPIA, all needing to be characterised. However, the Framework needs to be able to describe such a system and should be able to with the enhancements suggested.

It is not possible to fully express certain functionality using the Framework because it does not contain a time element. Restrictions on the sequencing of session management transitions cannot be fully described. For example, the restriction that users in a tightly-coupled session must join at the same time because SEPIA does not support latecomers cannot be fully described. In concurrency control, GroupKit, Suite and SEPIA allow any user access to any data object but allow only one user access to a particular data object at any one time, which cannot be fully described. The Framework needs to be extended to include an integrated time entity, which would enable it to express these transitions fully.

4.7. MEAD

MEAD is a groupware toolkit, specifically applicable to the command and control domains, where a shared workspace is the focus of the work. MEAD allows users to view a common information space which is a record of the state of the application. Latecomers "catch up" by simply connecting into the space. There is no concept of a group in MEAD and no support for group-awareness.

MEAD supports view and information objects, which are described by physical representation and function data respectively in the Framework. The position attribute is seen as an aspect of physical representation data in the Framework, whereas in MEAD, the position is distinguished from view, which defines the other physical representation data aspects, probably because of its importance in the domain.

Information objects, i.e. aeroplanes, exist temporarily in the information space, and therefore, are considered as function data rather than persistent data.

Physical representation data sharing is supported by the master/ surrogate agent arrangement. Local tailoring of the view and position is performed by a surrogate agent according to view and position criteria. Function data sharing is supported by the agents, which have a filtering mechanism acting on the shared information space in order to select the information objects presented to users, and act according to selection criteria. By changing the definition criteria, it is possible for users to share different objects, have different views of objects and have different placement of objects.

The update handler deals with external events and changes to the information space, i.e. changes in aeroplanes' positions. Users interact with the air traffic control prototype to change the definition criteria. This is done by each user, and therefore, data inconsistencies cannot arise.

4.8. SOL

SOL is a groupware toolkit that aims to separate the application semantics from the collaboration aware interface objects. SOL does not explicitly support any session management functionality. Concurrency control and aspects of data sharing are provided by an access control mechanism. The permission matrix in the "use" domain supports an access ticket per user, per object and per method, instead of an access ticket per user and per object, as described in the Framework. Therefore, SOL provides a refinement to concurrency control detailed in the Framework. The permission matrix in the "sharing" domain indicates whether a method for an object is shared. Shared methods are described by physical representation data sharing in the Framework, such as, sharing the position, size, visual appearance of an object and the user input. Function data sharing is supported by the policy node. SOL does not support the extent of detail in sharing that Suite provides but it does make coupling more applicable to a greater range of applications.

4.9. Summary

The Framework is used as an analysis tool to characterise each of seven systems for the group-work functionality it supports. The systems are representative of software toolkits that help in the construction of applications in group-work. The comments at the end of each characterisation indicate any interesting features of a system and any features of the Framework arising from the analysis. Chapter 5 compares the characteristics in order to find out if the Framework proves a useful comparison tool.

Chapter 5:

Comparing Systems

5.1. Introduction

Chapter 4 characterised the group-work support functionality of a representative set of software toolkits using the Framework as an analysis tool. The Framework encapsulates a set of properties that formally address the ontological and functional aspects of systems which support group-working. Chapter 4 provided a description of the set of properties supported by each system. The set of properties support a set of user behaviours. For example, if a user can be added during a running session, latecomers are supported. This chapter compares the set of properties and user behaviours supported by the systems characterised in Chapter 4. The comparison is divided into the three broad categories of functionality outlined by the Framework: session management, data sharing and concurrency control.

5.2. Session management

The systems support the same entities and similar relationships to each other, as shown in figure 5.1, with the exception of Dialogo which takes an extended form and is discussed later.

Rendezvous, GroupKit, Suite and SEPIA do not consider a group as a distinct entity from an application. The application is viewed as supporting a task rather than supporting part of a task. A group is not considered to exist outside the context of application support. Dialogo considers a group as a distinct entity but associates limited functionality with the entity. In Dialogo, a group is formed outside the context of application support and can only be supported by collaboration transparent applications running in an application environment.

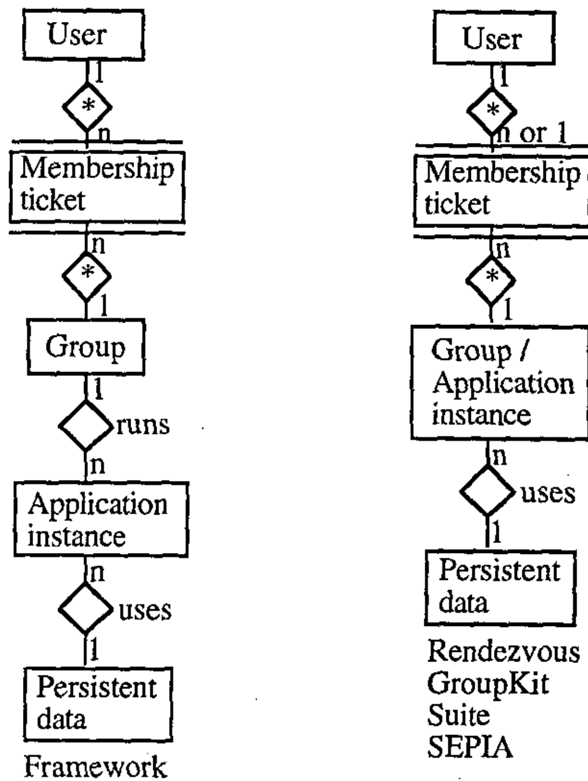


Figure 5.1 Session Management

Rendezvous, GroupKit, Dialogo and SEPIA support multiple applications. It seems that Suite supports only one application running at the same time. This is described by a one to one relationship between the user and membership ticket entities in the Framework, indicating a user can only run one application.

The membership ticket entity is not supported explicitly by any of the systems, although aspects of its associated functionality are supported. The membership ticket entity enables a group to know which user is interacting with it and enables a user to know which group it is interacting with. The first of these is commonly supported by a window system. In Rendezvous, GroupKit, Suite and SEPIA, a user knows which group he or she is interacting with since the systems support only one application or only one application window. Dialogo shares a window manager to enable users in a particular group to view and manipulate the windows supporting the group. GroupKit and SEPIA use the information described by membership ticket to provide a list of participants to the group. Rendezvous provides a list of running applications on request.

Dialogo is described by the extended form of the Framework which supports a groupware application environment entity. It is shown in figure 5.2. It is possible for

a user to belong to multiple groups, run multiple associated application environments and run multiple applications within an application environment.

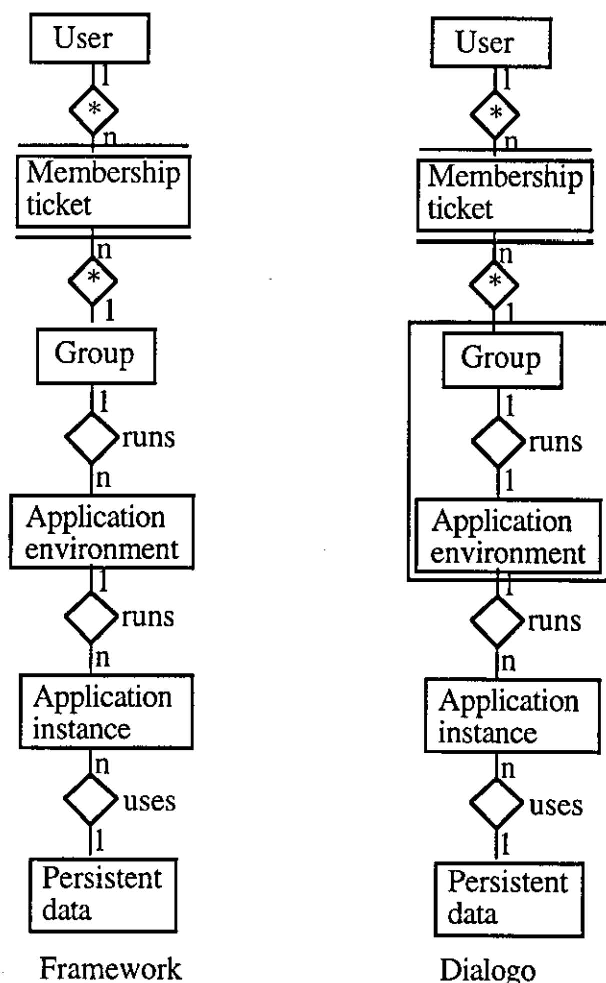


Figure 5.2 Dialogo Session Management

Rendezvous, GroupKit, Suite and SEPIA require an application to be started before users are present and start using the application. This is because the group and application entities are not distinct in these systems. Dialogo adds users to a group before applications or even the application environment are started.

