NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

DISPLAY MANAGEMENT FOR A SHARED VISUAL WORKSPACE IN A GAME PLAYING CONTEXT

Eric Le Bail

Department of Electrical Engineering
University of Ottawa [1]
Ottawa, Ontario

[1] A thesis submitted to the School of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Applied Science in Electrical Engineering.

ISBN 0-315-75015-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgment

I wish to express my gratitude and appreciation to my thesis supervisor, Dr Morris Goldberg, whose guidance and support were constant during my research program.

I would also like to acknowledge the financial support from TRIO (Telecommunications Institute of Ontario).

Abstract

In this thesis, the term display management refers to organizing the way users have access to the graphical interface to a system. The special class of systems considered is that of real-time multi-user applications, that present special characteristics, different from those existing for single-user programs. In this context, display management is achieved through the use of a production system controlling the actions of the users, and determining how the graphical objects on each screen are to be seen. A prototype is implemented in the particular case of poker game. The approach chosen can be characterized by three salient features. Firstly, it integrates a rule-based system in a real-time multi-user application. Secondly, it emphasizes the reconfigurability aspect. Thirdly, the system allows to define complex relationships between the users and the interface.

Contents

List of Figures

Chapter 1

Introduction

If display management deals with defining and organizing the information visible on a computer screen, shared visual workspaces refer to computer applications involving several users at the same time. These applications, which we call multi-user, although more commonly used terminology includes groupware [Taz88] and CSCW (Computer Shared Collaborative Work) [SBK+87], require special interfaces emphasizing the collaborative aspect of these systems.

To make things clearer, we will carry along a practical example: a poker game, which provides a challenging example of collaborative application in terms of the richness of possible interactions. Poker game involves different representations of the same object, e.g. cards, and requires different screen areas with different permissions. The number of graphical objects to be handled is rather large, as it includes a deck of cards plus a number of chips. Moreover, the degree of complexity in terms of permitted or non-permitted actions, or, stated differently, the complexity of the relationships between the different actors of the game is fairly high.

It should be clear that we do absolutely not intend to code the rules of poker itself. We assume that players know them. Our general guideline is to be as close to real-world interaction as possible. In that respect, we do not want to introduce arbitrary constraints on participants. In real-world poker, people can make mistakes when dealing cards, so we will not prevent them from dealing six cards instead of five, for instance. However, there are a number of facilities we must provide so that the game is carried out in a reasonable fashion. For example the players must have the ability to hide their cards from their fellow players.

As far as display management is concerned, it is possible to distinguish a number of levels. At a first level, the graphical objects involved in a particular application have to be defined. At a second level, it is necessary to control what actions are allowed or not allowed on these objects, and how users see these objects. Compared to single-user interfaces, particular constraints arise from the fact that the same graphical object can be seen or accessed by several users simultaneously. In addition to the questions of floor control and of assigning user- and object-specific access rights to graphical objects, other specific issues include maintaining group cohesiveness, which can be done through mechanisms preserving geometric integrity of screens and enhancing mutual awareness between users.

In the approach we choose, interactions between users and the application are expressed through a set of atomic actions, which are elementary mouse actions performed on the graphical objects composing the interface. It becomes then possible to define rules specifying, for each user, each atomic action, each type of object, and each workspace, whether the action is permitted or not. Another parameter of each rule indicates how the corresponding object should be viewed. The main

motivation for using a rule-based system is reconfigurability. Multi-user systems are generally more complex and take more time to develop than single-user ones (the case of the "pseudo" multi-user systems will be studied later). With rules, modifying the behaviour of the system or defining a new application is easy. A last aspect of display management considered is to dynamically change the knowledge base associated with a given application. This will be solved using special rules. It is to be noted that the following two aspects of display management, geometrical constraints and window management, will not be directly addressed.

Original contributions of this work include the use of production rules in a real-time multi-user system. The use of artificial intelligence techniques in real-time applications sets particular constraints on the design of the system, as will be explained in Chapter 5 and 6. Another distinctive feature is the complexity of the relationships that can be defined between the users and the interface, in terms of permitted or invalid actions and display modes. The notion of public and private workspace is extended to that of possible actions/display modes combinations. Finally, more generally speaking, the system implemented is one of a few examples of multi-user applications built with a concern for reconfigurability in mind.

To achieve display management, two main techniques will be used: object-oriented design and rule-based systems. For reader's convenience, we will dedicate a short chapter on object-oriented techniques and the X Window system, as they have been used extensively throughout this work. In developing this system, the expectations are manyfold and include deriving system feasibility in terms of response times, user feedback and obtaining more insight into the components required in a multi-user interface. Indeed, even though many systems exist, none of them has reached a large community of users: multi-user systems are not ready for public use yet [Che89].

The outline of the thesis is as follows: Chapter 2 reviews the main aspects of multi-user systems. Chapter 3 presents several knowledge representation techniques, assessing each of them according to their suitability to organize the knowledge associated with multi-user systems. Chapter 4 is an introduction to object-oriented methodology and X Window, and helps explain the choice of the approach. Chapter 5 sets the requirements, in terms of knowledge, to achieve reconfigurable display management in multi-user systems, and, more particularly, in card games applications. In the same chapter, a solution is proposed to organize the corresponding knowledge in a suitable way. The result is a working prototype that is presented in Chapter 6. The tests are discussed and the last remarks deal with other types of applications. The conclusion follows in Chapter 7.

Chapter 2

Computer shared collaborative work

Before reviewing different aspects of multi-user systems, we must define what we understand by the term, knowing that there is no standard definition existing yet [Gru91]. We refer to

```
computer-based systems allowing several users to exchange
information between each other and with machines, in which
the same piece of information can be processed differently
according to a user-specific criteria.
```

One important point to remember is that multi-user systems assume the existence of a common goal or activity between users, and, as such, provide collaboration and decision support.

This type of systems is usually referred to as groupware, collaborative groupware, computer supported collaborative work (CSCW), and multi-user systems. The emphasis is put on user interfaces taking user differences into account, inter-user relationships and intelligent decision support. Some researchers [Ish89] consider that there are two types of groupware, one that is computer-based, and another that is telecommunication-based and can handle information outside of computers. We are primarily concerned with computer-based groupware.

We do not consider as genuine multi-user systems those systems that are simply extensions of single-user systems to support several users, where the user interface is the same for each user. These systems lack an important feature of multi-user systems, which is to provide different data representations in different situations (this point will be further explained below). Their main adavantage is that they are easily implemented, and that they permit to reuse a large number of existing programs within multi-user environments. Because these systems do not provide user-specific data representation, they are called collaboration-transparent software, whereas the "true" multi-user systems are termed collaboration-aware [LL90]. Even though we do not investigate collaboration-transparent software, we believe that they can play an important role in multi-user systems, as they do allow a user to import a single-user program during a multi-user session to show it to his fellow users. An example of collaboration-transparent software is the multi-user system described in [Lee90] which provides a shared drawing surface.

In what follows, we will consider several aspects of multi-user systems: user interface, communications, existing applications, possible architectures and software tools used to build them.

2.1 Multi-user interfaces

Although multi-user interfaces can involve vocal commands and vocal feedback from the machine, most existing systems deal primarily with graphical interactions via mouse and keyboard devices. In what follows, we will focus on graphical interfaces, which have become a de facto standard for interfaces to commercial computers. We review a number of properties that differentiate multi-user interfaces from single-user interfaces.

2.1.1 Private and public workspaces

Most systems use the idea of shared and private workspaces, which was introduced by Engelbart in his Augmented Knowledge Workshop in the 60's [EE68]. The idea is that each user's display is divided into special geometric areas, called private and shared (or public) workspaces, or areas. The way information is displayed depends on the type of workspace it is in. Data located in a private area can be seen by one user only, whereas data in public areas can be seen by everybody (see Figure 2.1). This simple scheme can be enhanced to fit a variety of needs. In our case, we will see that it is necessary to operate more subtle nuances.

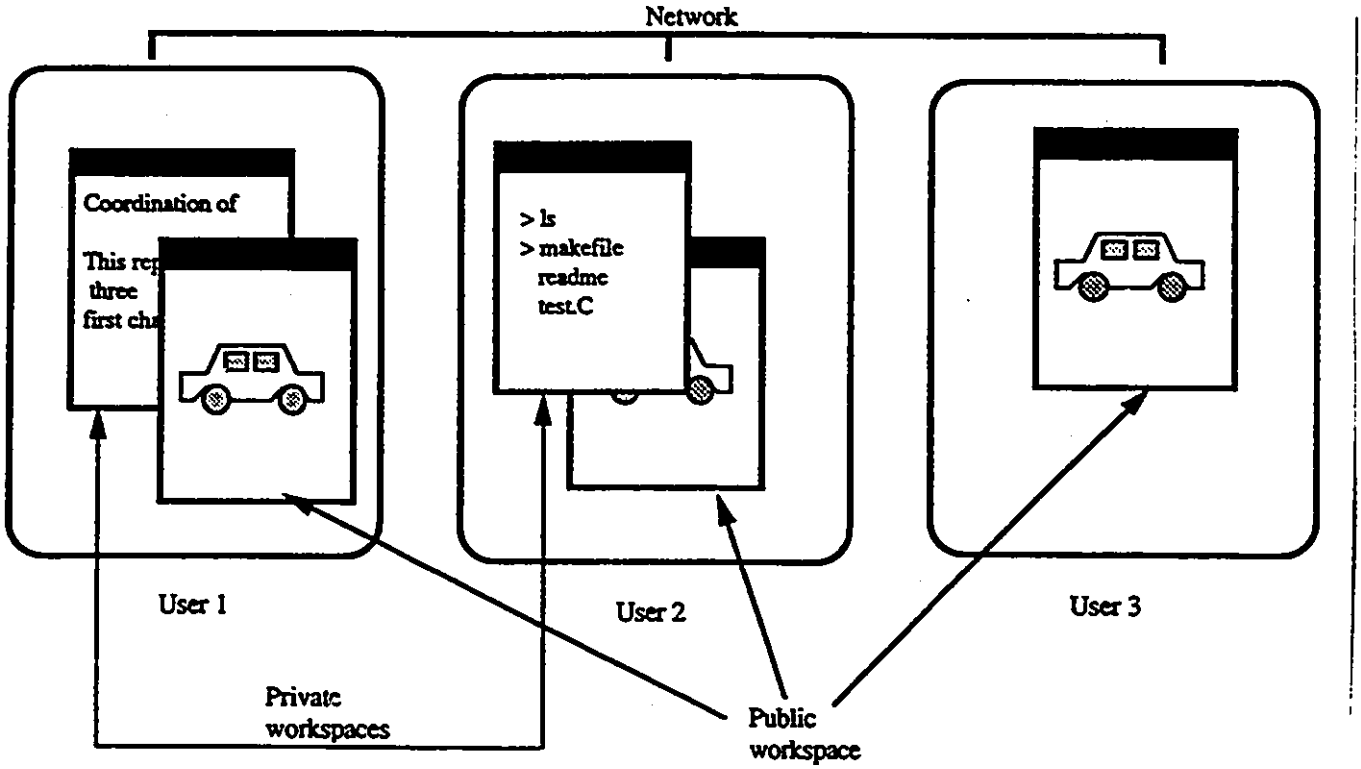

Figure 2.1: An example of private and public workspaces in a hypothetical multi-user system

One possible enhancement is to associate permissions with each area, defining the way information inside this area will be displayed. This amounts to refining the notion of public space, which becomes then a common space, accessible with different permissions according to the display considered. In the same way, private spaces can be viewed as a special case of common spaces restricted to one

person. An example is the Grove multi-user editor [EGR91] where permissions can be set on windows to specify which users have access to them.

A further enhancement consists in defining access control lists for each space: for instance, each space bears a list indicating, for each user, what actions are allowed on what objects or types of objects. With this system, permissions can be set at the level of individual objects for each possible action, each user and each space. We will see later that our system uses similar principles. For instance, in Rendezvous [PHMR91], as well as in [DC91], flexible coupling between workspaces is supported, in particular at the level of access rights to data presentation. In both cases, whether permissions are at the space level or use access control lists, the system should accept modifications to be made during the session, reflecting changes in inter-users relationships.

2.1.2 Different views for different users

Multi-user interfaces should permit two types of specification on a per-user basis: personal preferences of users concerning the "look and feel" of the application, and differences in data representation for reasons internal to the application, which can not be modified by the average user. Having said this, we are aware that one of the approaches to multi-user interfaces is based on the WYSIWIS (What You See Is What I See) principle[SBF+87]. The idea is that every user sees the same thing as every other. However, early experiences with WYSIWIS showed that strict WYSIWIS is not desirable, and needs to be relaxed [SBF+87]. Our argument is that it is always possible to simulate WYSIWIS in the kind of systems we want to design, whereas the contrary is not possible.

As an example of personal preferences, in Diamond [TFC+88], different users can choose to tailor their user interface to their own preferences, specifying characteristics such as text font or window placement.

On the other hand, the same piece of information may have to be displayed differently to different users, as, for instance, in a poker game, where the same card can be displayed face up or face down to two different players. Although necessary, this notion introduces several issues. First, every user should be aware of what the others do or do not see. If they were not, users could be mistaken about what the others actually see, which would lead them to false interpretations of the situation and to wrong decisions. Secondly, it raises the problem of knowing to what extent an individual's display can be independent of the other displays. In this case, we must be careful to keep the representation consistent over all the workstations. Both of these issues will be addressed further below. There is a tradeoff between user's wish to have personalized interfaces and the need for keeping the group coherent [SG88].

For now, let us note that different representations may be necessary because of different hardware capabilities. We will nevertheless not specifically address the problem of designing multi-user software that can be used on several hardware platforms, even though, as will be seen later, the tools we use (e.g. X Window) ensure a broad software portability.

From an implementation point of view, special-purpose layout objects can be developed to obtain different views from the same common data. Examples of such graphical objects can be found in the InterViews package [LVC89], and in the MEL system [Hil91].

2.1.3 Providing information to one user about the others

One of the main differences between single- and multi-user programs is that, in the former, there is obviously no need to inform the user about what he is doing. In a groupware context, clues are needed to keep every participant aware of the group's activity [EGR91]. As was pointed out above, this means knowing what the others are doing, but also, when applicable, whether they are seeing the same objects, and, in some cases, how they are being seen. In the case of private data, however, there are two alternatives: either show the others that some user has a strictly private space, or hide this fact completely. As a particular example of visual clues, it may be necessary to let users tell the difference between shared and private windows, or windows from different conferences [LL90]. However, this approach is not workable in the case of common windows, i.e. windows visible to a group of users, but not to all, as the extra-information needed to indicate who has access to the window is too cumbersome when the number of users is large. Moreover, it does not apply to all types of applications. In our case, in particular, it will appear that it is irrelevant.

As users move about with their pointing device on the screen, it is relatively easy to follow them on the other displays by reproducing these movements on all the workstations. Resulting multi-users cursors, also called telepointers, allow everybody to locate everybody else, as user-specific information such as the owner's name can be easily introduced. Multi-user cursors can also give hints about the owner's activities by changing their aspect according to the current activity. Figure 2.2 illustrates the case of three users, each seeing two telepointers.