When an application is started in Rendezvous and GroupKit, the user who starts the application can automatically start using the application, i.e. the first user is added automatically. Other users in Rendezvous and GroupKit and users in Suite join the application of their own accord. Users in Rendezvous and Dialogo can be invited and are connected if they choose to accept. Users in SEPIA are automatically added to a loosely-coupled session when there is more than one user of the application, i.e. more than one user working on the same composite node. If a user initiates a tightly-

coupled session in SEPIA, the user automatically joins and invites others to join. If users choose to accept the invitations, they are connected. In all the systems, users can leave an application at any time.

Rendezvous, GroupKit and a SEPIA loosely-coupled session enable users to be added to a running application and so support latecomers. GroupKit even supports latecomers to catch up with the state of the application by transferring the state from that of an existing user.

None of the systems support suspended sessions, and therefore, a long-running session cannot be broken and re-started at a more convenient time.

The initiator of a tightly-coupled session in SEPIA brings an application state to the group which is subsequently shared with other users who join the group. The application state in this case is the browser state and comprises the scroll position and size of the browser.

Four main implications for design can be drawn from the discussion on session management functionality. Firstly, by not supporting a group as a distinct entity, as in many of the systems analysed, it is not possible for a user to be a member of a group. As a result, a group cannot exist without any associated application and a group cannot run multiple applications.

Secondly, by not supporting a membership ticket as an explicit entity, user feedback is compromised. A membership ticket contains information relating users to groups and their associated applications, or if a group entity does not exist, relating users to an application. This information is used to inform users about other users in the group. Collectively, membership tickets inform users about other users in groups and their associated applications, thus providing some indication of users' focus. A user needs to know which applications are in a group and, if an application has multiple windows, which windows belong to an application and a group. Functionality can be associated to a membership ticket to support workspace management, such as facilities to handle windows in the same application and to handle different applications on the display. Further, membership ticket information is used by applications to distribute output to users and can be used to tailor output. If the membership ticket does not exist explicitly, the information must be embedded in the applications, where it cannot be modified easily, changed or re-used. Current systems support very little membership ticket information, and therefore, very little user feedback.

Thirdly, the initiator is the person who adds a user to a group and indicates whether users can join a group themselves or whether users must be connected (and possibly invited) to a group by someone else. Some systems support both, such as Rendezvous.

Fourthly, if a system does not support latecomers, all users must belong to a group from the beginning. Similarly, if the application state cannot be transferred, users must be running the application from application start-up and not from a later application state.

5.3. Data sharing

Rendezvous, Suite and SEPIA distinguish between function data and physical representation data. This enables these systems to support different views which are tailored for and manipulated by individual users. GroupKit and Dialogo aim to support the same views and so do not make a distinction between function data and physical representation data.

The physical representation data is sometimes decomposed into sub-classes based on the implementation. For example, Suite has a window object and an interaction variable which are both described by physical representation data in the Framework, and Rendezvous has both a display and a view which are described similarly. The Framework has not proceeded with further classification in order to avoid becoming implementation-dependent.

None of the systems distinguish between logical representation data and physical representation data, maybe because logical representation data is regarded as novel in user interface management and is not yet accepted in other areas. All the systems support persistent data. SEPIA supports a hyperdocument database to store the persistent data.

All the systems support physical representation data sharing. GroupKit and Dialogo strive for WYSIWIS views, whereas Rendezvous and Suite enable the application developer to choose which aspects of a view are shared amongst users. Suite in particular is distinguished by its extensive support for the different aspects of physical representation data and thus its support for different shared states.

The systems can be compared according to the aspects of physical representation data that they enable users to share, as shown in the following table:

	Rendezvous	GroupKit	Dialogo	Suite	SEPIA
Application output	√	√	√	√	√
User input	√	√	x	o	√
Colour	o	√	√	o	√
Visibility	o	√	√	o	√
Telepointer	o	√	√	x	o
Format	√	√	√	o	√
Window management functions:					
Scroll position	x	x	√	o	o
Size	x	x	√	o	o
Placement	o	x	√	o	x

√ = a shared aspect of physical representation data
o = option of sharing this aspect of physical representation data
x = an aspect not shared

Various observations can be made from this table. Suite is seen as offering the most extensive options to share aspects. Dialogo does not share users' input to applications, whereas Suite gives the option with its ValueCoupled attribute. GroupKit shares data objects but it is assumed that it does not support window management functions since these are not mentioned in the publications.

Physical representation data sharing is described in the Framework as sharing the following aspects: application output, user input , window management functions (which include scroll position, size, placement, icon state, stack position and "look and feel"), pointer position, cursor position and annotations. Several aspects of physical representation data described in the Framework are not supported by any of the systems, such as window stack position and annotations. Window stack position is probably not supported because the systems support applications with commonly only one window. Annotations are considered by GroupDraw, a system developed prior to GroupKit.

Several aspects of physical representation data sharing are supported by systems and are not described in the Framework, such as colour, format and visibility. Colour is

considered as part of the "look and feel" of an interface in the Framework, although it need not be a window management function and needs to be considered separately. Format is a structured representation of an interaction object and needs to be considered as an aspect of physical representation data by the Framework. For example, a value can be represented as a text string or a filled bar in Suite and the format indicates which it is and how it is displayed. Visibility is considered to be an aspect of function data sharing in the Framework and described as such for MEAD. However, there are applications where visibility is an appropriate option in physical representation data sharing, and therefore, the Framework should consider it as such. For example, parts of a data structure might not be visible to some users in Suite.

Rendezvous, Suite and SEPIA support function data sharing. Users' different interactions with function data result from either selecting different aspects of the application output and tailoring the presentation of the output to each user, as demonstrated by the Rendezvous application CardTable, or enabling users to carry out actions on the display which are not shared, such as scrolling the information space in SEPIA. Suite and SEPIA support function data sharing as a result of not sharing aspects of physical representation data, rather than as a positive effort to support function data sharing. Therefore, the characterisation of Suite and SEPIA as supporting function data sharing is ambiguous. Rendezvous goes some way to positively supporting function data sharing but focuses on tailoring application output rather than on supporting different interactions.

None of the systems support logical representation data sharing as a result of not identifying logical representation data as a distinct entity. Suite considers a view format which logically is an output device representing an amount, and might be represented on a display as a text string or a filled bar. No other output devices are supported in Suite and so the view format is not distinguished from an aspect of physical representation data. All the systems operate on one type of machine platform and do not run in environments with different window systems.

SEPIA is the only system to really support persistent data sharing, although the literature on Rendezvous describes it. SEPIA supports users running browsers on different composite nodes at the same time, where the composite nodes are composed of potentially the same persistent data. The hyperdocument database component of SEPIA ensures that data consistency is maintained.

SEPIA and Suite support users moving dynamically between physical representation data sharing and function data sharing, i.e. changing shared state during a running

session. SEPIA in particular focuses on making this transition as easy as possible for users. It considers two fixed states, one of physical representation data sharing and the other of function data sharing. On the other hand, Suite supports many shared states and requires users to be aware of the aspects that might be shared and to change the sharing when appropriate. Suite offers users many options but requires users to understand these options.