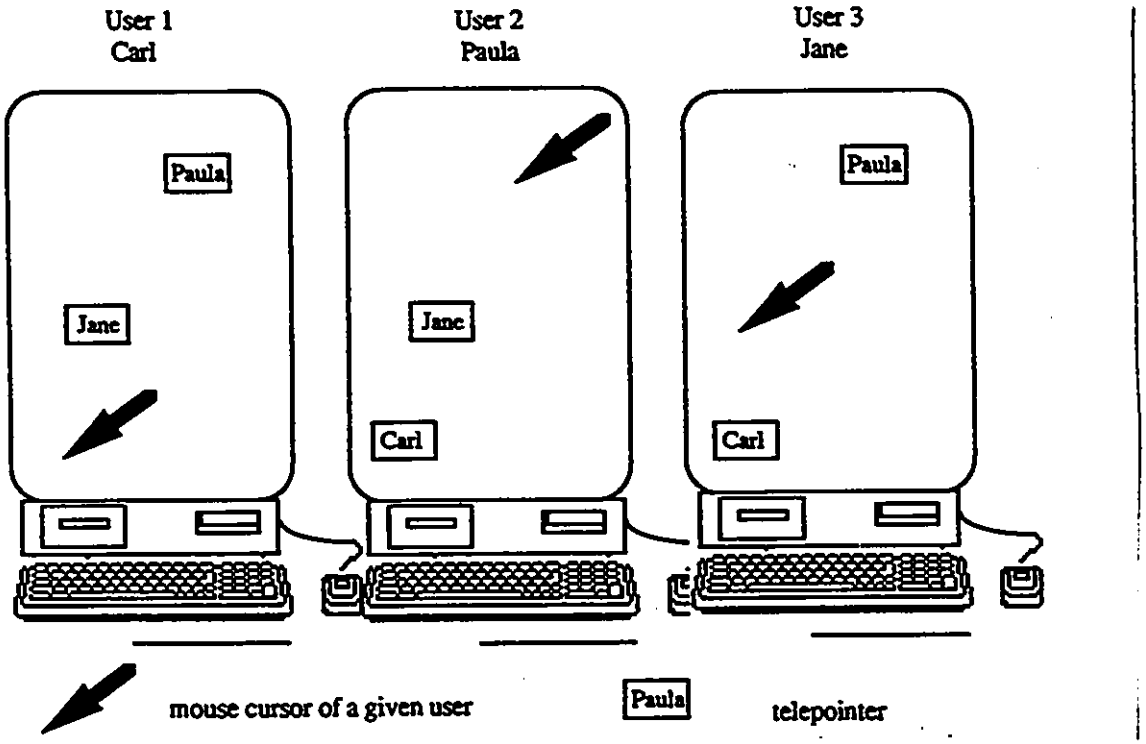

Figure 2.2: Telepointers: an example where the name of each user is associated with the telepointer

One possible refinement takes into account the time delay necessary to display the cursors over a network. Sarin and Greif, for instance, choose to have two cursors per user. The first one is the

actual hardware cursor, and the second one represents the way the others see it. Another possible refinement is to tie telepointers only when their owners are active [SGS8].

A variation of the previously presented telepointing principle consists in a shared pointer, displayed only in shared workspaces, that every user can grab [LL90]. In this case, floor control has to be defined, as is explained in what follows.

2.1.4 Concurrent access to the information

As several users have access to data, some of which is shared, concurrency problems of access to shared data necessarily arise. For data stored in a database, the problems are the sames as in multi-user databases [BG81], or in hypermedia systems [AMY88] [Nie90].

Controlling concurrency can be done by defining a floor control mechanism, in which only one user has the ability to modify data at one given moment, e.g. by first making a floor request. As an example, in Rapport [AHE88], it is possible to restrict the control of the input to one user only. Although the system is very safe and suitable for applications such as a shared blackboard, the authors point out that applications such as interactive games require more natural control mechanisms.

Knowing how much floor control is needed depends thus in a large measure on the type of application. Another example where strict floor management is desirable is a joint editing system in which two users try to delete the same line at the same moment, which could result in two lines being deleted if there is no floor control [AW90].

It is possible to automate the floor control mechanism, or not to have any control at all. Automation can for example consist in giving control in response to some action, such as starting to speak or modifying the position of a shared object [For85]. It is then necessary to determine who was the first requestor, which requires a way of synchronizing the different workstations. Synchronization mechanisms are a well known problem in operating systems, parallel programming languages, and databases (see for instance [Blo79]).

If one chooses no control at all, it is necessary to rely on polite turn-taking, which can lead to situations where some users would be denied access to the data by some others, in particular if the number of participants is large. Systems aimed at making users aware of what the others are doing (e.g. telepointers) are particularly useful in this case. An example of collaborative system with a common drawing area accessible without floor control is [IO90]. In this case, awareness is achieved through live video and audio face-to-face links. It has been noted that collaboration is encouraged when no floor control mechanism is provided [EGR91].

A discussion of floor control strategies for groupware systems is found in [GLACL89] or in [EGR91].

2.1.5 Integrity of the interface over all the users

The integrity problem sets constraints on the user interface flexibility according to the kind of application considered. To highlight this point, let us take two examples. In the first one, we have all the information contained in a single window. It can correspond for instance to a multi-user drawing tool [AW90]. In this case, it is relatively easy to allow each user to change the geometry of his window without any threat to the global integrity.

Now, if we consider an example where several windows are involved and where these windows are somehow linked to each other, for instance by the presence of a telepointer, then the set of windows has

geometric constraints that should not be violated. Firstly, if the size of one window can be changed, should it be permitted to overlap another window ? If yes, then one user will not be able to detect what may happen in the background window. Figure 2.3 highlights the case of information being hidden to user1 because he decided to resize the window w2.

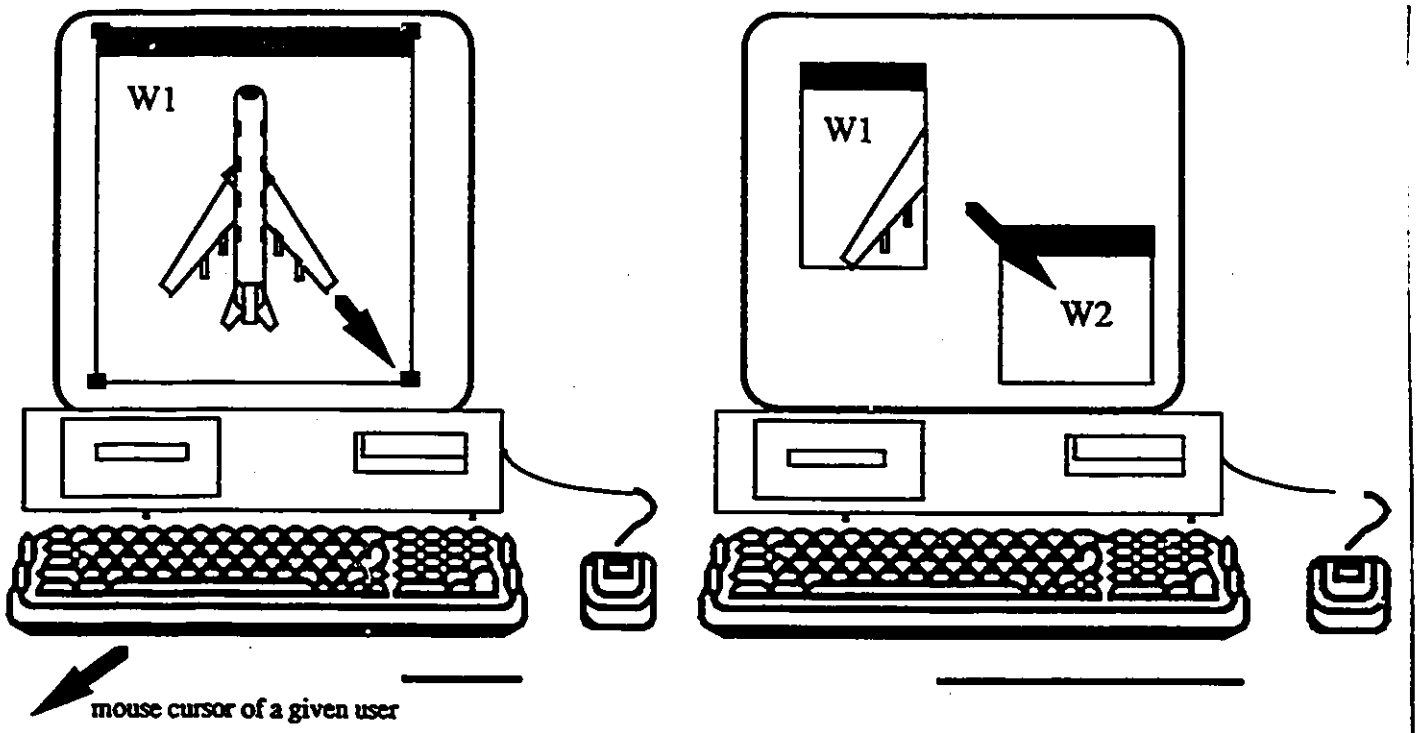

Figure 2.3: Example of the integrity problem: user1 resizes w1 as user2 works in w2- User2 has no way of knowing what user1 is doing

Secondly, should one permit to modify the screen location of some geometrical objects, e.g. windows ? If yes, then, in addition to the problem of overlapping objects that can hide some incoming information, another question arises: as the relative position of objects is modified, their geometrical relationships are modified, making the use of telepointers in particular very awkward, as, instead of having a continuous progression on the display, they will leap from object to object, confusing the user instead of helping him. Figure 2.4 illustrates the problem of coherence between two displays when the geometric layout is not the same. In particular, the question is how to represent the mouse movements between the two windows on user2's display. As a compromise, in [SBK+87], public windows can be positioned at will on the screen, and telepointers appear only inside the windows. It permits more screen personalization and less ability to refer to things by screen position.

Some systems [Gus88] allow both independent window resizing and repositioning, even though the resulting display is not clear. In [Lan88], each conferee can position windows on his screen at will, but this is justified by the fact that the relative position of windows is unimportant in the conferencing application considered.

Thus, the problem of knowing to what extent users should be allowed to modify their interface's layout at run-time is not easy, as, if one chooses to permit window overlapping for instance, it is necessary to devise some feature to keep the user aware of what the others are doing. The resulting

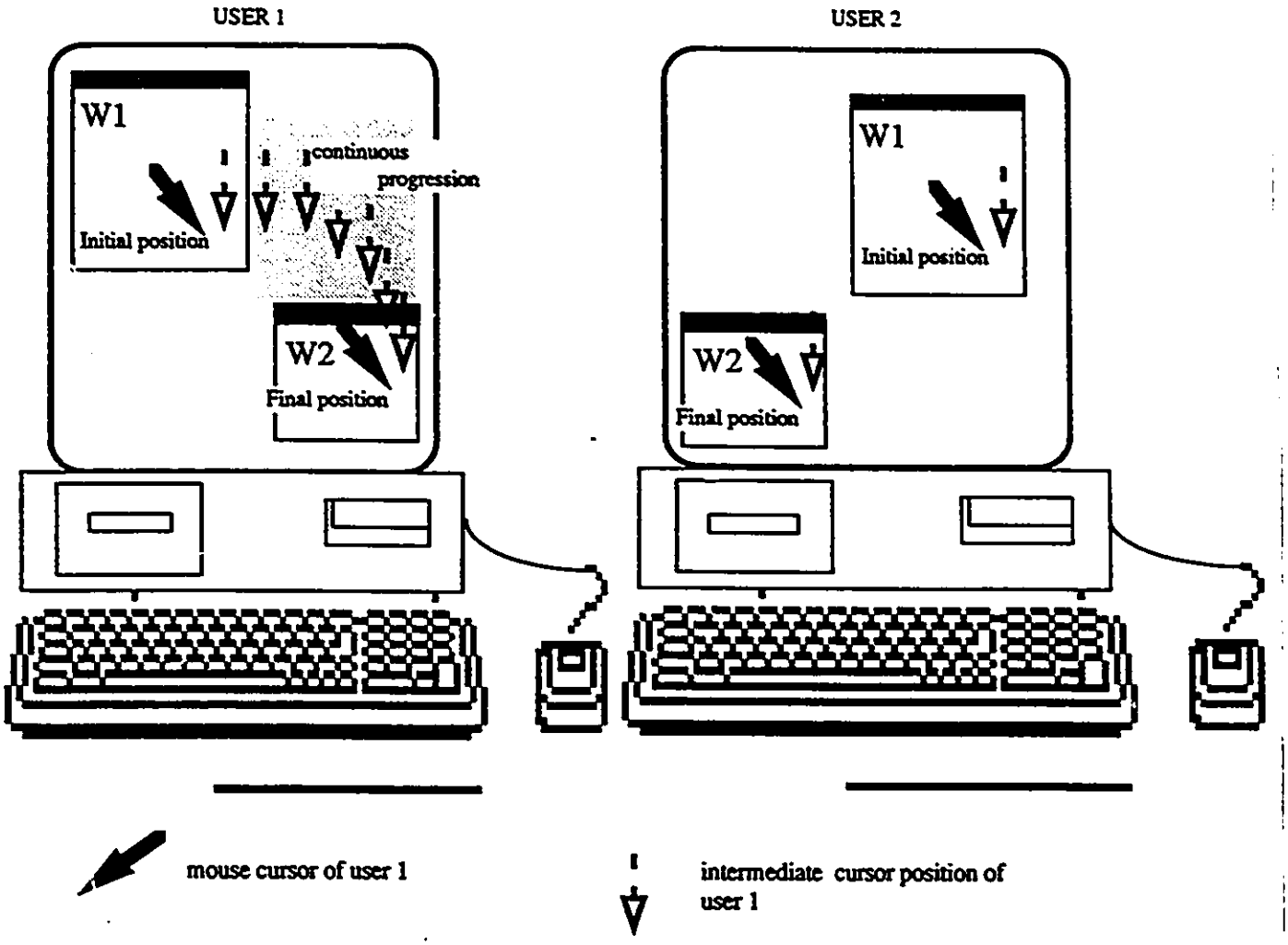

Figure 2.4: User 1 dragging the mouse from w1 to w2, and resulting display for w2

programming complexity may be the reason why several systems do not let the user break the geometrical layout of the interface. In PAGES [MAH90], for instance, screen layout is fixed, and the problem of screen congestion is solved by automatically closing obsolete windows.

There are, however, at least two ways of maintaining the integrity of the displays. In the first one, when someone changes something, it is reported to all the others, ensuring that everybody has the same amount of information at any time. For instance, in MMConf, any change such as window resizing or iconification is reported to all the workstations [CMB+90].

The second alternative consists in allowing limited differences in the layout as are obtained by rotations, translations and scalings.