The scope of sharing varies between systems. Rendezvous and GroupKit consider sharing per data object. SEPIA considers sharing per application, which effectively is per composite node. Dialogo considers sharing per application environment, which is effectively all applications running in the same shared window system. Suite enables the application developer to specify the scope by defining a "value group". A "value group" is a set of related interaction variables that store attributes shared by these variables. A "value group" could comprise one data object. The shared state is specified for the defined scope, and therefore, might need to be specified a number of times, e.g. per object.

Users within a group with the same shared state and scope can be classed into a user sub-group. Rendezvous supports this concept with its use of roles, and Suite enables the system or an application developer to define users in the same sub-group.

Suite is the only system that enables application developers to indicate when a change to a shared object is communicated to others. This illustrates how sophisticated the coupling model is in Suite.

GroupKit and SEPIA support the notion of a data state. GroupKit captures and transfers the application state to latecomers to enable them to catch up with the current session. SEPIA captures the state of the browser of the initiator of a tightly-coupled session. The browser state is then transferred to users who subsequently join the session to support a group sharing the same views.

The data object transactions described in the Framework as "create data object", "delete data object" and "realise data object" do not add to the analysis and need to be reconsidered. In addition, the data distribution mechanisms do not represent the movement of data in a replicated system effectively and need further work.

Several implications for design can be drawn from the discussion on data sharing functionality. If systems support physical representation data sharing, they support aspects of the application output which are shared. If systems support function data

sharing, they support different views of the application output. Different views can result either from a more relaxed physical representation data sharing and therefore greater user flexibility, or with the intention of supporting different views of the application, such as different players in a card-game. Different aspects of the application output can be shared such as colour, format and window management functions. These impact differently on the shared context as understood by individual users. For example, if iconizing a window is not shared, it would be difficult to maintain a shared context amongst users. If the scope of the sharing is per application object, users are more likely to have different views to each other than if the scope of the sharing is for the complete application. Users need mechanisms to specify easily what aspects are shared.

5.4. Concurrency control

Dialogo is the only system that does not support concurrent access by multiple users to the same application. This is attributed to Dialogo's use of collaboration-transparent or single-user applications, and other systems use of collaboration-aware applications. In fact Dialogo limits input to one user per application environment because the window manager is shared and therefore only supports one input focus for all applications running in the same environment. Shared window systems that do not share the window manager are less restrictive and limit input to one user per application.

Suite and SEPIA are very similar in terms of concurrency control. Both support access by multiple users to an application at the same time and support a pessimistic data consistency policy per object. Thus, only one user is allowed access to an object at any one time. The pessimistic policy is enforced by locking an object when it is accessed by a user. SEPIA supports a technique to reduce conflict whereby an object is coloured red when it is in use. This enables other users to be visually aware of an object's availability. SEPIA also distinguishes between the different application operations, read and write, in terms of access rights. It is only write access which can restrict a user. Suite employs a novel technique to resolve conflict which involves setting priorities for users and application operations. When conflict occurs, conflict resolution rules examine the priorities to determine which user can access the object.

Both Rendezvous and GroupKit support access to an application by multiple users at the same time and support an optimistic data consistency policy per object. Both offer affordances of shared objects to support people's natural abilities to cooperate. In the applications built with these systems, operations are generally concerned with the

presentation of objects, and therefore, it is possible to see where a user is working and what they are doing. In addition, Rendezvous supports a sophisticated repair mechanism whereby if conflict occurs, each operation can perform a local undo but checks on the pre-conditions before doing so. GroupKit offers no repair mechanism, pointing out that the most effective mechanisms make recourse to application semantics and are not easily supported in a generic toolkit. Further, GroupKit does not suffer a performance penalty as a result of trying to maintain or repair data consistency on the rare occasions it occurs. However, the applications built with GroupKit do not require data integrity.

Both Rendezvous and Dialogo support user input control. Rendezvous supports user input control in order to implement applications that have rules involving the turn-taking of users. The emphasis is on controlling user activity rather than on maintaining data consistency. Dialogo supports user input control in order to maintain the data consistency of single-user applications. Both employ the locking model described in the Framework with a pessimistic policy which allows only one user access to an application at any one time. In Dialogo, access to an application is managed by a floor policy which is formed from a combination of consistency ticket transitions defined in the Framework and which can be changed dynamically.

If an optimistic policy is employed by a system, only a "create consistency ticket" transition is supported. If a pessimistic policy is employed by a system, both the "create consistency ticket" and "delete consistency ticket" transitions are supported.

Rendezvous, Dialogo, Suite and SEPIA support the locking model as described in the Framework. None of the systems support the serialisation model despite supporting replicated components. This is possibly because the locking model is easier to implement.

The Framework describes an access right parameter in the "create consistency ticket" transition and specifies the value as read, write or none. Only SEPIA makes use of the distinction between the read and write values, probably because it is a text-based system and read and write are two major operations in an application. The values should be either access permitted or access denied, with an additional parameter specifying the application operation to which the value applies. SOL identifies some application operations and associates a value with them.

Several implications for design can be drawn from the discussion on concurrency control functionality. Concurrency control indicates users' ability to access an

application. An optimistic data consistency policy indicates multiple user access and can lead to data inconsistencies if other mechanisms are not in place, such as an undo mechanism. A pessimistic data consistency policy indicates only single user access at one time. With a pessimistic data consistency policy, if the scope of the policy is per application, then concurrent access by multiple users is not supported, and therefore only one user can access an application at one time. With a pessimistic data consistency policy, if the scope of the policy is per application object, then it is possible for one user to access one object and another user to access another object at the same time. Sometimes, controlling access to an application is decided by the application requirements, such as turn-taking in a card-game.

5.5. Conclusions

This chapter shows the Framework is a useful analysis tool for comparing the functionality of software toolkits which help to construct applications in group-work. The characterisations described in Chapter 4 provide a sound basis on which to assess a toolkit's functionality and thus indicate those aspects of group-working that are supported or not supported by a particular toolkit. By making a toolkit's functionality explicit, it can be compared.

The comparison indicates commonalities and differences between systems. For example, GroupKit and Dialogo aim to support WYSIWIS views, whereas Rendezvous, Suite and SEPIA give application developers a choice in the specific aspects of presentation that might be shared. The choice of aspects supported by systems can also be compared.

The comparison indicates the focus of a particular system because it is referenced to a complete description of the functionality. For example, it becomes apparent that SEPIA focuses on supporting users move between three shared states as easily as possible.

The Framework can be seen as a predictive tool where given specific group-work support requirements, appropriate toolkits can be identified. For example, if sessions are expected to be dynamic and users are joining and leaving at different times, it is necessary that a toolkit supports latecomers. Using the characterisations in chapter 4, it is apparent that Rendezvous and GroupKit support latecomers and therefore, either of these toolkits would be suitable and would meet this particular group-work support requirement.

In comparing the characterisations, it becomes apparent that certain functionality described in the Framework is not supported by any of the systems. There may be explanations for implementation reasons why some functionality is not supported. For example, functionality associated with the membership ticket entity is not fully supported. This is probably because of the trade-off between employing some existing technology, such as a window system, and building this technology from scratch. However, other functionality described in the Framework provides new areas for design, such as support for logical representation data sharing which would enable a toolkit to run on multiple machine platforms. Another area for design is support for function data sharing where users have different interactions with the same application functionality.

In comparing the characterisations, it also becomes apparent that certain functionality described in the Framework does not add to an analysis or needs further enhancements. For example, the data distribution mechanisms do not helpfully describe the support offered by systems, and colour is an aspect of physical representation data. This functionality is summarised in the section on further work in Chapter 6.

Chapter 6:

Conclusions

6.1. Introduction

This chapter presents a summary of the thesis. It is followed by a discussion of the research method adopted by the thesis and how the Framework could be used by others. A section on further work discusses: firstly, refinements to the Framework within its current scope, secondly, extending the scope of the Framework, and thirdly, suggestions for additional uses of the Framework. A final section presents the concluding remarks.

6.2. Summary

A framework has been developed for understanding and comparing software toolkits which support the development of group-working applications. The Framework offers a description of the functionality in group-work support. It is assumed that group-work support is domain and application independent, and can be considered separately to application functionality. The Framework identifies a set of properties in terms of entities, entity relationships, states and state transitions, by applying extended entity relationship and state transition techniques to collaborative work. For example, transitions in session management are described, such as joining a group, and four states of data sharing are identified.