As an example of scaling, different users may see the same data at different scales to emphasize those pieces of information that are most relevant to them. In hypermedia systems, this is called the fish-eye view [Nie90]. This idea can also be useful to solve the problem of small displays, as will be seen in the next section. In Boardnoter [SBF+87], each user can create several drawing surfaces (i.e. chalkboards) on his screen, but only one of these is normal-sized, for screen space reasons. The others are displayed in a shrunken form. This implementation ensures that all users have a global view of all existing chalkboards at every moment.

As an example of rotation, different users can see the same global structure but from different viewpoints. For example in a card game each user could see his cards at the bottom of the display, and the other players could be placed around him as if they were sitting around him [PHMR91].

A last aspect of integrity deals with the granularity at which changes take place on all the displays after some action is performed by one of the participants. Granularity takes two aspects: the first one is the response time, i.e. the delay necessary for a change to take place on the requestor's display. The second aspect is the notification time, i.e. the delay necessary for a change to be reported to all users. In order to keep the cohesiveness of the group, it is necessary that these changes occur as simultaneously as possible [EGR91]. Concerning the notification time, there are two main alternatives [DC91]. The first one is to report changes as soon as they occurred. The second one is to wait for the user to request explicitly that his modifications be reported to the others. In the latter case, group cohesion decreases as grain-size increases [SBF+87].

2.1.6 Screen space

Screen space is often a problem when dealing with user interfaces. With single-user programs, the classic solution consists in using items such as windows and pop-up or cascading menus. Multi-user systems, that typically involve even more information than conventional programs, make this issue crucial.

One possible solution, as explained in the previous section, is to rescale some screen data according to the fish-eye idea. Iconifying can be very helpful, in particular if it employs dynamic icons, the aspect of which changes according to the actual value of the data they represent. Dynamic icons for single-user programs are used in the interface of the Macintosh [Sch86].

Another approach tried at Xerox PARC [SBF+87] is to separate windows in different sets, called rooms [HC86]. Windows in a given room all correspond to the same task, such as editing or electronic mail. In order to keep global information about group's activity available, an overview room displays miniaturized active images of all the rooms. Moving from room to room is done through doors that connect different window sets.

At MCC [Rei89], one of the users is assigned the task of maintaining the set of windows. This approach does however not seem to be very satisfying: it may be more practical to automate the process, using for instance an expert system.

Scrolling may also be a way of saving space in multi-user systems, but it should be used only with great care. Indeed, if a shared window can be scrolled independently by each user, one looses group cohesiveness. And scrolling is not suited in the case where it is necessary to have a global view of the information at any moment [Cou85].

A last alternative to increase screen space consists in associating several terminal screens per workstation. Although it may seem at first sight that using larger screens would have the same effect, large screens are expensive, and it appears [Cla91] that users like to have several displays to organize their programs in separate functional clusters of windows related to the same tasks. A variation on the multi-display theme is to allow screen switching [Sak90], the drawback of which is to reduce the cohesiveness of the group.

2.2 Communications

In this section, we are primarily interested in desktop multi-user systems, which are usually based on LAN (Local Area Network) technology, as opposed to videoconferencing [Hef84]. Examples of such systems include [CFS9] and [CMB+90].

A general property of multi-user systems is that they require a high degree of connectivity between participants [AHE88]. Another general feature is that they are usually also distributed, and thus share a number of problems with distributed systems (see for example [RSSS90]), in particular concerning networking and modes of interaction (synchronous or asynchronous).

To start with, let us recall that there are two main types of multi-user systems: synchronous ones and asynchronous ones. In synchronous, also called real-time systems, users interact within the application at the same moment. An example could be a software review application, where all participants discuss the same code at the same time (see for example [MMM91]). Note that synchronous systems do not necessarily actually implement synchronous network communications. In asynchronous, or non real-time systems, people interact with the system at different times, without waiting for immediate feedback from the other users (e.g. electronic mail).

Generally speaking, protocols for multiparty conferences should provide the following services [ZW90]:

- initiate the conference,
- request to join an ongoing conference,
- invite a new user to join,
- leave a conference,
- remove a participant,
- obtain data transmission privileges.

2.2.1 Integration of different media

Standards

Most multi-user systems are also multi-media, and need to treat different kinds of data: text, graphics, video and voice. As these media have different properties, different protocols should be developed accordingly. Several standards already exist, in particular ODA, X400 and RAVI. ODA (Office Document Architecture) is a model for data format to build, revise and format multi-media documents [HKN89]. The X400 Recommendations provide a standard for computer-based messaging systems (i.e. E-mail). CCITT (Consultative Committee on International Telegraph and Telephone) and ISO (International Organization for Standardization) are studying extensions to X400, in particular for joint editing, and to include ODA documents [Wil87] [doa88]. They are also considering the introduction of RAVI (Representation for Audio/Visual Interactive applications) that proposes an interchange format as well as a set of operators for document handling, and addresses the problem of communication between different distributed subsystems [OSKQ90].

Text and graphics

Text and graphics, which are very sensitive to data loss or damage, require reliable communication mechanisms, such as messages, possibly improved by virtual circuits [SG88].

Voice transmission

Voice signals can be transmitted using datagrams, as they can tolerate some data corruption. As multi-user systems typically require sending data to a group of machines, multicasting can be fruitfully implemented [Bar90], as in Etherphone [Swi87]. Etherphone uses also datagrams and provides facilities to organize telephone-based conferences, including call filtering, broadcasting, and distributed meetings.

Video transmission

Similar to audio signals, video signals demand much more network capability. This is why, by now, integrated multi-media real-time systems are more advanced in the field of voice transmission. This situation is changing with more powerful CPU's and increased network capacity [Nic90].

A comparison of several protocols for multi-media communications on LANs can be found in [ZW90].

2.2.2 Synchronization of data streams

In a multi-media system, a number of media (e.g. voice, audio and video) can be expressed as a function of time, whereas others (e.g. pictures, text, drawings) are time-independent [Ste90]. In a multi-media multi-user system, synchronization problems arise at several levels. Firstly, different copies of the same data should reach all workstations synchronously, as was already pointed out earlier. Secondly, when multi-media documents are retrieved, it is necessary to synchronize their different components, that can be located on different machines in a distributed environment. Different problems correspond to the case of documents that have to be presented sequentially or simultaneously.

In [Scr90], the problem is stated in terms of concurrent processes to what a set of constraints representing the synchronization relationships is applied.

Another approach [Ste90] uses an object-oriented representatio. of multi-media documents, and analyzes the requirements through a comparison with synchronization in concurrent languages.

2.2.3 Managing heterogeneous hardwares

To ensure maximum flexibility, multi-user applications should be capable of being run on different computers and networks. This can be achieved through the use of standards in the fields of data transmission, data representation and user interface. As far as user interface is concerned, the solution seems to reside in products such as X Window [SG86], which provides a hardware-, network- and operating system-independent tool to create user interfaces. Several researchers, among which [BS90], believe that the X Window protocol will be the first windowing protocol to be added to the ISDN standard. The same effort should be devoted to the other fields relevant to multi-user systems.

In the audio domain, for instance, the VOX audio server, developed at Olivetti Research Center [LLA+89], provides device-independent,
network-transparent access to audio hardware, allowing it to be shared between several clients. Olivetti plans to extend VOX to handle video too, and to use the same marketing strategy as that used in the case of X Window, which is to leave the product in the public domain.

2.3 Existing applications

In this section, we review a number of systems covering a number of applications and with varying capabilities.

2.3.1 Electronic mail systems

There are roughly two types of electronic mail: standard mail, and advanced electronic mail systems. In standard mail, users exchange textual information in an asynchronous way. The problem with these basic systems is that users can become quickly overflowed by information upon which they have no control. In more advanced systems, intelligence can be added to automatically process incoming messages. In the Information Lens, for instance [MGT+87], it is possible to reroute or classify messages based on their social or economic meaning. In another system, developed at the University of Milan [GS90], if-then rules can be built to filter incoming messages and/or trigger some predefined actions.

In CCWS [PGLAC+85], a protocol has been developed to manipulate multi-media mail containing graphics, text and voice.

2.3.2 Computer conferencing

The distinction between computer conferencing and multi-user systems is rather subtle, as many multi-user systems include some form of meeting support: one may say that multi-user systems are more concerned with the collaborative aspect of communication, and with user-specific adaptivity. There are roughly three types of computer conferencing: real-time, videoconferencing, and desktop conferencing [EGR91].

Real-time computer conferencing is equivalent to what we called multi-user systems, as the terminology evolved from real-time computer conferencing to multi-user systems, groupware and CSCW (Computer Shared Collaborative Work). It is generally admitted that real-time computer conferencing does not include video, as opposed to videoconferencing and desktop computer conferencing [EGR91]. Moreover, conferencing systems present special features such as voting tools and chairpersons [SG88].

Videoconferencing is a video-based version of teleconferencing [Hef84], both of which requiring specially equipped rooms which make their use very awkward. Teleconferencing, in its simplest form, connects different groups of users at distance via audio and data links. However, newer systems make teleconferencing more accessible by providing workstation-based facilities (see for instance [NTH+90]). New compression technologies now permit good video quality at data rates much lower than was necessary at the beginning of the eighties. Typically, 384 kbps (kilobits per second) or even 112 kbps are now sufficient where previously 1.5 Mbps were required [Bak89]. Simultaneously, reduced costs of electronic components and telecommunication services make teleconferencing rooms available to many corporations. Nevertheless, teleconferencing does not, in its standard form, let participants share data or windows. This is where desktop conferencing comes into play.

Desktop computer conferencing combines the features of real-time computer conferencing and teleconferencing. One example of such a system is the Rapport system developed at AT&T Bell Laboratories [AHE88]. Rapport uses two types of application programs: single-user programs made visible to every participant, and true multi-user applications. Typical applications include a drawing program, a facsimile annotation program and a terminal emulation program.

Other examples include the MMConf system [CF89], and TeamWorkStation, developed at NTT Human Interface Laboratories [IO90]. An original feature of TeamWorkStation is the use of a video overlay technique permitting to combine computer screens and desktop images such as handwritten or typed documents. Generally speaking, overlay windows, which can be implemented with invisible standard windows, can be useful for annotating documents during a multi-user session, when only temporary changes are wanted [Pat90].

2.3.3 Joint editing

Multi-user editors are of two types: asynchronous and synchronous. The principles involved are very similar to those used in hypertext. Texts are usually divided into several logical segments, and users are given read or write access to each of the segments. A locking mechanism is necessary to ensure that only one user has write access to a given segment at a given moment. Some products include the Collaborative Editing System [GSW86], Shared Books [LH88], and Quilt [PL89]. Quilt is an asynchronous system where each user can choose to use his favourite editor to process documents. It provides facilities for annotations.

Joint editing can also be applied to desktop publishing. The difference with multi-user editors is that documents can include other media than text, and that extended document processing and formatting facilities are available. An example of a publishing package is Concordia [Wal88]. Concordia uses hypermedia concepts, object-oriented graphics, WYSIWYG (What You See Is What You Get) editors, and configuration tools. However, even if the system is designed to allow several people to maintain the same document, writers do not have simultaneous access.

2.3.4 Group decision support and coordination systems

In group decision support systems, the goal is to improve the productivity of a group by speeding up the decision process or by improving the quality of the decision. Classic tools include voting, ranking and brainstorming facilities. Such systems are usually implemented in so-called electronic meeting rooms, which contain audio and video equipment, large computer displays, as well as possible individual conference table workstations. An example is the Planning Lab at the University of Arizona [AKN86].

It is interesting to note that studies by Doyle and Straus [DS84] show that group brainstorming is less efficient than individual brainstorming. According to the authors, the reason seems to be that everybody has to wait for the latest idea to have been explained before being able to add a new one. By enabling participants to add ideas simultaneously, multi-user systems could permit to enhance overall productivity.

Similar to group decision support are coordination systems, the goal of which is to help users coordinate their efforts toward a common goal. An example is PAGES [MAH90], a system in which users can cooperate by interchanging and sharing form objects (a form is equivalent to an object in object-oriented terminology. See Chapter 4). The principle is to provide a set of base form objects in order to be able to develop groupware systems rapidly. Each user is assigned an user agent, i.e. a form manager controlling the user's private form base, and providing communication with other agents. PAGES is one of the few examples of systems having some reusability concern in mind.

One of the major realizations in the field of multi-user systems designed to study face-to-face meetings is Colab [SBK+87] at Xerox PARC, which developed the concept of WYSIWIS. Colab uses several tools, e.g. Cognoter to produce presentation material on a large display screen for several participants located in the same room.

2.4 Architecture

2.4.1 Centralized and replicated architectures

There are two main architectures for multi-user systems. In centralized architectures, a single copy of the application distributes outputs via a conference manager to a set of local servers located at each workstation. The role of the conference manager is to handle inter-users communications. Window servers at each site send input back to the application via the conference manager. In replicated architectures, each workstation runs its own copy of the application and of the conference manager. Each local manager is able to send inputs to every other participant. Output from each copy is visible only locally. Outputs are synchronized because conference managers make sure that all the copies have the same input. Figure 2.5 illustrates the principles of both approaches.

2.4.2 Advantages of replicated over centralized systems

There are at least three advantages. Firstly, replicated architectures theoretically offer increased performances. Indeed, replicated systems are less sensitive to network latency, as people interact with local copies of the application, and less data has to be transmitted between workstations.

Another advantage is that, unlike centralized ones, replicated systems can be used with kernel-based windowing systems [CMB+90]. By kernel-based, as opposed to server-based, one means that

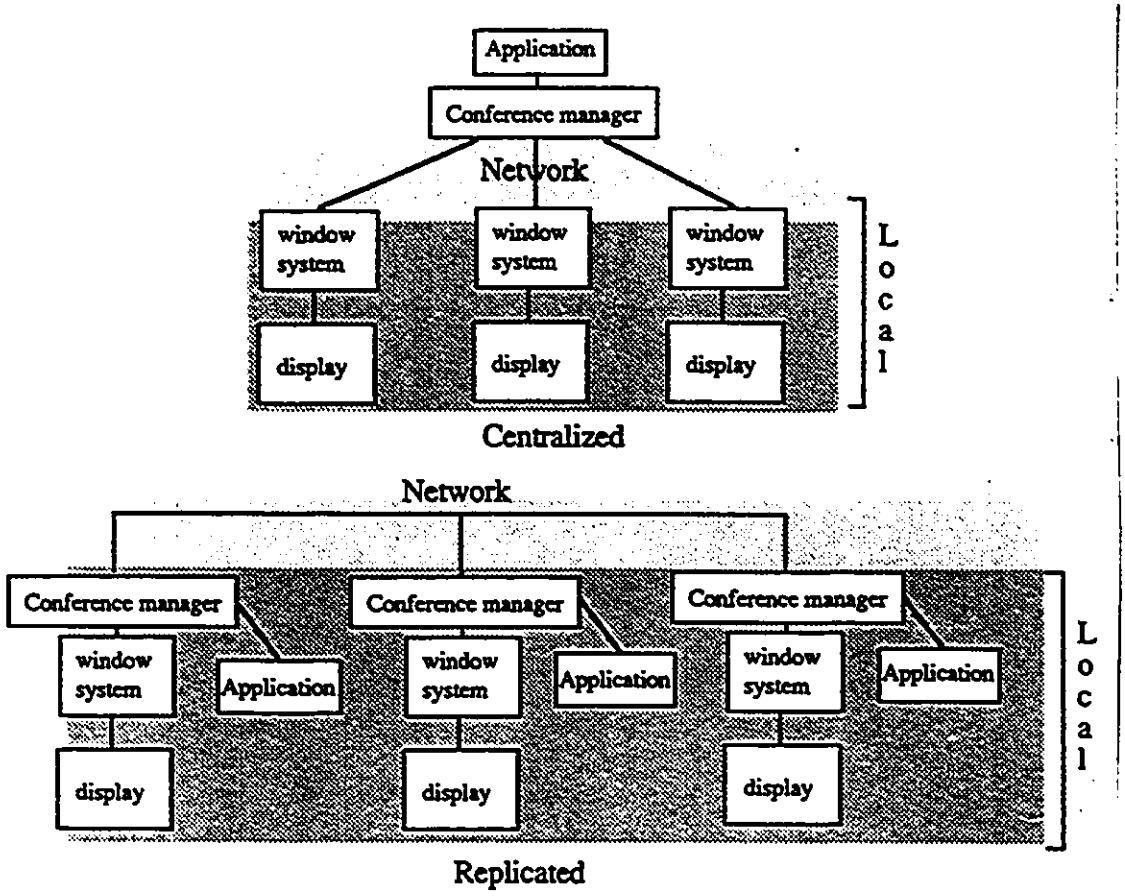

Figure 2.5: Centralized versus replicated architecture

input and output are handled via function calls. Centralized systems are limited to server-based windowing systems.

The third advantage arises from the fact that each user interacts with his local copy of the application only. This allows for more flexibility in terms of software or hardware differences between the workstations. However, it should be noted that per-user customizations increase the risk of falling out of synchronization [CF89], which is, as will be seen below, the major drawback of replicated systems.

2.4.3 Disadvantages of replicated systems and advantages of centralized ones

Perhaps the greatest problem with replicated systems is to keep the workstations synchronized. Synchronization can be lost because of different initial states, or because of misordered input events. Misordering can for instance result from bad communications between workstations. Loss of synchronization can also result from different machine speeds or loads. Also, some applications can not be included in replicated systems because of their non-deterministic character. As these applications can behave differently according to the actual timing of the sequence of events that drive them, they are extremely prone to synchronization problems [CF89]. To guard against all these potential risks, it is necessary to devise a mechanism capable of resynchronizing the workstations.

Another issue is how to include late conferees to the session. There must be some way of knowing the state of the system in order to initialize the new workstation with the correct parameters. State information can be found at each copy (of the application) level and at the server level. Thus, part of the program state could be downloaded to the new copy. In a centralized architecture, the updating is much simpler, as there is only one copy of the application: the only component to bring up to date is the windowing system [MAH90].

If system calls are made from within the application, it is necessary that they have the same significance on each workstation. The question is how to ensure this.

A last problem deals with the fact that running a replicated system on a given station requires that a copy of the application be already on that machine. If it is not there, it will have to be installed through the network. Running a centralized system using a standard windowing system is a lot easier.

Centralized systems offer two main advantages: they are simple to implement, and they benefit from the wide distribution of server-based windowing systems.

2.4.4 Architecture for voice and video transmission

As a lesser number of applications involve voice and video, it is not possible to make generalizations upon what kind of tools are used. As an example, however, the TeamWorkStation project [IO90] uses a separate NTSC network for video, and still another network for voice.

In the Integrated Interactive Intermedia Facility, a desktop conferencing system developed at Xerox's Europarc [BM90], audio protocols match closely the MIDI standard, and audio and video signals are transmitted on different cables in an Ethernet-based network.

In Rapport [AHE88], a separate network for voice is used in addition to an Ethernet for data. The voice network provides full duplex communications, and connects each participant to every other, in order to provide a "natural" conferencing environment. A similar architecture is used for video.

2.5 Development tools

2.5.1 Tools to create multi-user interfaces

There is currently no commercial special-purpose tool available to create multi-user interfaces. However, there are tools to create single-user interfaces, that can be used to build multi-user systems. These tools are described in Chapter 4.

At Bell Core, an object-oriented programming language built on top of Common Lisp is being developed to create multi-user interfaces. The system, called MEL [Hil91], uses standard classes which can be reused in different applications. MEL implements a system that is able to maintain one-way constraints among arbitrary values. One-way constraints correspond to constraints systems (see Chapter 3) where, for each property, there is at most one formula calculating its value [Lel88]. This system is used to control the objects on the screen.

It is to be noted that multi-user toolkits should be more than simple extensions of existing ones. For instance, it is reported [Rei89] that scrollbars are more distracting than useful in group applications. Thus, group toolkits should introduce new graphical constructs, such as the transparent windows for annotation seen previously.

2.5.2 Common programming environments used

In the newer systems, object-oriented techniques are quasi automatic, along with UNIX operating system and local area networks, in particular Ethernet and TCP/IP [tcp] [ip85] networking environment.

User interfaces are often built with the X Window system, less frequently with other network-based windowing systems like NeWS [Sun87] (see for instance [Gus88], [AHE88] and [Pat90]). Programming languages, as can be expected, include C and C++, which provide the speed required by real-time conferencing.

2.5.3 Creating multi-user applications using a virtual terminal protocol

The idea of virtual terminals was introduced by Sarin and Greif in 1985 [SG88]. In the basic approach, the output from a single-user program is sent to a virtual terminal program, whose role is to redistribute this ouput to a set of workstations. Similarly, input from one of the workstations is directed to the virtual terminal controller which forwards it to the single-user program. The principle can be seen in Figure 2.6.

The obvious advantages of the virtual terminal approach are that it is not very complicated to implement, and that it can thereafter be applied to any single-user program. However, the simple version of Figure 2.6 does not permit to implement such features as floor control, private/public spaces or user-specific views. Even though it is possible to add all these features, which leads to more interesting applications from a multi-user perspective, creating different views with a virtual terminal protocol is extremely tricky, as the interactions between the application and the virtual terminal are very low-level.

Many implementations of the virtual terminal idea, as can be expected, use X Window as a basis. Examples include work done at Bell Communications Research [Pat90], at Olivetti [LLA+89], and Shared X at Hewlett Packard [Gus89]. A general concern is to find a standardized virtual terminal protocol which would be a precondition to a widespread use of pseudo multi-user systems [Pat90].

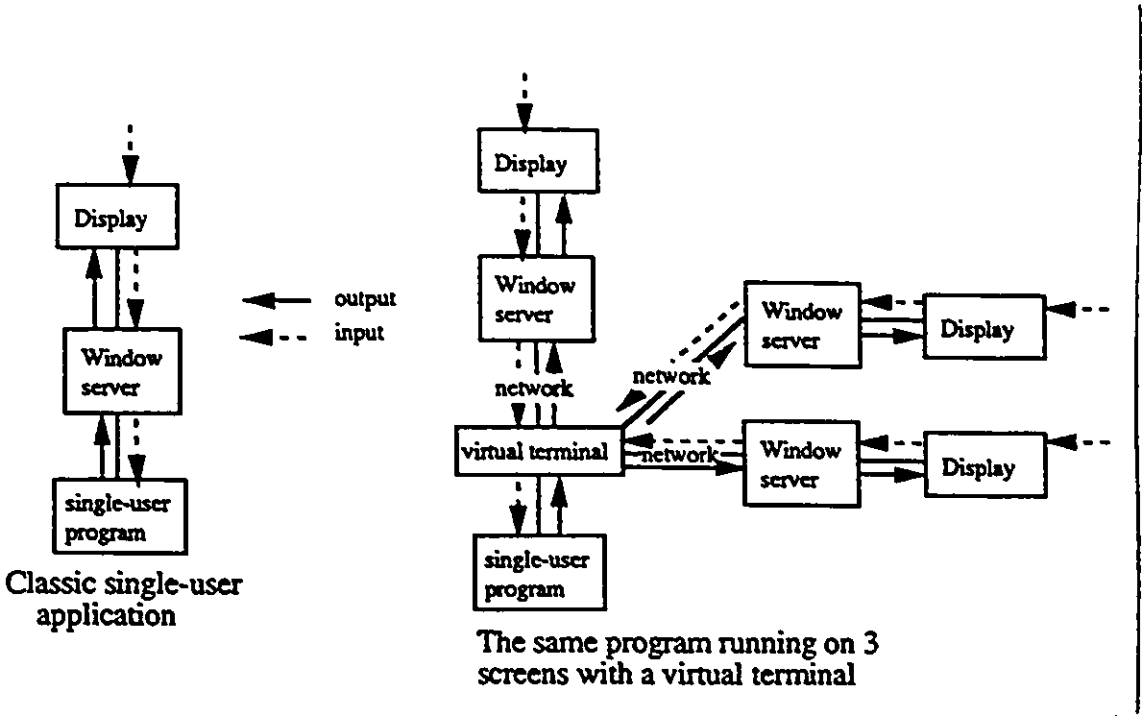

Figure 2.6: The virtual terminal principle

An example of a multi-user system permitting to include any single-user keyboard-driven program into a conferencing session can be found in [AWGN88].

As a conclusion to this chapter, we will first summarize the main issues involved in multi-user systems design. As far as floor control is concerned, one has to decide between extensive control, which can be boring from users point of view, and no control at all, which may not be always possible. In real-time systems, the granularity of the system should correspond to acceptable response times.

Concerning workspace management, there is probably no solution suitable for all applications, even though there are nevertheless several general ideas: shared and private workspaces, user-specific representation of information, geometrical integrity of the interface over all the workstations, and mechanisms to make each user aware of the others.

New tools have to be researched to build multi-user interfaces, keeping in mind that what is good for single-user programs is not necessarily adapted to multi-user environments. Also, new graphical constructs may be necessary.

In terms of architecture, centralized systems, although theoretically slower and less flexible, are more workable than their replicated counterparts. This is the approach that we choose, as will be seen later. Concerning single-user programs in multi-user environments, it is likely that multi-

user systems will be adopted by a large community of users only if they provide a continuity with existing environments. In that respect, groupware applications should include facilities to run private applications and to extend current single-user applications to group use.

We turn now to the issue of knowledge representation, keeping in mind that our goal is to apply it to display management.

Chapter 3

Knowledge representation

If we are to model the relationships between different users, as well as to represent the knowledge associated with a particular application, we need to choose a knowledge representation technique. We review the most popular methods, namely, logic, semantic networks, frames, and rule-based systems. In each case, we will try to consider them with respect to our eventual goal.

3.1 Predicate calculus and logic

Predicate calculus, or first-order logic, is at the origin of several other knowledge representation techniques.

A predicate is an expression that can take two values, e.g. true or false. Predicate calculus itself is based on propositional calculus (see [Tan87], Chapter 6, for an introduction to propositional calculus). In first-order logic, predicates can have objects as parameters, but not other predicates. This type of logic makes up the frame of Prolog, developed in 1970 by A.Colmerauer and R.Kowalski, which is one of the main languages in the field of logic programming [SS86]. Two important concepts in Prolog are unification and resolution which are used to solve problems presented under the form of queries. As we will need a simplified forward-chained unification and resolution procedure, it is interesting to present briefly what it consists in.

3.1.1 Resolution

The resolution principle [Rob65] can be presented as follows. To start with, we define a literal as a predicate (e.g. P) or a predicate preceded by a negation sign (e.g. ˜ P). A clause is defined as a set of literals connected by "v" (i.e. or) signs. The resolution principle states that, for two clauses L v M, and ˜ L v P, it is possible to derive M v P. One notices that, as resolution requires clauses, and as predicates can be written in formulae involving implication signs or quantifiers, it is necessary to transform formulae into clause form. This is a deterministic process that can be done automatically ([Tan87], Chapter 6).

Resolution alone is not sufficient to determine whether a set of wff (well formed formulae: see [Tan87] Chapter 6) is inconsistent (i.e. implies a contradiction) or not. Unification is needed.

3.1.2 Unification

Unification, also called pattern-matching, is the process of finding a set of possible substitutions between a set of facts and a formula containing a number of variables. When a match is found between the predicate symbols of the formula to unify and one of the facts, it is possible to substitute the arguments of the predicates in all the relevant literals.

In order to select rules and facts effectively, clever search techniques need to be designed. Various enhancements of the two basic methods, breadth-first and depth-first search, permit to cut down the number of necessary computations. Depth-first search consists in examining each solution completely before trying another one. This technique is usually associated with backtracking, as in Prolog. Backtracking is the process of trying a new substitution after realizing that the previous one does not lead to a solution.

In breadth-first search, all the solutions are examined at the same time. Heuristics that improve the efficiency of the base techniques often use evaluation functions whose nature depends upon the problem to solve [Tan87]. Figure 3.1 presents depth-first, backtracking and breadth-first search.

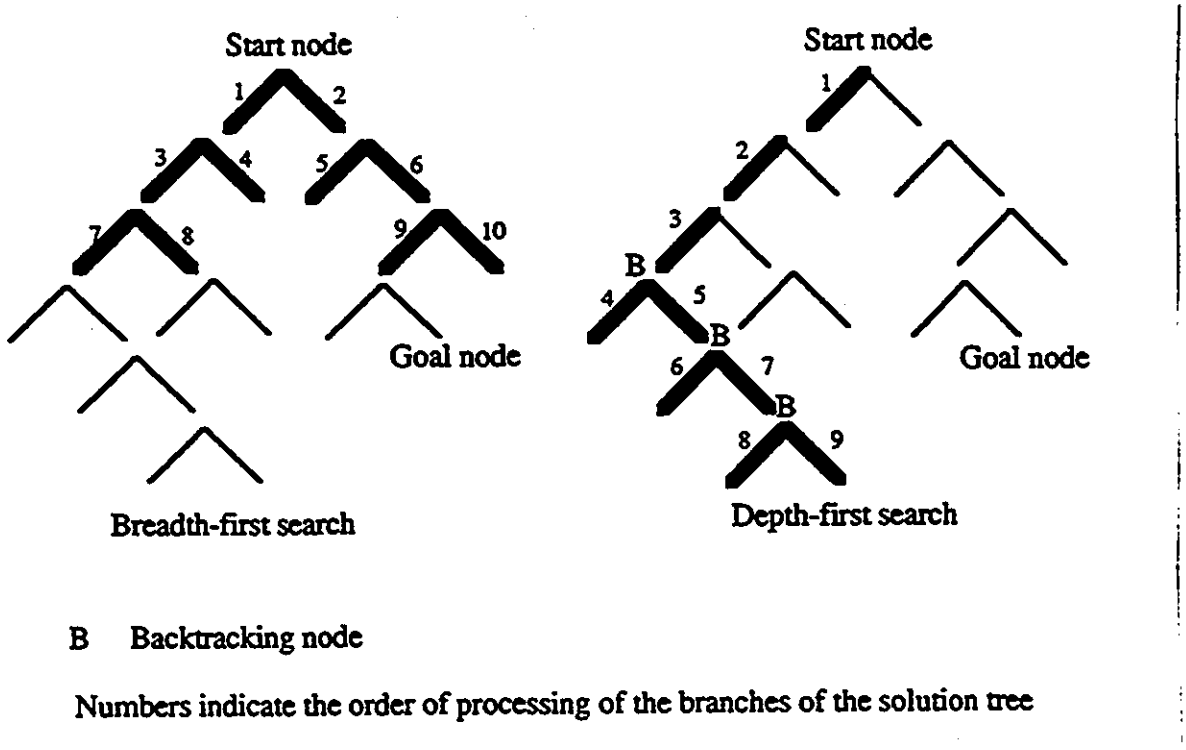

Figure 3.1: An example of depth-first search, breadth-first search, and backtracking

Some of the major appeals of Prolog are its declarative programming style and great expressiveness which allow to describe some real-world knowledge in very few lines. Programs are clear and concise. Another advantage is that most expert systems's inference engines can be derived easily from that of Prolog [Sim85].