By using the Framework as an analysis tool, it is possible to identify the properties supported by a particular system in a form described in the Framework. The Framework was used to characterise a set of systems which are considered to be representative of systems in CSCW. The set included the following systems: Rendezvous, GroupKit, Dialogo, Suite, SEPIA, MEAD and SOL. The properties supported by each system became apparent by comparing the system with the

properties described in the Framework. For example, it was possible to detail the specific session management transitions and shared states supported by each system. The set of properties described in the Framework correspond to a set of user behaviours, and as a result, it is possible to indicate those aspects of group-working that are supported or not supported by a particular toolkit. For example, if a user can be added to a running session, then the system is seen to support latecomers. Thus, the functionality of group-work support provided by each toolkit was made explicit and described in a common form.

By comparing the characterisations of each toolkit, commonalities and differences between the systems became apparent. For example, it was possible to see which systems supported a particular shared state. Thus, the Framework was shown to be a useful comparison tool. In addition, it became possible to assess a toolkit for its support for specific application requirements. For example, if a particular shared state is required in an application, it was possible to identify the toolkits that support the shared state. Further, aspects of functionality which are not supported by the representative set of systems become areas for potential development. For example, one area for development would be to enable users to have different interactions with the same application functionality.

6.3. Method of analysis

This section comprises two parts. The first part comments on the research method adopted by the work presented in this thesis. The second part discusses how the Framework might be used by others.

6.3.1. Research method

The research method employs an extended form of entity relationship modelling and state transition techniques to provide a complete and consistent description of group-work support in the Framework. Because the Framework proves to be a useful comparison tool, the properties identified in the Framework must be appropriate for characterising toolkits at this granularity, and therefore, the techniques used to derive these properties must also be appropriate.

When studying an area of work, the techniques employed in this thesis direct the researcher to examine the entities which are central to collaborative work. This might be a fairly straightforward activity. It is then necessary to examine changes that occur to these entities. This thesis focuses on generic group-work support and not on task-specific support. Therefore, the Framework describes entities and their associated

changes pertaining to group-work functionality rather than application functionality. Entities and changes to the entities that are specific to an application could also be described by employing these techniques.

The limitations of the analysis are largely based on ignoring some entities that may be important. There has been no detailed analysis of objects such as documents which may be part of the system, but the data on which a document is based is a subset of the persistent data. Many other entities have similar relationships with the higher level entities addressed by the Framework. The deficiencies of the approach are therefore based on the lack of fine detail which is on the one hand a weakness and on the other hand a strength in that the Framework can be applied to a large number of actual systems. The level chosen has enabled a comparison of several disparate systems which an approach based on a finer level of detail would have precluded.

On reflection, the set of software toolkits chosen to represent systems in CSCW proved fruitful. The toolkits captured most of the functionality described in the Framework. The aspects of functionality not supported by the systems lead to future areas of design rather than the conclusion that other toolkits would have supported this functionality or that the Framework does not provide an adequate analysis tool by characterising the aspects not supported and not characterising aspects that are supported.

The selection of toolkits also enabled the Framework to characterise the variety of functionality supported and enabled commonalities and differences between toolkits to be shown. Rendezvous, GroupKit, Dialogo, Suite and SEPIA are used as the primary systems and it was not considered necessary to analyse MEAD and SOL in the same detail, nor was it considered necessary to add other systems to the set.

6.3.2. Method of use

The method of analysis for characterising or comparing software toolkits that could be followed by others is not dissimilar from the research method adopted by this thesis. A difference is that the Framework already exists and so does not need to developed. In addition, exemplary characterisations of existing systems are available and can be used to guide the analysis.

The method involves using the Framework as an analysis tool to identify the properties supported by a particular system. The Framework divides the group-work support functionality into three broad categories: session management, data sharing

and concurrency control. Within each category, the entities in a system are compared to those described in the Framework. This is followed by examining the different entities, the relationships between the entities, the states of the entities and their relationships, and finally, identifying the state transitions. The transitions are compared to those described in the Framework to check on any that have been omitted. The entities, entity relationships, states and state transitions constitute a characterisation of a system in terms of its group-work support functionality. The characterisations of existing systems provided in Chapter 4 should help with any difficulties encountered at this stage.

Comparing characterisations is relatively straightforward. Again, aspects of the functionality are divided into the three broad categories and within each category, the entities, entity relationships, states and state transitions supported by each system are compared. Commonalities and differences between toolkits become apparent. Comparing these results and the results in Chapter 5 with the Framework, indicates aspects of functionality not supported by any of the systems and potential areas of design for future toolkit development.

6.4. Further work

This section is divided into three parts. The first part discusses refinements to the Framework within its current scope. It summarises the additions to the Framework suggested in Chapters 4 and 5, and the aspects of functionality that need a fuller description in the Framework, suggested in Chapter 5. The second part discusses extending the scope of the Framework. The third part suggests potential uses of the Framework in addition to its use as an analysis and comparison tool.

6.4.1. Refinements to the Framework

Chapters 4 and 5 highlighted aspects of the functionality supported by systems but not described in the Framework, and suggested appropriate refinements to the Framework. The refinements are listed below:

• Provide explicit support for roles, particularly with regard to access permissions (refer to section 4.2.4 for details).

• Consider a group supported by part of an application (refer to section 4.6.4 for details).

• Expand the membership ticket entity functionality to describe relationships between applications and restrictions on users running these applications (refer to section 4.6.4 for details).

• Incorporate colour, format and visibility as aspects of physical representation data (refer to section 5.3 for details).

• Change values of the access right parameter in the "create consistency ticket" transition from read, write or none, to access permitted or access denied. In addition, there needs to be the possibility of including an additional parameter in the transition for the application operation. This would result in a permission matrix similar to that described in SOL (refer to section 5.4 for details).

Chapter 5 discusses aspects of functionality described in the Framework that do not add to the analysis as they currently stand. The descriptions of these particular aspects in the Framework need to be re-worked. The aspects referred to include:

• The "create data object", "delete data object" and "realise data object" transitions (refer to section 3.3.3 for the description of these transitions).

• The data distribution mechanisms which extend the "create data object" transition with a location parameter and introduce the "copy data object" transition (refer to section 3.3.3 for the description of these transitions).

The Framework could have been refined and re-worked but the aim was to explore how such a framework could be exercised. The Framework proved to be substantial and to stand up to the exercise, only uncovering these minor defaults. With the additions listed above, the Framework would describe all the functionality supported by the systems that are analysed. When the aspects of functionality described above are re-worked, the Framework would provide a complete and consistent description of group-work support within its scope.

6.4.2. Extending the scope of the Framework

The Framework could be extended to describe the implementation of group-work support functionality, the performance of an implementation, the presentation of the group-work support functionality and coordination mechanisms. Each of these extensions is discussed below.

Implementation is raised as an issue when describing physical representation data sharing in particular, and specifically for Rendezvous and Dialogo. The Framework describes physical representation data as a composite data object that can be further unwrapped into different types of specification where the specifications are implementation-dependent. Since window systems have become common place and window system software is of a structure that has become normalised, i.e. the client/ server architecture, describing a window system specification for physical representation data is likely to contribute to an analysis and guide an implementation. However, it is important to remember that the specification is implementation-dependent in order to be open to innovative alternatives.

Because the description of the Framework is not implementation-dependent, the scope for assessing the performance of a system is limited. Performance benefits include improved response time, which is crucial in the group context, and lower network traffic. An indication of the performance is given by the structure of the support, and in particular, whether entities are replicated or not. If entities are replicated, then there is a possibility that the replicated copies run as multiple processes, possibly on different processors. However, in Rendezvous, view objects are replicated but still run as one process. An implementation-dependent description would allow the performance to be assessed and would involve closer examination of the processes, their distribution and the inter-process communication.