It is generally admitted that Prolog belongs only to the first generation of logic programming languages. There are some more complex types of logic, such as second-order logic or temporal logic, which are more restricted to the research field [All83]. One promising field of investigation is non-

monotonic logic, where it is possible to add or delete rules and facts during the reasoning phase, in the same manner as humans generally do [Tan87]. Another example of a logic language is OBJ [F+85], which is based on lambda calculus, a mathematical frame developed mainly by A.Church in the 1930s.

With logic, it is possible to translate events or facts occurring in the world into predicates. The main drawback of logic is the complexity of the resulting computations. This accounts for the low performances of logic languages like Prolog, that prevent their use in real-time systems involving a fair number of predicates. However, it is quite possible to use the expressiveness power of logic to design prototypes, and to rewrite the working system in some faster language.

Another drawback of logic is that facts and rules are written down in a non-organized way. Issues include knowing what order should be chosen to try rules, and how not to test the preconditions for predicates one knows are not relevant to the current goal. Other types of knowledge representation technique, as will be seen next, help use predicates in a more efficient manner.

3.2 Rule-based systems

Rule-based systems, also called production rules or production systems, consist of a set of rules, a database of facts, and an inference engine (see Figure 3.2). Knowledge is represented by the set of rules acting on the database of facts. Rules, or productions, are usually written in the following EMycin-style [VM81]:

```
if ''premise1'' and ... and ''premisen'' then
''conclusion : certainty''.
```

Certainty is used to specify how much confidence one has in the conclusion being true. From this model of rule syntax it is possible to derive new variations as needed, including for instance several conclusions.

To use rules, one needs a reasoning mechanism, i.e. inference engine, to be applied to the database of the production system. There are two main techniques: forward chaining and backward chaining. In forward chaining, the algorithm tries to match the premises of a rule. If it succeeds, the rule is fired and the conclusion can be integrated as a new fact. In backward chaining, the algorithm starts from the final goal and tries to find rules whose conclusion would be the same provided their variables are instantiated to suitable values. Thus, production systems need resolution and unification strategies similar to those explained in the section on predicate calculus. Many expert system shells, in particular EMycin, use backward chaining, like Prolog. Some of the most popular systems using forward chaining are the members of the OPS family [BFKM85].

3.2.1 Metarules

Logic programs and production systems are inefficient because of the search associated with the resolution of predicates. This drawback can be reduced by keeping the number of predicates small enough. When this is not possible, a standard technique consists in breaking the set of rules into small subsets, each one corresponding to a specific context, usually called a subgoal. Special rules, called metarules, allow to move from goal to goal. In this respect, metarules are a way of organizing rules in different functional sets [Dav80]. Metarules also simplify the processing of complex information.

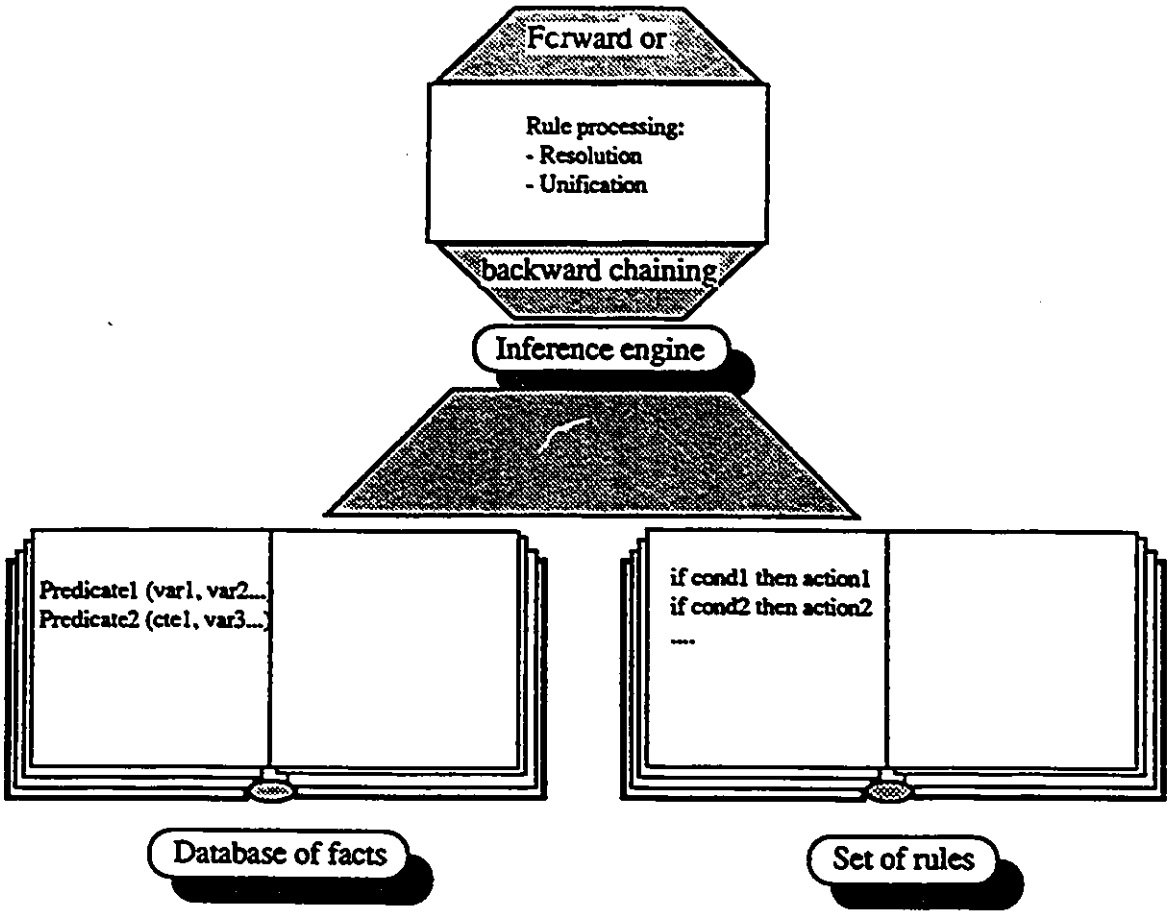

Figure 3.2: Components of a rule-based system

Another attractive feature of metarules is that it is normally not difficult to add them to an existing production system [Tan87].

Examples of successful production systems using metarules include XCON, also known as R1 [McD80], a system capable of configuring VAX 11/780 computers and implemented with OPS5, and Mycin [VM79], for diagnosis and treatment of blood diseases.

Metarules partially solve the problem of lack of control structures in production systems. Indeed, rules are simply added one after the other, which is also one of the strengths of production systems. As only very few programs can afford this lack of control, the current trend is to build hybrid systems including constructs such as iterations, function calls or recursion [Tic87].

Constraints are another way of using rules that is related to the idea of metarules. A constraint can be stated as a set of conditions and actions that has to remain true [Lel88]. As soon as one of the conditions is violated, rules indicate how the system has to update itself. In particular, constraints are used in computer graphics, for scene management or to manage a set of windows on a display. Graphical constraints were used for the first time in Sketchpad [Sut63].

3.2.2 Context-free grammars

As will be seen in Chapter 5, we will have to define a simple language in order to write down the rules for an application. The purpose of this section is to provide the information necessary to set this language against the background of the most popular type of grammars used in computer science.

In 1956, Chomsky proposed a classification of grammars in four different families [Cho56]. Context-free grammars, that correspond to the second class, are characterized by the fact that the left-hand side of each production rule consists exactly of a single nonterminal symbol. Nonterminal symbols are symbols that can appear in partially derived forms of a sentence, but not in an actual sentence. As a whole, the grammar includes a finite set of nonterminal symbols, a finite set of terminal symbols, and a finite set of production rules. The right-hand side of each production is a list of terminal and nonterminal symbols.

Later, circa 1960, BNF (Backus-Naur Form) was used as a notation for context-free languages. Thus, programming languages corresponding to context-free grammars can be defined using BNF notation with Chomsky's mathematical background [Nau60].

3.2.3 Advantages and disadvantages of production rules

Production rules constitute the most widely used knowledge representation technique employed in AI [OB85]. Their advantages are as follows. Firstly, rules are well suited to transcribe knowledge about the world, as man usually tackles problems in terms of condition-action schemes. Secondly, the separation between facts and rules makes it possible to modify the database of facts without changing the system's behaviour. The third advantage is flexibility: adding or removing rules is easy. Lastly, providing explanations to the user about the behaviour of a production system simply resorts to listing what rules were fired.

Among the disadvantages, large production systems are computationally inefficient. This can however be solved by using metarules, as was seen earlier. Another drawback is that it may prove extremely difficult to understand the behaviour of the system by looking at the rule set. Also, as rules are limited to relatively simple conditional statements, they may prove unsuitable to express some

problems, in particular when knowledge can not be separated into small pieces. A last drawback is that efficiency of searching depends upon the order in which rules are written in the rule base.

However, in the cases where knowledge can be partitioned in small chunks, and actions to be performed are reasonably simple, production systems are a good choice [OB85].

3.3 Semantic networks

Semantic networks, or semantic nets, were invented by Quillian in 1966 [Qui68]. From this initial work, developed to process natural language, the similarity existing with the associative aspect of human knowledge made semantic nets applicable to a wide range of problems.

In the case of natural language processing, a semantic network is built by representing each word of a sentence as a node. Nodes are linked by labelled arcs actually representing the knowledge (see Figure 3.3). There are several alternatives when it comes to decide what type of information should be recorded in the label. In simple networks, the label can specify whether the node pointed at is a grammatical object, a subject, or a recipient. In more complicated ones, it may be necessary to take into account the fact that the same word can be used in different sentences: the network has to represent the fact that the sentences are not identical, and to be able to retrieve the nodes of a given sentence.

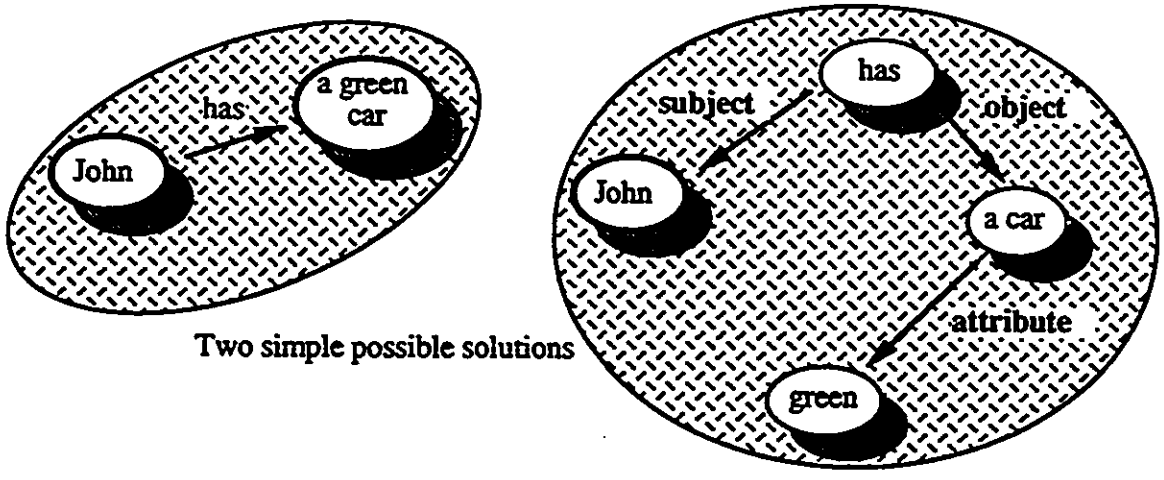

Figure 3.3: A sample semantic net representing the sentence: John has a green car

One of the advantages of semantic networks is the little redundancy, resulting from the use of pointers (i.e. arcs) between data elements (i.e. nodes), occurring in representing the information. Semantic nets are more flexible than production rules in that they can short-circuit a series of atomic

steps with the addition of a new arc. Also, they are better suited than rules to represent hierarchies [OB85]. Nevertheless, their main drawback is that, in order to manipulate a semantic net, one needs an algorithm, similar to the inference engine used in production systems. But it is computationally much more expensive to move through a net than to find a matching rule.

To make searching more efficient, however, it is possible to partition the network in several subsets, in the same way as metarules can be used to break large production systems. Partitioned semantic nets are very similar to frames, which are presented next.

3.4 Frames

Frames were intoduced in 1974 by Minsky [Min75]. A frame is a data structure meant to represent a special situation. For instance, a frame to register new employees can contain fields corresponding to relevant knowledge such as employee's name or age. Each field is called a slot (see Figure 3.4). Frames can also include functions to be applied to data, some of which are to be applied only in the case some assumptions are verified. Thus, it is quite possible to write rules inside frames expressing conditions on the frame's data. To use standard terminology, this means that frames permit to represent both declarative and procedural knowledge. Declarative knowledge is concerned with facts and relationships between frames. Procedural knowledge is concerned with conditions to do something and how to do it [Hu89].