The Framework focuses on describing the group-work support functionality and not on how the functionality might be presented to users, as explained in section 3.5. Any system has to explore how the functionality might be presented. Some systems support particular aspects of the presentation. For example, Suite and SEPIA explore supporting users moving between different shared states as easily as possible. As a result, an understanding of the user requirements is acquired and possible solutions presented from the systems that have been built. User requirements are known to be domain, task, group and user specific. Therefore, it would be inappropriate to try and specify the presentation of the functionality out of the context of its actual use. The Framework makes the entities and transitions that need supporting explicit, enabling designers to exploit the description and come up with novel presentations in particular contexts (Jones & Edmonds, 1995). A description of user requirements, as they are currently understood, would also help a designer.

The Framework does not describe additional coordination mechanisms which might be employed to facilitate the group process. Many of the theories and techniques described in the review of systems in Chapter 2 analyse and represent such

coordination mechanisms. It is necessary to introduce a time element into the Framework to support a richer structure, and initially, to examine the notions of simultaneous activity, sequenced activity and shared resources. It is also necessary to reflect on how the various theories and techniques are related to the Framework. This is an area of future work which the author hopes to pursue.

6.4.3. Potential uses of the Framework

The set of properties described in the Framework could be said to define the complete group-work support functionality of a software toolkit in CSCW. This assertion is true if three distinct parts are proven. Firstly, it requires that the set of properties are complete by derivation. This is considered to be the case because of the use of extended entity relationship modelling and state transition techniques to derive the properties. Secondly, it requires that the set of toolkits used to test the Framework are representative of all toolkits in CSCW and therefore incorporate any functionality that a toolkit might support. This is considered to be the case because of the toolkits that are chosen and the breadth of functionality encompassed by the toolkits in the set. Further, there is no evidence to suggest this not to be the case. Thirdly, it requires that the set of toolkits' functionality can be described by the properties. This is not considered to be the case because of the additions and enhancements detailed above. However, by following the suggestions and modifying the Framework, the properties could be considered to describe all the toolkits' functionality in the chosen granularity and therefore define the complete group-work support functionality within the defined scope.

The properties can be shown to detail an executable specification that can be implemented and dynamically changed. GroupKit illustrates how "open protocols" are used to support applications which are tailored for particular groups and for individual users. The protocols employed by GroupKit are not dissimilar to the transitions described in the Framework. Further, some of the properties described in the Framework have been used in the development of the MUMS toolkit (Jones et al., 1997). For example, the "add user" transition has been implemented using a declarative programming language. It can be invoked during a running session, allowing the specification to be dynamically changed.

Having examined the functionality of group-work support, the next stage is to consider the structure of the support and to develop a reference model (Abowd, 1994). A system architecture defines how a system is statically divided into a collection of interacting computational elements. Software toolkits always embody

some sort of structure but some make it central to their development. Rendezvous, Suite and SOL support the notion of separating user interaction and application functionality. Rendezvous supports a constraint-based User Interface Management System (UIMS) and Patterson (1994), a developer of Rendezvous, outlines a taxonomy of architectures, described in section 2.2.2. MUMS employs a UIMS and agent-based technology to structure aspects of the group-work support described in the Framework (Edmonds et al., 1994, Jones et al., 1997). For example, a user agent supports aspects of data sharing, a floor agent supports user input control, a conference agent supports aspects of session management and an application agent records the state of an application. The Framework can be used as a basis on which to develop a reference model.

6.5. Concluding remarks

The thesis has shown how the Framework can be used to gain an understanding of real-time CSCW toolkits. It has analysed existing toolkits and demonstrated interesting insights into toolkit functionality. It has compared toolkits and demonstrated their commonalities and differences, and the focus of their support.

The Framework brings benefits for a variety of potential users. Four potential users of the Framework have been identified: application developers, toolkit designers, requirement specialists and groupware evaluators. A description of how they might use the Framework is detailed below.

The Framework can be used by application developers to assist in the selection of an appropriate toolkit to be employed to meet specific application requirements. For example, if supporting latecomers is an application requirement because the application will be used in an environment where the users of the application cannot be pre-determined, then GroupKit would be an appropriate toolkit. Some of the requirements might be supported by one toolkit and other requirements supported by another toolkit, so priorities will need to be addressed.

The Framework can be used by future designers of toolkits as a specification of the functionality. For example, in the "open protocols" approach adopted in GroupKit, one of the problems highlighted is that the designer is required to specify the state changes and protocol, and thus predict the system's future use. The Framework provides a sound basis on which to develop such an approach.

Because the Framework indicates the implications for users of specific system functionality, it can be used as a tool to help assess the group requirements for an application. For example, the Framework specifies aspects of data sharing which can be considered as requirement options. The aspects of data sharing that need to be shared in a particular application context must be decided. It might be considered necessary that users maintain the same window size but can scroll the workspace individually to maintain a shared workspace and provide sufficient flexibility.

The Framework can be used to help evaluate a groupware application by comparing the functionality supported by the application and that described in the Framework. Similar to the analyses of toolkits carried out in Chapter 4, a comparison would result in an understanding of the functionality the application does support and the functionality the application does not support. The implications for user support become apparent.

References

Abowd, G.D. (1994). Defining reference models and software architectural styles for cooperative systems. Position paper in CSCW'94 Workshop on Software Architectures for Cooperative Systems, held October 21, Chapel Hill, North Carolina.

Ahuja, S.R., Ensor, J.R. & Lucco, S.E. (1990). A comparison of application sharing mechanisms in real-time desktop conferencing systems. In Proceedings of Office Information Systems 1990, pp. 238-248. ACM.

Altenhofen, M. (1990). Upgrading a window system for tutoring functions. In Proceedings of the European X Window System Conference, London, November 1990, pp. 37-44. CEP Consultants Ltd.

Baecker, R.M., Nastos, D., Posner, I.R. & Mawby, K.L. (1993). The user-centred iterative design of collaborative writing software. In Proc. InterCHI'93, pp. 399-405. ACM.

Bannon, L. & Schmidt, K. (1991). CSCW: Four characters in search of a context. In Bowers, J.M. & Benford, S.D. (eds), Studies in Computer Supported Cooperative Work: Theory, Practice and Design, pp. 3-16. Elsevier Science, Amsterdam.

Bass, L. (1994). The role of a body worn CSCW system in the generation of a CSCW reference architecture. Position paper in CSCW'94 Workshop on Software Architectures for Cooperative Systems, held October 21, Chapel Hill, North Carolina.

Bell, D. & Johnson, P. (1993). A contingency model for groupware design and tailoring. Presented at The Schaerding Workshop on Design of Computer Supported Cooperative Work and Groupware Systems, Austria, June 1993. To be published by Elsevier.

Bentley, R., Rodden, T., Sawyer, P. & Sommerville, I. (1992). An architecture for tailoring cooperative multi-user displays. In Proc. CSCW'92, pp. 187-194. ACM.

Bentley, R., Rodden, T., Sawyer, P. & Sommerville, I. (1994). Architectural support for cooperative multiuser interfaces. Computer, May 1994, pp. 37-46. IEEE.

Berlage, T. & Genau, A. (1993). A framework for shared applications with a replicated architecture. In UIST'93, Proceedings of the ACM Symposium on User Interface Software and Technology (Nov. 3-5, Atlanta, Georgia), ACM/ SIGCHI, NY, pp. 249-257.

Bignoli, C. & Simone, C. (1991). AI techniques for supporting human to human communication in CHAOS. In Bowers, J.M. & Benford, S.D. (eds), Studies in Computer Supported Cooperative Work: Theory, Practice and Design, pp. 103-118. Elsevier Science, Amsterdam.

Bly, S.A., Harrison, S.R. & Irwin, S. (1993). Media spaces: Bringing people together in a video, audio, computing environment. Communications of the ACM, 36, 1 (January), pp. 28-47. ACM.

Bobrow, D.G., Stefik, M., Foster, G., Halasz, F., Lanning, S. & Tatar, D. (1990). The Colab project final report SSL-90-45. Xerox Palo Alto Research Centre.

Booch, G. (1986). Object-Oriented Development, IEEE Trans. on Software Eng., 12, 2 (Feb.), pp. 211-221.

Bracchi, G. & Pernici, B. (1984). The design requirements of office systems. ACM Transactions on Office Information Systems, 2, 2, pp. 151-170. ACM.

Brinck, T. & Gomez, L.M. (1992). A collaborative medium for the support of conversational props. In Proc. CSCW'92, pp. 171-178. ACM.

Brinck, T. & Hill, R. (1993). Building shared graphical editors using the abstraction-link-view architecture. In Proc. ECSCW'93, pp. 311-324. Kluwer Academic, Dordrecht.

Chen, P.P. (1976). The Entity-Relationship Model - Toward a Unified View of Data, ACM Trans. on Database Systems, 1, 1 (March), pp. 9-36. ACM.

Conklin, J. & Begeman, M.L. (1988). gIBIS: A hypertext tooling for exploratory policy discussion. In Tatar, D. (ed), Proceedings of the 2nd Conference on Computer-supported Cooperative Work, pp. 140-152. ACM, New York.

Cook, P., Ellis, C., Graf, M. et al. (1987). Project NICK: meetings augmentation and analysis. ACM TOIS 5, 2, pp. 132-146. ACM.

Coutaz, J. (1989). UIMS: promises failures and trends. In Sutcliffe, A. & Macaulay, L. (eds), People and Computers V, Proceedings of HCI'89 (Nottingham), pp. 71-84. Cambridge University Press, Cambridge.

Cox, B.J. (1986). Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H. & Tomlinson, R. (1990). MMConf: An infrastructure for building multimedia applications. In Proc. CSCW'90, pp. 329-342. ACM.

Dance, J.R., Granor, T.E., Hill, R.D., Hudson, S.E, Meads, J., Myers, B.A. & Schubert, A. (1987). The run-time structure of UIMS-supported applications. Computer Graphics 21 (2), pp. 97-101.

Danielson, T., Pankoke-Babatz, U., Prinz, W., Patel, A., Pays, P., Smaaland, K. & Speth, R. (1986). The AMIGO project: advanced group communication model for computer-based communication environment. In Proc. CSCW'86, pp. 229-246. MCC, Austin, Texas.

De Cindio, F., De Michelis, G., Simone, C., Vassallo, R., & Zanaboni, A. (1986). CHAOS as a coordinaton technology. In Proc. CSCW'86, pp. 325-342. MCC, Austin, Texas.

De Michelis, G. & Grasso, M.A. (1994). Situated conversations within the language/ action perspective: The Milan Conversation Model. In Proc. CSCW'94, pp. 89-100. ACM.

DePaoli, F. & Tisato, F. (1991). A model for real-time co-operation. In Proc. ECSCW'91, pp. 203-217. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Desanctis, G. & Gallupe, B. (1987). A foundation for the study of GDSSs. Management Science, 35, 5, pp. 589-609.

Dewan, P. (1990). A tour of the Suite user interface software. In Proc. UIST'90, pp. 57-65. ACM.

Dewan, P. & Choudhary, R. (1991a). Flexible user interface coupling in a collaborative system. In Proc. CHI'91, pp. 41-48. ACM.

Dewan, P. & Choudhary, R. (1991b). Primitives for programming multi-user interfaces. In Proc. UIST'91, pp. 69-78. ACM.

Dewan, P. (1992). Principles of designing multi-user user interface development environments. In Larson, J. & Ungar, C. (eds) Engineering for Human-Computer Interaction, pp. 35-50. Elsevier Science Publishers B.V., North-Holland.

Dewan, P. & Choudhary, R. (1992). A high-level and flexible framework for implementing multiuser user interfaces. ACM TOIS 1, 4 (October 1992), pp. 345-380. ACM.

Dewan, P. & Choudhary, R. (1995). Coupling the user interfaces of a multiuser program. ACM Transactions on Computer Human Interaction, 2, 1, pp. 1-39. ACM.

Dollimore, J. & Wilbur, S. (1991). Experiences in building a configurable CSCW system. In Bowers, J.M. & Benford, S.D. (eds), Studies in Computer Supported Cooperative Work: Theory, Practice and Design, pp. 173-181. Elsevier Science, Amsterdam.

Dourish, P. (1995a). The parting of the ways: divergence, data management and collaborative work. In Proc. ECSCW'95, pp. 215-230. Kluwer Academic, Dordrecht.

Dourish, P. (1995b). Developing a reflective model of collaborative systems. Transactions on Computer Human Interaction, 2, 1, pp. 40-63 (March). ACM.

Dourish, P. (1996). Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In Proc. CSCW'96, pp. 268-277. ACM.

Edmonds, E.A. (1993). The Separable User Interface. Academic Press, London.

Edmonds, E.A. & McDaid, E. (1990). An architecture for knowledge-based front-ends. Knowledge-Based Systems, 3 (4), pp. 221-224.

Edmonds, E.A., Candy, L., Jones, R.M. and Soufi, B. (1994). Support for collaborative design: agents and emergence. Communications of the ACM, 37 (7), pp. 41-47. ACM, New York.

Edwards, W.K. (1994). Session management for collaborative applications. In Proc. CSCW'94, pp. 323-330. ACM.

Edwards, W.K. (1996). Policies and roles in collaborative applications. In Proc. CSCW'96, pp. 11-20. ACM.

Ellis, C.A. & Gibbs, S.J. (1989). Concurrency control in groupware systems. In Proc. of the ACM SIGMOD International Conference on the Management of Data, pp. 399-407, Seattle, Washington, USA. ACM.

Ellis, C.A., Gibbs, S.J. & Rein, G.L. (1991). Groupware: some issues and experiences. Communications of the ACM 34, 1 (January 1991), pp. 38-58. ACM.

Ellis, C.A. & Wainer, J. (1994). A conceptual model of groupware. In Proc. CSCW'94, pp. 79-88. ACM.

Elwart-Keys, M., Halonen, D., Horton, M., Kass, R. & Scott, P. (1990). User interface requirements for face to face groupware. In Proc. CHI'90, pp. 295-301. ACM.

Ensor, J.R., Ahuja, S.R., Horn, D.N. & Lucco, S.E. (1988). The Rapport multimedia conferencing system - a software overview. In Proceedings IEEE Conference on Computer Workstations, Santa Clara, March 1988, pp. 52-58. IEEE.

Fish, R.S., Kraut, R.E., Leland, M.P. & Cohen, M. (1988). Quilt- a collaborative tool for cooperative writing. In Proc. ACM COIS'88, pp. 30-37. ACM.

Fish, R.S., Kraut, R.E., Root, R.W. and Rice, R.E. (1992). Evaluating video as a technology for informal communication. In Proc. CHI'92, pp. 37-48. ACM.

Fiske, J. (1990). Introduction to Communication Studies. Routledge, New York.

Fitzpatrick, G., Tolone, W.J. & Kaplan, S.M. (1995). Work, locales and distributed social worlds. In Proc. ECSCW'95, pp. 1-16. ACM.

Flores, F., Graves, M., Hartfield, B. & Winograd, T. (1988). Computer systems and the design of organizational interaction. ACM TOIS, 6, 2, (April 1988), pp. 153-172. ACM.

Freeman, S.M.G. (1994). An architecture for distributed user interfaces. Technical report 342, Computer Lab, University of Cambridge, July 19994.

Furuta, R. & Stotts, P.D. (1994). Interpreted collaboration protocols and their use in groupware prototyping. In Proc. CSCW'94, pp. 121-131. ACM.

Gaver, W. (1991). Technology affordances. In Proc. CHI'91, pp. 79-84. ACM.

Gaver, W., Moran, T., MacLean, A., Lovstrand, L., Dourish, P., Carter, K. and Buxton, W. (1992). Realizing a video environment: EuroPARC's RAVE system. In Proc. CHI'92, pp. 27-35. ACM.

Gerson, E.M. & Star, S.L. (1986). Analysing due process in the workplace, ACM Transactions on Office Information Systems, 4, 3 (July), pp. 257-270. ACM.

Gibbs, S.J. (1989). LIZA: an extensible groupware toolkit. In Proc. CHI'89, pp. 29-35. ACM.