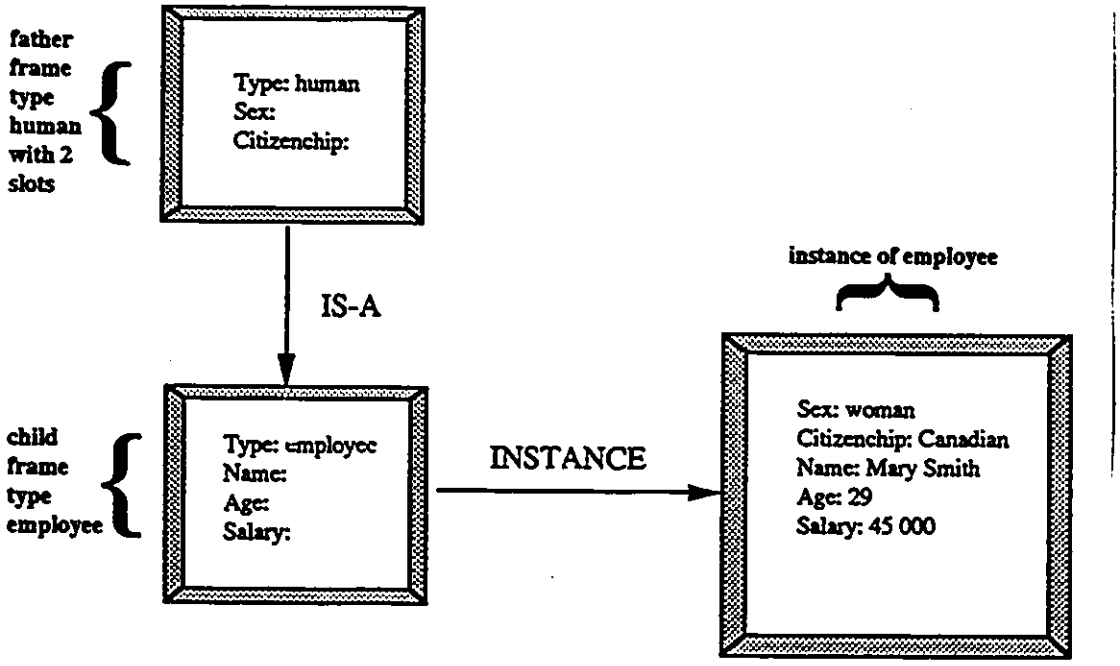

Figure 3.4: Example of frames

One of the main features of frames is that they are organized into hierarchies, each frame inheriting the properties of all the frames situated above in the hierarchy. This makes frame theory very close

to object orientation, as will be seen further below. In frame systems, it is also possible to express relationships between frames. The most common of these is the ISA link, corresponding to the inclusion relation between sets. More complex links can be created using action frames, that have some other frames as parameters to a given function. For example, it is possible to express cause-effect relationships by using action frames and a system of pointers from frame to frame along the list of frames involved in the action.

3.4.1 Advantages and disadvantages of frames

Perhaps the most attractive feature of frames is the similarity existing between frame representation and the way humans use to organize their knowledge about the world. This makes frames very natural to use, as the human mind is well trained to classify objects and actions into hierarchies. Other interesting features are linked to the inheritance property of frames, which enables the existence of default values, and fosters further specialization of existing frames when new concepts have to be represented.

Among the disadvantages of frames, there is the difficulty in knowing how to set up the hierarchy. What classes should be used, what slots should they contain ? In other words, the question is how to represent knowledge (declarative or procedural) cost-effectively. The efficiency of the search and the flexibility of the hierarchy will depend on this. Efficiency in searching can also be damaged by frames recursively calling frame slots [Hu89].

3.4.2 Frame-based languages

It can be said that frame-based languages are actually tools to manipulate semantic nets. One of the main differences between frame-based languages and object-oriented languages (see Chapter 4 for a description of the latter) is that frames can represent abstract data types as well as instances. Indeed, a frame-base language provides usually at least two relations: ISA and "instance". This makes possible to manipulate abstract types at run-time, which is not possible with most object-oriented languages. One example of frame-based language is SRL [WFA84] where there are about a dozen predefined relations, and the user can also define his owns.

A variation of frames is represented by scripts, that are a sequence of actions or events describing a scenario. With scripts, facilities to represent chronology are needed [SA75].

As a conclusion to this chapter, it may be interesting to note that some work is now done to integrate AI applications in multi-user environments. Traditionally, AI programs are strictly single-user, and all their resources are located on the same workstation. The problem, as evoked in [Tic87], is that if one wants AI to get distributed and multi-user, there may be some loss of efficiency, as AI programs typically require large amounts of data and memory. The question is then to access this information quickly enough. In our problem, however, there is no need to access large amounts of data via network connections, as it was said earlier that the application is centralized. In terms of knowledge

representation, we need real-time responses and flexibility, which makes us choose production systems. A more complete justification of this choice will be done in Chapter 5.

Chapter 4

Object-oriented techniques and the X Window system

In this chapter, we briefly review object-oriented methodologies, i.e. general concepts, languages and databases, and present the main ideas behind the X Window system.

4.1 Main object-oriented concepts

The keywords in object-based methodology are objects, messages, inheritance, encapsulation of data, methods, instances, classes, polymorphism, dynamic binding and persistent objects. These are explained below. A reference textbook for these notions is [WEK90].

4.1.1 Objects and messages

Traditionally, the object-oriented approach is presented as opposed to conventional data management. Conventional programs are composed of procedures and data written in different file modules. In object-oriented programming, procedures and data are encapsulated in entities called objects. Procedures inside a given object are called the methods of that object.

One direct consequence of encapsulation is that, from the outside, objects are like black boxes. To access their data and methods, it is necessary to use some strictly defined entry points, that are in fact a special case of methods. Thus, some methods and data can be accessible from the outside, whereas others are not.

Objects communicate between each other via messages, usually taking the form of methods. A class is a specific type of object, complete with its methods and data. An instance is a special case of a class just in the same way as an integer variable in C or Pascal is a special case of an integer type. Defining new classes thus amounts to defining new types in an object-oriented manner.

Figure 4.1 gives an example showing the relationships between data, methods, object, class, and instance. A Book object class is defined, and a particular instance is declared and initialized. In the example, open and read can be called from within the instance.

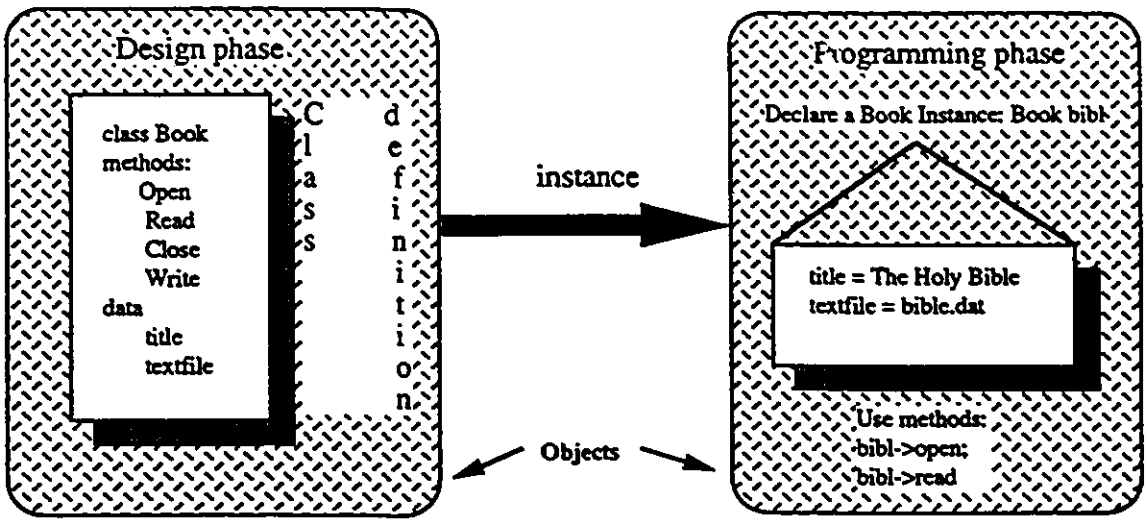

Figure 4.1: Example of objects, data, methods, classes, and instances

4.1.2 Inheritance

A new class can be derived in terms of an already existing one. This new class inherits all the properties of all its ancestor classes, i.e. their data and methods. This process is called the inheritance mechanism. Creating iteratively new classes produces hierarchies of classes which can be viewed as trees. Inheritance is a very powerful feature of object-based methodology as it saves much development time by allowing one to reuse existing code.

Some languages permit to have one class derived from more than one parent class: in this case, one speaks of multiple inheritance. Figure 4.2 presents inheritance and multiple inheritance. In this example, two class hierarchies are defined, one for organisms, and one for animals. Mammals can be considered as both hairy organisms and vertebrate animals, this is an example of multiple inheritance.

4.1.3 Polymorphism and dynamic binding

With the object-based approach, an object can send the same message to objects belonging to different classes, given that the receiving classes have implemented the corresponding method. Polymorphism refers to the fact that a same request can trigger several different answers.

An example of this is a system where we have a list of objects, and all the objects are derived from the same base class, say BaseObj. However, each individual object in the list can be of any class derived from BaseObj. If we assume a list of animals, we could find elements of class Beaver, Racoon and Moose (see Figure 4.3). We now further assume that each of the derived classes, e.g. Beaver and Moose, have implemented a method called Draw to display the information contained in an object. In

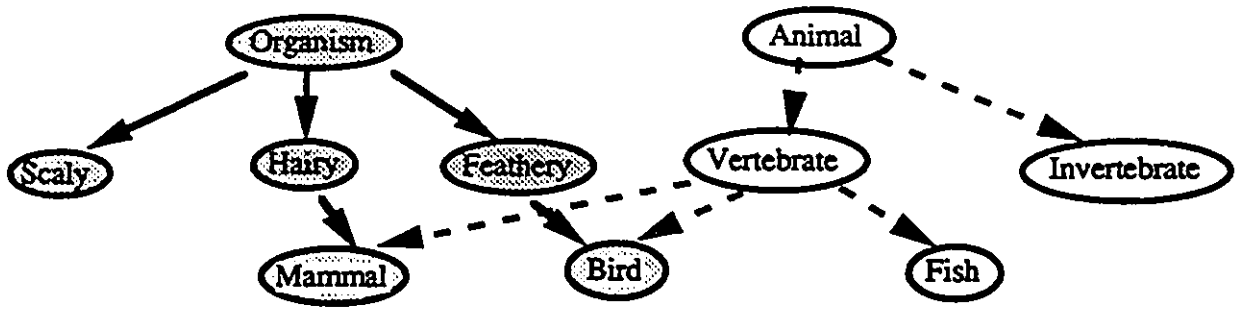

Figure 4.2: Example of inheritance and multiple inheritance

order to draw all the objects contained in the list, the requesting object simply calls the Draw method for each of them, without knowing what their actual class is. The implementation of Draw normally depends on the class of objects which is to be drawn.

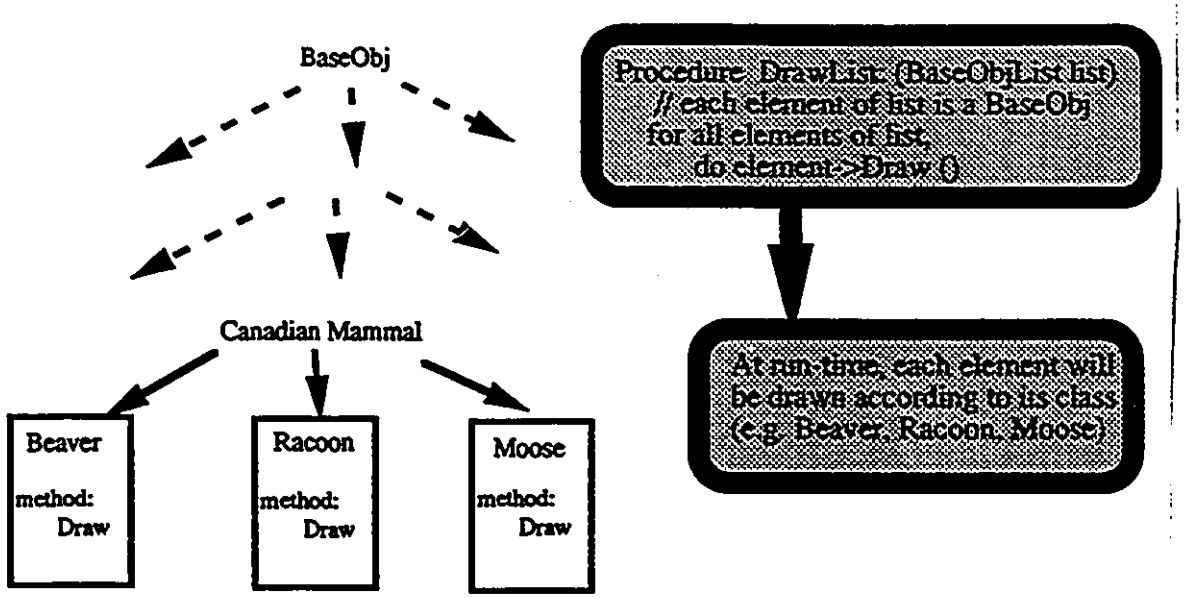

Figure 4.3: Example of polymorphism: the Draw method

Polymorphism requires dynamic binding. In conventional programming, binding, i.e. resolving links between the different components of a program, occurs at compilation time. However, with polymorphism, it is often necessary to locate dynamically methods in a class hierarchy. Binding between a method's symbol and the relevant data then occurs at run-time.

4.1.4 Persistent objects

As an object-oriented program handles objects, it may want to store some of them permanently, i.e. at least until after the execution is over. Such permanent objects are called persistent objects.

As far as the other objects are concerned, it is necessary to free their associated memory space when they are no longer needed. Automatically restoring previously allocated memory is called garbage collecting.

4.2 Object-oriented languages

One of the first object-oriented packages was a language developed in Norway in the late 1960s by Kristen Nygaard and Ole-Johan Dahl. This language, called Simula67, introduced the concepts of classes, coroutines and subclasses [DMN70].

However, the first complete OOL (object-oriented language) was Smalltalk [GR83], developed in the mid-1970s at the Xerox Palo Alto Research Center. Today, Smalltalk remains one of the purest OOLs, as it represents everything in its programming environment as an object. Smalltalk has never become widely used in the industry for several reasons, in particular because it is slow and because it comes along with a complete working environment.

Other examples of OOLs are Eiffel [Int87], and Actor [Whi89]. Both are primarily used for rapid prototyping and investigation. Generally speaking, advantages of OOLs can be described as follows. Compared to procedural languages, they provide reduced program's size through inheritance (although the size of binaries may increase because of the additional class libraries), modularity of programs, easier maintenance, and greater reusability of code.

In addition to pure OOLs, there are also hybrids. By hybrid one means languages that are object-oriented adaptations of procedural languages. These include C++, developed by Bjarne Stroustrup at AT&T [ES90], Stepstone's Objective C, CLOS (Common Lisp Object System) [Moo89], as well as several object-oriented Pascal.

The strength of these hybrid languages is that they are almost completely compliant with their respective procedural counterparts, which allows programmers to reuse all the already developed software at the cost of minor changes. Another of their salient features is their speed: C++ code, for instance, is almost as fast as C code, fast enough anyway for AT&T to have rewritten UNIX's kernel in C++. These features, added to the standard advantages of true OOLs, account for the success of hybrids in industry.

4.3 Object-oriented databases

Several factors motivate the need to apply object-based technology to databases. The first one is an increase of complexity of software modules. The second one is an increasing diversity of data types, including images, voice and video. Relational databases have difficulties coping with expressing the semantics of complex objects with the relational table model, and do not handle large quantities of data very efficiently.

Nevertheless, there are cases where relational databases are more suitable, in particular when only independent records have to be accessed. OODBs (Object-Oriented DataBases) are more suitable

when complex relationships between the objects in the database are involved. Corresponding applications include multi-media databases, artificial intelligence applications, mechanical or electrical CAD, computer-aided software engineering (CASE), or computer-aided publishing (CAP) [Hod89].