Greenberg, S. (1990). Sharing views and interactions with single-user applications. In Proc. OIS, pp. 227-237. ACM.

Greenberg, S. (1991). Personalisable groupware: accommodating individual roles and group differences. In Proc. CSCW'91, pp. 17-31. Kluwer Academic Publishers, Dordrecht.

Greenberg, S. & Marwood, D. (1994). Real time groupware as a distributed system: concurrency control and its effect on the interface. In Proc. CSCW'94, pp. 207-217. ACM.

Greenberg, S., Roseman, M., Webster, D. & Bohnet, R. (1992). Human and technical factors of distributed group drawing tools. Interacting with Computers, 4, 3, pp. 364-392. Butterworth-Heinemann.

Greif, I. & Sarin, S. (1988). Data sharing in group work. In Greif, I. (ed), Computer-Supported Cooperative Work: A Book of Readings, pp. 477-508. Morgan Kaufmann, San Mateo. ISBN 0-934613-57-5.

Grudin, J. (1989). Why groupware applications fail: problems in design and evaluation. Office: Technology and People 4, 3, pp. 245-264. Elsevier Science Publishers, England.

Gust, P. (1988). SharedX: X in a distributed group work environment. In Proceedings of 2nd Annual X Technical Conference, MIT, Boston.

Gutwin, C., Roseman, M. & Greenberg, S. (1996). A usability study of awareness analysis widgets in a shared workspace groupware system. In Proc. CSCW'96, pp. 258-267. ACM.

Haake, A. & Haake, J. (1993). Take CoVer: exploiting version management in collaborative systems. In Proc. InterCHI'93. ACM.

Haake, J. & Wilson, B. (1992). Supporting collaborative writing of hyperdocuments in SEPIA. In Proc. CSCW'92, pp. 138-146. ACM.

Halonen, D., Horton, M., Kass, R. & Scott, K. (1990). Shared hardware: a novel technology for computer support of face to face meetings. In Proc. OIS'90, pp. 163-168. ACM.

Henderson, D.A. & Card, S.K. (1986). Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. ACM Transactions on Graphics, 5, 3 (July 1986), pp. 211-243. ACM.

Hennessey, P. (1990). Modelling group communication. PhD Thesis, Nottingham University.

Hill, R.D. (1992). The abstraction-link-view paradigm, using constraints to connect user interfaces to applications. In Proc. CHI'92, pp. 335-342. ACM.

Hill, R.D., Brink, T., Patterson, J.F., Rohall, S.L. & Wilner, W.T. (1993). The Rendezvous language architecture. Communications of the ACM 36, 1 (January 1993), pp. 62-67. ACM.

Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F. & Wilner, W. (1994). The Rendezvous architecture and language for constructing multi-user applications. In ACM Transactions on Computer-Human Interaction 1, 2 (June 1994), pp. 81-125. ACM.

Holt, A.W. (1988). Diplans: A new language for the study and implementation of coordination. ACM TOIS 6, 2, (April 1988), pp. 109-125.

Hopgood, F.R.A., Duce, D.A., Gallop, J.R. & Sutcliffe, D.C. (1983). Introduction to the Graphical Kernel System (GKS). Academic Press, London.

Hughes, J., King, V., Rodden, T. & Anderson, H. (1994). Moving out from the control room: ethnography in system design. In Proc. CSCW'94, pp. 429-439. ACM.

Hutchins, E. (1990). The technology of team navigation. In Intellectual Teamwork, (eds) Kraut, R.E. & Egido, C., pp. 191-220. Lawrence Erlbaum Associates, Hillsdale, New Jersey. ISBN 0-8058-0533-8.

Jeffay, K., Lin, J.K., Menges, J., Smith, F.D. & Smith, J.B. (1992). Architecture of the artifact-based collaboration system Matrix. In Proc. CSCW'92, pp. 195-202. ACM.

Jefferson, D.R. (1985). Virtual time. ACM Transactions on Programming Languages and Systems 7, 3 (July), pp. 404-425. ACM.

Jones, R.M. & Edmonds, E.A. (1994). A framework for negotiation. In Connolly, J.H. & Edmonds, E.A. (eds), CSCW and Artificial Intelligence, pp. 13-22. Springer-Verlag, London.

Jones, R.M. & Edmonds, E.A. (1995). Supporting collaborative design in a seamless environment. Concurrent Engineering: Research and Applications, 3 (3), pp. 203-211. Academic Press, London.

Jones, R.M., Copas, C. & Edmonds, E.A. (1997). GIS support for distributed group-work in regional planning. International Journal of Geographical Information Systems, (January). Taylor & Francis.

Kaplan, S.M. (1990). ConversationBuilder: an open architecture for collaborative work. In Diaper, D., Cockton, G., Gilmore, D. & Shackel, B. (eds), Proceedings of the IFIP TC 13 Third International Conference on Human-Computer Interaction, Cambridge, (INTERACT'90), pp. 917-922. Elsevier Science Publishers b.v. (North-Holland).

Kaplan, S.M., Tolone, W.J., Bogia, D.P. & Bignoli, C. (1992). Flexible, active support for collaborative work with ConversationBuilder. In Proc. CSCW'92, pp. 378-385. ACM.

Karsenty, A. & Beaudouin-Lafon, M. (1993). An algorithm for distributed groupware applications. In Proc. of the 13th International Conference on Distributed Computer Systems ICDCS'93, Pittsburg, May 25-28.

Kensing, F. & Winograd, T. (1991). The language/ action approach to design of computer-support for cooperative work: A preliminary study. In Stamper, R.K., Kerola, P. and Lyytinen, K. (eds), Collaborative Work, Social Communication and Information System, pp. 311-331. North Holland, Amsterdam.

Kreifelts, T., Hinrichs, E., Klein K.-H., Seuffert, P. & Woetzel, G. (1991). Experiences with the Domino office procedure system. In Proc. ECSCW'91, pp. 117-130. Kluwer Academic, Dordrecht.

Kuutti, K. (1991). The concept of activity as a basic unit of analysis for CSCW research. In Proc. ECSCW'91, pp. 249-264. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Lai, K.-Y., Malone, T.W. & Yu, K.-C. (1988). Object Lens: a "spreadsheet" for cooperative work. ACM TOIS 6, 4 (October 1988), pp. 332-353. ACM.

Lantz, K.A. (1987). Multi-process structuring of user interface software. Computer Graphics 21, 2, pp. 124-130.

Lantz, K.A. (1988). An experiment in integrated multimedia conferencing. In Greif, I. (ed), Computer-Supported Cooperative Work: A Book of Readings, pp. 533-552. Morgan Kaufmann, San Mateo. ISBN 0-934613-57-5.

Lauwers, J.C. (1990). Collaboration transparency in desktop teleconferencing environments. Technical Report: CSL-TR-90-435. Stanford University, US.

Lauwers, J.C. & Lantz, K.A. (1990). Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In Proc. CHI'90, pp. 303-311. ACM.

Lauwers, J.C., Joseph, T.A., Lantz, K.A. & Romanow, A.L. (1990). Replicated architectures for shared window systems: a critique. In Proc. OIS'90, pp. 249-260. ACM.

Malone, T.W., Grant, K.R., Lai, K.-Y., Rao, R. & Rosenblitt, D. (1987). Semistructured messages are surprisingly useful for computer-supported coordination. ACM TOIS 5, 2 (April 1987), pp. 115-131. ACM.

Malone, T.W. & Crowston, K. (1990). What is coordination theory and how can it help design cooperative work systems? In Proc. CSCW'90, pp. 357-370. ACM.

Malone, T.W., Lai, K.-Y. & Fry, C. (1992). Experiments with Oval: a radically tailorable tool for cooperative work. In Proc. CSCW'92, pp. 289-297. ACM.

Malone, T.W., Lai, K.-Y. & Fry, C. (1995). Experiments with Oval: a radically tailorable tool for cooperative work. ACM Transactions on Information Systems, 13, 2 (April 1995), pp. 177-205. ACM.

Mantei, M. (1988). Capturing the Capture Lab concepts: a case study in the design of computer supported meeting environments. In Proc. CSCW'88, pp. 257-270. ACM.