OODBs can be fruitfully coupled to OOLs as they supplement the latter in a number of fields, including persistent objects management, version control, concurrrent access to information, and code sharing.

As in the case of languages, there are two types of OODBs: pure, built from scratch, and hybrid that are derived from relational databases. Examples of pure OODBs are Graphael's G-Base, O2 [D+90], Ontologic's Ontos [Ont89], and Servio Logic's GemStone [B+89]. Hybrids include POSTGRES [SRH90], based on Ingres, Oracle, and Hewlett Packard's IRIS [F+87].

4.4 Object-oriented graphical user interfaces and the X Window system

The emergence of GUIs (Graphical User Interfaces) is usually associated with the release of Apple's Macintosh in 1984. In 1985, Apple released MacApp, an object-oriented framework to develop applications on the MacIntosh [Tes85]. Since then, many other systems were developed, including NeXTStep for UNIX-base platforms [Web89], developed by the NeXT Computer company, Microsoft's Windows for DOS, Presentation Manager from Microsoft and IBM for OS/2, and X Window developed at MIT and DEC [SG86].

All these systems share several characteristics: they are window-based, use different kinds of icons and menus (e.g. pop-up, pull-down and cascade menus), implement the point-and-shoot idea, and typically foster the use of a two- or three-button mouse, except MacApp that uses a one-button mouse. In addition to this, they provide a number of graphical objects to ease the man-computer relationships.

4.4.1 The X Window system

The X Window system is supported by the X Consortium, which gathers primarily UNIX-based hardware and software companies. One of the goals of this consortium is to establish X as a standard base for user interfaces on UNIX.

It is necessary to make a difference between the core X library, the many widget sets and toolkits that are based upon this library, and the family of UIMS (User Interface Management System).

The core X library

X is server-based. One server can drive one or several display devices, but there can not be more than one server per display. Servers send or receive requests that correspond to functions of the X library.

As X is also network-based, it is possible to send orders from any machine to any X server in the network. In other words, it is possible to display any X application on any display, provided that display is controlled by an X server. One of the main strengths of X is its transparency across devices, networks, and operating systems: there are now versions running on DOS, AmigaDOS, and VMS [BTW91].

Widget libraries and toolkits

Thus, the X library provides a way of sending graphic requests across a network of machines hosting X servers. However, the level of access is limited to window management. For higher-level programming, it is necessary to resort to a new class of graphical objects. Widgets are such objects.

A widget can be defined as a graphical object whose behaviour can be partially specified by programmers, either as standard application code, or in special configuration files read by the X application at run-time. As an example, there are scrollbar widgets that can be combined to drawing surface widgets and push-button widgets to create a simple user interface to a drawing program.

Widget libraries implement pseudo or true object-oriented principles, as widgets are organized in tree-like hierarchies where each one inherits properties from its ancestors. As many widget libraries are written in C, the inheritance mechanism is only simulated via a system of include files.

In order to be able to manipulate widgets, the core X package, in addition to the X library, comes along with the Xt library that provides widget-level tools. The main difference is that the X library provides only facilities to handle windows, whereas the Xt library can handle higher-level entities. Figure 4.4 indicates the relationships between the different components of the X package.

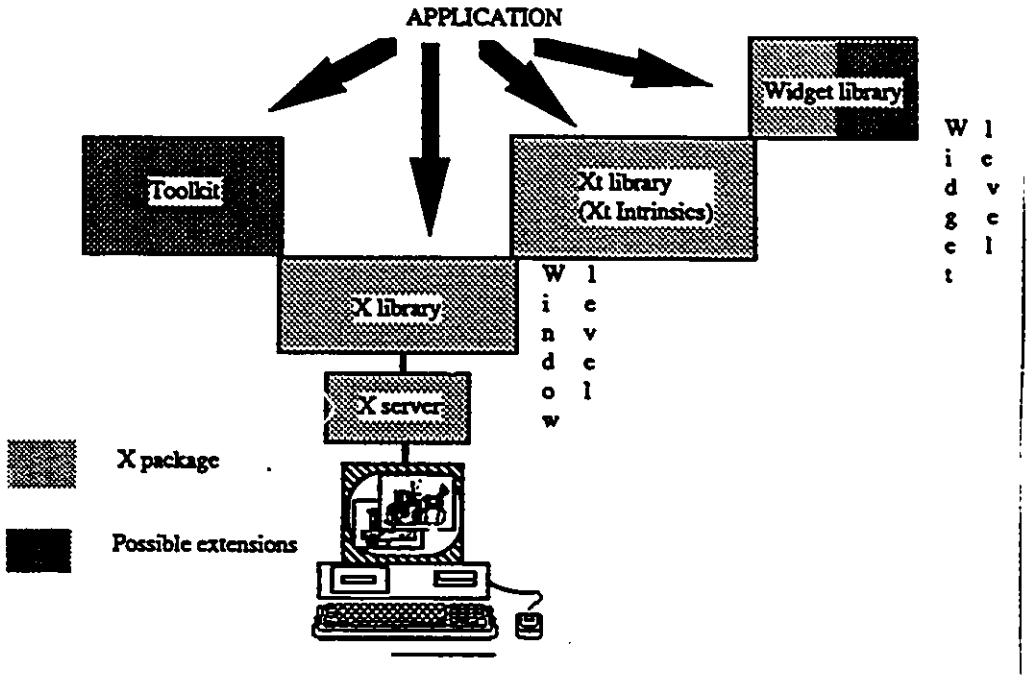

Figure 4.4: Main components of the X package

There are many different widget libraries, among which the best known are OSF's Motif [Fou90] and OpenLook from AT&T and Sun Microsystems. Others include standard CoreComponent widgets from the X Consortium, X Widgets from HP, Xsw from Sony, Athena widgets from MIT, DecWindows widgets from DEC, and Common X Interface from HP and Microsoft.

In addition to these widget sets, there are other high-level X toolkits, like Clue from Texas Instruments and Xray from HP. InterViews, from Stanford University[LVC89], is a true object-oriented

graphical system written in C++. Andrew Toolkit, from Carnegie Mellon University, has capabilities to create multi-media documents [P+88]. It can handle objects made up of text, bitmaps, images, as well as more complex items like spreadsheets or animation editors.

4.4.2 UIMS

One of the problems related to the more popular widget sets such as Motif or OpenLook is the quantity of functions a newcomer has to learn in order to be able to write a few lines of code [Sou90]. This is where UIMSs come into play.

We will consider mainly User Interface Management Systems based on X Window, although there are many others existing for other graphical interface packages. UIMSs help reduce the learning curve otherwise necessary to X newcomers. They provide a number of features to perform very high-level interface programming, and usually sit atop a few widget sets. Some systems provide WYSIWYG (What You See Is What You Get) interfaces, which means that the user defines his interface directly on the screen with continuous feedback, whereas other systems simplify the programming task by providing an easier and shorter syntax. One of the drawbacks of UIMS is that they enforce more or less a special type of interfaces, which results in some lack of flexibility [Sou90].

Among WYSIWYG products, Guide from Sun for Sun's OpenWindow environment, generates C code that can be easily tied up to the application program. The separation between application and interface is a common feature of all UIMSs, whereas toolkits allow more flexibility. As it is emphasized in [RHM+88], this separation is not always desirable nor possible for all applications.

One of the most popular UIMSs is UIMX from Visual Edge Software. In addition to be WYSIWYG, UIMX includes a C interpreter and debugger, as well as capabilities to build custom graphic objects. UIMX is also able to convert existing C programs to GUI programs.

Another product, X-Pression from Unicad, uses an user interface server, called the UI Server, containing all executable code necessary to build user interfaces. Consequently, building an interface with X-Pression simply amounts to initializing configuration files, as no linking to the application is required.

An example of UIMS combining object-oriented methodology and rule-based systems can be found in [Son88]. This system, which [illegible] t X-based, proposes a modular environment to build interfaces completely separated from the application part.

Generally speaking, object-oriented systems make possible a direct, natural correspondence with the world. Indeed, man is used to reason in terms of separate entities (i.e. objects), each having a set of own attributes (i.e. data and methods), and classified in a hierarchical form (e.g. inheritance).

Chapter 5

Display management requirements for a multi-user application

This chapter tackles two main problems. The first is to specify the requirements. Given the assumptions, we want to determine the parameters necessary to manage displays. The second problem is how to organize the corresponding knowledge. A simplified rule-based system will be used to represent the constraints upon inter-user relationships. As it was said earlier, we will focus on the example of a poker game to make things clearer. A list of the type of rules or constraints that may be necessary will be presented further below.

5.1 Assumptions

There are several aspects of multi-user systems which we take for granted. In particular, we do not address communication problems, e.g. protocols for data transmission over a network of interconnected workstations.

5.1.1 A notion of display management

Our first assumption is that, at first sight, display management can be divided into two distinct subtasks: system initialization, and dynamic control during the session. Initialization uses some general knowledge about the session that can be typically expressed in the form of facts. Dynamic behaviour of the interface is determined by the way the interface reacts to user action according to some controlling scheme. As far as the control part is concerned, it is important to note that, as possible user actions consist of requests through pointing devices or keyboards, display management can be limited to determine whether a given request is valid, and to provide the relevant feedback. Control and general knowledge will be treated after reviewing the other assumptions.

5.1.2 Subjects and views

We assume the existence of multi-user objects, which we define as entities consisting of a subject core part linked to a list of graphical views. The subject contains intrinsic information which does not depend on the display. User-specific information is contained in the views. Another way of putting

it is to say that views are the user-specific graphical interface to the associated subject. Each view knows how to display itself on one workstation using the common data stored in the subject. As each view can be accessed by users through mouse action for instance, subjects can be modified through changes to one of the views. The change then is propagated to the other views.

With this subject-views principle, also used in the InterViews package [LVC89], although InterViews does not permit to distribute the views on different displays, one makes an assumption on the architecture of the multi-user system. Indeed, it corresponds to a centralized architecture. However, it would be possible to extend what will be said below to replicated architectures, even though this may not be straightforward.

In the case of poker game, a good example of subject and views is a card that is to be displayed on several displays. The subject contains intrinsic information, such as what type of card it is and its location on a reference screen. Each view can use these coordinates to display the card on its associated screen. Coordinates can possibly be transformed if one wants to have different screen orientations. Each view then needs to create a graphical object, for instance a widget, and to implement input-output facilities for interaction with the player. Figure 5.1 gives an example of a card subject linked to several views, where an ace of clubs is visible on four displays. On two of these, it is displayed back face up, on two others, back face down. Note that views are related to a given area, but the area itself is not in a fixed position.

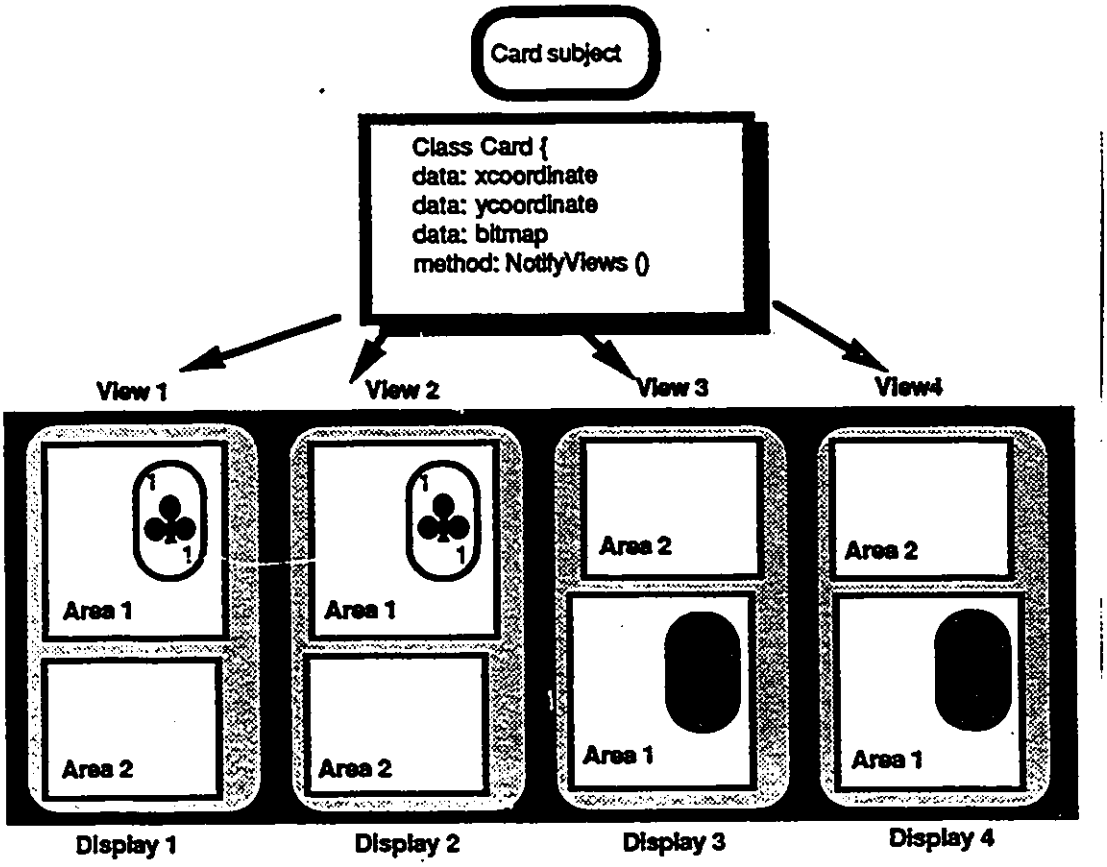

Figure 5.1: A sample Card subject with its associated views

5.1.3 Permission system

Another major assumption concerns the permission system. Each display is to be divided into distinct areas. Each area is represented according to the subject-views principle. That is, there is a subject containing size and coordinates information, and several views distributed over different displays. It is up to the application programmer to know whether areas should be allowed to overlap. Similarly, one has to decide whether or not some areas should be visible only on some users's screens, and whether different users should have different screen orientations, for instance using rotations, scalings or translations as was explained in Chapter 2. A priori, no graphical objects should be displayed between the areas. However, some type of pop-up menus or general information could be inserted there.

The general idea underlying the permission system is that each view of each multi-user object has access to some information indicating what its permissions are for a given area view of the corresponding display, i.e. of the corresponding participant. Each set of permissions contains two types of information, some related to atomic actions, and some to display permissions. These ideas are explained in what follows.

Permissions on atomic actions

When dealing with multi-user objects, there is only a limited set of actions users can perform upon them. These actions, which can be called atomic actions, as they represent the lowest-level, application-independent possible interactions between users and the graphical interface, include object selection, de-selection and motion across the screen. Atomic actions are part of the methods of multi-user objects. Selection can correspond to button pressing or, in the case of cards for instance, can be the first step in moving a card from one location to another. We call this action SelectionRequest. Motion occurs when an object has been selected and is dragged using the mouse. De-selection can correspond to the end of a motion, in which case we define the EndMove action, or can follow a former selection action, without movement. In this case, we call the action StaticRelease.