Mantei, M. (1989). Observation of executives using a computer supported meeting environment. Decision Support Systems 5, pp. 153-166. Elsevier Science Publishers, North-Holland.

Mantei, M., Baecker, R.M., Sellen, A., Buxton, W., Milligan, T. & Wellmen, B. (1991). Experiences in the use of a media space. In Proc. CHI'91, pp. 203-208. ACM.

Munson, J.P. & Dewan, P. (1994). A flexible object merging framework. In Proc. CSCW'94, pp. 231-252. ACM.

Neuwirth, C.M., Kaufer, D.S., Chandhok, R. & Morris, J.H. (1990). Issues in the design of computer support for co-authoring and commenting. In Proc. CSCW'90, pp. 183-195. ACM.

Neuwirth, C.M., Chandhok, R., Kaufer, D.S., Erion, P., Morris, J. & Miller, D. (1992). Flexible diff-ing in a collaborative writing system. In Proc. CSCW'92, pp. 147-154.

Newman-Wolfe, R.E. & Pelimuhandiram, H.K. (1991). MACE: a fine grained concurrent editor. In Proceedings of the ACM COOCS Conference on Organizational Computing Systems, pp. 240-254. ACM.

Nunamaker, J.F., Dennis, A.R., Valacich, J.S., Vogel, D.R. & George, J.F. (1991). Electronic meeting systems to support group work. Communications of the ACM, 34, 7, pp. 43-61. ACM.

Patterson, J.F. (1991). Comparing the programming demands of single-user and multi-user applications. In Proc. UIST'91 (Proceedings of the 4th ACM SIGRAPH Conference on User Interface Software and Technology, November 1991), pp. 87-94. ACM.

Patterson, J.F., Hill, R.D., Rohall, S.L. & Meeks, W.S. (1990). Rendezvous: an architecture for synchronous multi-user applications. In Proc. CSCW'90, pp. 317-328. ACM.

Patterson, J. (1994). A taxonomy of architectures for synchronous groupware applications. Position paper in CSCW'94 Workshop on Software Architectures for Cooperative Systems, held October 21, Chapel Hill, North Carolina.

Pfaff, G. (ed) (1985). User Interface Management Systems, Springer Verlag, Berlin.

Prakash, A. & Knister, M.J. (1992). Undoing actions in collaborative work. In Proc. CSCW'92, pp. 273-280. ACM.

Prinz, W. (1989). Survey of group communication models and systems. In Pankoke-Babatz (ed), Computer Based Group Communication: the AMIGO Activity Model. Ellis Horwood, 1989.

Rao, V.S. & Jarvenpaa, S.L. (1991). Computer support for groups - Theory-based models for GDSS research. Management Science, 37, 10, pp. 1347-1362.

Rein, G.L. & Ellis, C.A. (1991). rIBIS: a real-time group hypertext system. IJMMS 34, pp. 349-367.

Rodden, T. (1991). A survey of CSCW systems. Interacting with Computers, 3, 3, pp. 319-353. Butterworth-Heinemann.

Rodden, T. & Blair, G. (1991). CSCW and distributed systems: The Problem of Control. In Proc. ECSCW'91, pp. 49-64. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Roseman, M. & Greenberg, S. (1992). GROUPKIT: A groupware toolkit for building real-time conferencing applications. In Proc. CSCW'92, pp. 43-50. ACM.

Roseman, M. & Greenberg, S. (1993). Building flexible groupware through open protocols. In COOCS'93, pp. 279-288. ACM.

Roseman, M. & Greenberg, S. (1996a). Building real time groupware with GroupKit, a groupware toolkit. Transactions on Computer Human Interaction, 3, 1, pp. 66-106. ACM.

Roseman, M. & Greenberg, S. (1996b). TeamRooms: Network places for collaboration. In Proc. CSCW'96, pp. 325-333. ACM.

Sarin, S. & Greif, I. (1988). Computer-based real-time conferencing systems. In Greif, I. (ed), Computer-Supported Cooperative Work: A Book of Readings, pp. 397-420. Morgan Kaufmann Publishers, San Mateo, California.

Searle, J.R. (1969). Speech Acts. Cambridge University Press, Cambridge.

Shen, H. & Dewan, P. (1992). Access control for collaborative environments. In Proc. CSCW'92, pp. 51-58. ACM.

Simone, C., Divitini, M. & Schmidt, K. (1995). A notation for malleable and interoperable coordination mechanisms for CSCW systems. In Proceedings of the Conference on Organisational Computing Systems, pp. 44-54. ACM.

Smith, H., Onions, J. & Benford, S. (1989). Distributed Group Communication - The AMIGO Information Model. Ellis Horwood.

Smith, G. & Rodden, T. (1995). SOL: A shared object toolkit for cooperative interfaces. International Journal of Human-Computer Studies, 4, pp. 207-234.

Stefik, M., Foster, G., Bobrow, D., Kahn, D., Lanning, S. & Suchman, L. (1987a). Beyond the chalkboard: computer support for collaboration and problem solving in meetings. Communications of the ACM 30, 1 (January 1987), pp. 32-47. ACM.

Stefik, M.J., Bobrow, D.G., Foster, G., Lanning, S.M. & Tatar, D.G. (1987b). WYSIWIS revised: early experiences with multiuser interfaces. ACM TOIS 5, 2 (April 1987), pp. 147-167. ACM.

Strauss, A. (1993). Continual Permutations of Action. Aldine de Gruyter, New York.

Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schnett, H. & Thuering, M. (1992). SEPIA: A cooperative hypermedia authoring environment. In Proc. of the 4th ACM European Conference on Hypertext (ECHT'92), Milan, Italy (December, 1992), pp. 11-22.

Streitz, N. (1994). Putting objects to work: Hypermedia as the subject matter and the medium for computer-supported cooperative work. Invited talk at ECOOP'94. In Tokoro, M. & Pareschi, R. (eds), Object-Oriented Programming. Lecture Notes in Computer Science 821, pp. 183-193. Springer, Berlin.

Streitz, N. (1995). Designing hypermedia: a collaborative activity. Communications of the ACM, 38, 8 (August 1995), pp 70-71. ACM.

Suchman, L. (1987). Plans and Situated Actions. Cambridge University Press.

Tang, J.C. (1989). Listing, drawing, and gesturing in design: a study of the use of shared workspaces by design teams. PhD thesis Department of Mechanical Engineering, Stanford University. (Also available as research report SSL-89-3, Xerox Palo Alto Research Centre.)

Tang, J.C. & Leifer, L.J. (1988). A framework for understanding the workspace activity of design teams. In Proc. CSCW'88, pp. 244-249.

Tatar, D.G., Foster, G. & Bobrow, D.G. (1991). Design for conversation: lessons from Cognoter. IJMMS, 34, pp. 185-209. Academic Press.

Tichy, F.W. (1982). RCS: a revision control system. In Proceedings of the ECICS'82 European Conference.

Tolone, W.J., Kaplan, S.M. & Fitzpatrick, G. (1995). Specifying dynamic support for collaborative work within WORLDS. In Proceedings of the Conference on Organisational Computing Systems, pp. 55-65. ACM.

Twidale, M., Randall, D. & Bentley, R. (1994). Situated evaluation for cooperative systems. In Proc. CSCW'94, pp. 441-452. ACM.

Valacich, J.S., Dennis, A.R. & Nunamaker, J.F. (1991). Electronic meeting support: the GroupSystems concept. IJMMS 34, pp. 261-282. Academic Press.

Wilbur, S.B. & Young, R.E. (1988). The COSMOS project: a multi-disciplinary approach to design of computer-supported group working. In R. Speth (ed), EUTECO 88: Research into Networks and Distributed Applications, Vienna, Austria, April 20-22. pp. 147-156.

Winograd, T. (1987). A language/ action perspective on the design of cooperative work. Human Computer Interaction 3, 1, pp. 3-30.

Winograd, T. and Flores, F. (1986). Understanding Computers and Cognition: A New Foundation for Design. Ablex Publishing Corp, Norwood, NJ.