For each type of action, the exact number and nature of which depends on the application and the objects considered, we can specify whether it is allowed or not for a given user and a given screen area. Determining the set of required atomic actions has to be included into the design phase of the multi-user objects.

In the case of card games, we need at least the three atomic actions SelectionRequest, EndMove and StaticRelease. Note that it is possible to associate different responses with the same atomic action for different classes of multi-user objects. For instance, StaticRelease, when applied to objects of type Card, can mean flip the card. It can mean something else for objects of type Chip, which constitute another type of objects needed to play poker. As we will see later, other atomic actions may be necessary.

Permissions on how to display objects

In addition to this set of permissions on the atomic actions associated with multi-user objects, we add another one indicating the way the object has to be displayed. The three possible ways are:

- NormalDisplay, in which case the object is displayed without modification to its current state. Each object can be in two graphical states. In the case of a card, this corresponds to each face

of a card.

- HideDisplay, meaning that the user is aware that the object exists, but without knowing its exact nature. For instance, for cards, this corresponds to display the back face of the card, without letting the user know whether the card is a king, a jack or whatever.
- NoDisplay: nothing is displayed.

It is to be noted that permissions are set at the object level and not at the area's. In the latter case, all the objects inside an area would have the same permission. However, this approach, although simpler to implement, is not satisfying for our purposes, as it is not flexible enough.

5.1.4 How multi-user objects handle permissions

Multi-user objects are accessed through their views. Each view knows how to react to any user action according to the permission associated to the action taken and to the area it is in.

Actions on a view are requested via mouse clicking. As soon as the corresponding event is detected by the application, the view has to check whether the requestor has permission to select it. If not, nothing should happen. If yes, the view can be modified. If it is still in the same area at the end of the action, one just has to notify all the other views that the position or the state of the subject has changed. For instance, if a player decides to flip one card, this information has to be transmitted to the others. When the view has to be redisplayed, it is necessary to check whether the permission is NormalDisplay, HideDisplay or NoDisplay.

Then, each other view will redisplay the subject according to the relevant permission. If it is NormalDisplay, all the information should be displayed. If it is HideDisplay, only some partial information should be shown, the exact nature of which should be known by the view. If the permission is NoDisplay, nothing should be visible. The same principle holds in the case of an object being moved from one source area to a destination area different from the source.

Thus, views need to know in which area they are, or areas should know what objects are inside them. However, the second alternative puts the emphasis on the areas, which does not fit well with the object-oriented principles that foster autonomy of objects. Moreover, the first alternative requires less processing, as objects are moving, whereas areas are fixed. In the second alternative, it would be necessary to detect and maintain a list of objects entering or leaving each area, whereas, in the first one, the layout of the areas on the screen is static. This is the approach we choose. Each view has access to an object specifying where the different areas are. It is then straightforward to know in what area one view is.

We would like to point out that we have not addressed yet the problem of how permissions are to be set, and where the corresponding knowledge is to be stored. Although this will be solved later in this chapter, let us give a flavour of it now. There are at least two alternatives. The first one is to explicitly set all the permissions at the beginning of the session. As this has to be done for each atomic action of each view, for each area, this can be quite cumbersome. Moreover, such a process is error-prone and extremely difficult to modify. A second alternative consists in coding rules which will be accessed by objects at run-time. Indeed, it seems intuitive that the actual process of setting and checking permissions would benefit from a flexible knowledge representation technique. We will see later in this chapter that rule-based systems provide such a flexibility.

Before turning to the next section, which deals with the requirements for display management, let us summarize what our main assumptions are. We have at our disposal a set of multi-user objects built according to the subject-views principle. Each of these knows how to manage itself for display and permission checking. Permissions are of different types and are action-, layer-, object- and area-specific and are set at the object level.

5.2 General knowledge about a session

A number of parameters are necessary to describe the initial state: who is participating, what objects are involved, what is the screen layout for each participant, what are the initial permissions.

5.2.1 Participants

It is necessary to know how many participants are involved, their names, and the location of their workstations. This information will be used to set up the connections with window servers, and to set constraints and relations on the participants explicitly.

5.2.2 Multi-user objects

We have to specify the type of multi-user objects, where they are located, as well as graphical attributes relative to their size, color, and so on. Objects needed include cards, chips, telepointers, menus and areas. We assume some audio/visual link between participants independent of display.

5.2.3 Areas

We need to know how many areas there are, their size and position, color, and possibly other graphical attributes. In addition, we will see next that we need to define properties on objects, in particular on areas. For instance, in card playing, it will be necessary to define an area representing a table. One of the areas will then be assigned the "table" property.

5.2.4 Menus

Some built-in menus are necessary, in particular to obtain information about the game, to stop it or restart a new session. We do not provide menus outside the scope of the game (e.g. editors or shell commands).

5.3 Using rules to manage the display

5.3.1 Why rules

The problem of knowledge representation is closely related to that of how the knowledge is to be used [Pau86]. In our problem, knowledge has to be easy to modify, as we want to define different applications at a minimum cost. Moreover, it has to be easily retrieved, and quickly processed. Rules seem particularly suited to our purposes. They provide flexibility and speed, as long as their number

is not too large. We will see next that less than ten rules are necessary to model a poker game application. Moreover, it is easy to write in rule form the permission system controlling what actions are allowed and in what context. Among all the knowledge representation techniques, this approach is the most natural one to meet our requirements.

Another possible candidate would be frames, but they are less flexible and more complex to manipulate. They would be preferable to production rules in the case where rules have to be organized in a hierarchical way, but this is not the case in our problem.

5.3.2 Are metarules necessary ?

Different phases in a session

A poker game session can be separated into different chronologically ordered phases. Although it may appear at first sight that such a separation is useless, and that rules involving only atomic actions are sufficient to express everything needed, it presents nevertheless several advantages. Indeed, different sets of rules can be associated with different phases. This results in at least two consequences. Firstly, processing time is reduced, enhancing the system's response time. Secondly, it becomes possible to associate different responses with the same user actions according to the context. For instance, the action of moving a card from one area to another can mean "play a card" in one case and "give a card back" if it is the end of the game. There may be some applications where it would be necessary to have different semantics for the same lower-level actions, even though this does not seem to be the case for poker.

Defining high-level actions

Metarules could also permit to define higher-level actions, such as dealing cards or betting chips, in terms of a sequence of atomic actions. Such high-level actions would be much more application-dependent than atomic ones, and correspond to a different level of abstraction. Indeed, it is possible to consider the interface between objects and users as the first level, followed by the atomic action-level, still largely application-independent, then by the higher-level of metarules and actions such as betting chips or grabbing cards.

Possible phases for poker

Let us consider now what phases could be defined for poker (see [Ank85], chapter 2, for more about poker). The game starts with everybody dropping chips on the table, building up what is called the "blind". Then, each player receives five cards. A second round of betting takes place followed by a dropping and drawing of cards. A third round of betting takes place and the game ends when one player grabs all the money on the table, which happens after one of the players requires to see others's games by betting the same amount of money as the previous player. The deck is shuffled, and the process starts again from the beginning with a new dealer. The corresponding phases can be called Betting1, Dealing, Betting2, Drawing cards, and Betting3. In this sequence of states, the problem is to be able to recognize the beginning and termination of each.

Assuming all the cards are in the table area at the beginning, Betting1 is over when every player has dropped chips on the table. Those who do not want to play have to give their cards back. Detecting

such a condition is possible if we are able to count the number of objects of a given type in a given area. This can be done by defining a new class of atomic events, termed atomic functions, as opposed to atomic actions. Such a function for area objects would return the number of objects inside the area. The difference between action and function is that the former represents a user action on some object, whereas the latter gives some information on the state of the interface, and has to be triggered by the application itself, not by the user.

Dealing is completed when each player has five cards in his hand. This event can be detected with the same type of atomic functions as previously.

Betting2 is similar to Betting1.

Drawing cards can also be ruled using atomic functions that count cards: each player should receive as many cards as he put back on the table. Betting2 ends when everybody puts his cards back on the table, resulting in no cards being left in the own areas of the different players.

However, the question is: is this separation in phases necessary to manage the display in a poker game session ? Our feeling is that it is not, based on the fact that the implementation of the system provides satisfying results, as will be seen in Chapter 6. This does not mean that such an improvement could not be considered as a possible extension, even though response times may not be short enough to provide smooth interactions. But this would not change the principle of display management, just the level of control on the interface. And we find, looking at the results, that this level is high enough when only atomic actions are used.

5.4 Minimum configuration for poker game

Our concern is now to find out what kind of rules we need, knowing that we have a set of atomic actions at our disposal. We start with actually reviewing what objects, rules and actions are necessary to play poker. Then, we try to think of all the rules necessary to control a poker session.

5.4.1 Objects, areas, players, and atomic actions

We have four players, one table, represented by one of the areas, and four hands, one for each player, corresponding to four areas. As the general information about the session contains named references to the objects and participants making up the game, we assume we have access to these names.

One of the players will be the dealer, which can be noted using a predicate-like notation such as Dealer (Carl), meaning that Carl is the dealer. Another predicate can be used to specify what area is the table, e.g. Table (area1). As we will need the concept of an area specific to one player, we define the Ownarea predicate: Ownarea (Carl, area2) then assigns area2 to Carl.

Concerning atomic actions, we assume that cards and chips can be accessed by SelectionRequest, StaticRelease and EndMove actions, which would be noted according to the following Card::EndMove template.

5.4.2 Permission rules

In this section, we review the possible rules, assuming that the same rules apply to the entire game session. In other words, we assume there is only one phase in the session. We consider only what we call permission rules, i.e. rules whose function is simply to return a permission for a given request.

Rules for permissions on atomic actions inside a given area

Rules concerning permissions on atomic actions occurring in the same area are as follows:

- every player is allowed to handle objects in his own area.
- no player can select or flip objects in another player's own area.
- every player can handle objects inside the table area.

Rules for permissions on atomic actions between different areas

Rules concerning permissions on atomic actions between two areas are:

- every player can move objects from his own area to the table area.
- only the dealer can move cards from the table to someone's own area.
- every player can move chips from the table to his own area.
- no player can move objects from another player's own area to his.
- every player can move objects from his own area to another player's.

Rules for display permissions

Rules concerning display permissions are:

- every player has NormalDisplay permission on objects in his own area.
- every player has HideDisplay permission on cards located in other players' own areas.
- every player has NormalDisplay permission on chips located in other players' own areas.
- every player has NormalDisplay permission on objects in the table area.

From the English formulation of the preceding rules, one sees that, in addition to the requirements reviewed above, it is also necessary to be able to express where an action is to take place. For each rule involving atomic actions, we need one, sometimes two, parameters, to specify what the source and possibly destination areas are.

It can be noted that the preceding set of rules is not minimal nor unique.

5.4.3 Turn-taking and starting a new session

A classical problem in multi-user games is to be able to define turn-taking [PHMR91]. As we choose a predicate-oriented approach, it is easy to define predicates such as SuccessorOf (Carl, Bill) to indicate that Bill is Carl's successor. In the case of card games, we define turn-taking only as the succession of different dealers, not as the succession of players inside a single turn. Defining turn-taking at this level would require a few metarules to behave properly.

When one turn is over, it is necessary to modify the identity of the dealer according to the turn-taking procedure (that can be defined with predicates such as SuccessorOf). Thus, at some time during the game, we need to dynamically modify the knowledge base. We define a NewSession atomic action, and assume that changes can be made to the knowledge base each time this action is requested. The exact process of how NewSession is used will be explained further below.

In the case of poker, we state that the dealer is the only one allowed to request NewSession. When the request is accepted, a new dealer has to be entered in the base of facts, the old one is deleted, and the game is restarted with this new knowledge.

5.5 Rule syntax

5.5.1 Reserved keywords

Reserved keywords are the names of the atomic actions: SelectionRequest, StaticRelease, EndMove, and NewSession. Atomic actions have one or two parameters for source and possibly destination areas, except NewSession which has no parameters.

Other reserved keywords are "if" and "and", which are used to build logical expressions. Similarly, Equal and Different predicates are used to test the values of two variables.

The special atomic action NewSession introduced previously is associated with two reserved predicates: AddFact {predicate}, and DeleteFact {predicate}. These predicates add or remove facts from the knowledge base.

The last reserved predicate is

```
DisplayPermission (party, area) : permission if predicate1 and
predicate2 ... and predicaten.
```

The permission parameter can take the three following values: NormalDisplay, HideDisplay, and NoDisplay. DisplayPermission is used to initialize the permissions for each view of each area. These permissions are needed to know how to display an object's view on the screen of a non-requestor player. The exact mechanism involved to display objects will be described in the last section of this chapter.

5.5.2 User-defined predicates and variables

It is possible to define predicates such as Dealer (Carl), Table (area1) and Ownarea (Carl, area2). These predicates will be used to build rules. Also necessary are variables: variable names are capital letters from A to Z.

5.5.3 Rules to specify permissions

From the previous discussion of the requirements, there are two types of rules. The first type returns a permission in response to a request through a mouse action. The second type, presented in the next section, modifies the knowledge base.

Rules are written as

```
Party : Object::Action (Source, [Dest]) : Permission if
condition1 and condition2 and ... and conditionk,
```

where

- Party designates the participant considered, and can be a variable or a constant.
- Object::Action can be for instance Card::SelectionRequest or Chip::EndMove.
- Source, and possibly Dest, are the source and destination areas, and can be variables or constants.
- Permission can have four values: NormalDisplay, HideDisplay, NoDisplay, and PermissionRefused. The first three correspond to the case where the action is permitted, and indicate what the display permission is. The fourth value indicates that the action is not allowed.
- each condition is a predicate.

5.5.4 Rules to change the knowledge base

These actions are tied to menu selections. Indeed, as the system does not automatically detect a change between different game phases, the only way to initiate a new phase (e.g. a new session) is to do it explicitly, in particular through a menu.

There is only one such rule in the implementation of poker, even though the system permits to define similar rules for other actions, such as stopping the game, or canceling the current session without changing the knowledge base. This rule is associated with the NewSession atomic action. As stated before, NewSession is different from the other three atomic actions. Unlike the latter, it is not associated with a given area, nor with a display permission. All that is needed for NewSession is who is allowed to request it. Another distinctive feature is that NewSession requires that a list of facts be added/removed to/from the database. The syntax of rules using NewSession is as follows:

```
Party : NewSession [action1 and action2 and ... actionn if
condition1 and condition2 and conditionp] if
condition1 and condition2 and ... and conditionk,
```

where

- Party designates the participant considered, and can be a variable or a constant.
- each action is DeleteFact {predicate} or AddFact {predicate}.
- each condition is a predicate.

5.5.5 Backus-Naur Form

In an attempt to give a general feeling of what our simple language consists of, we give its definition using a BNF-like notation (see [Els73], Appendix 1).

```
<exp> ::= <predicate>
        | <actionrule>
        | <displaypermission>
        | <modifyrule>

<predicate> ::= <name> ( <paramlist> )
              | <reservedname> ( <param> , <param> )

<reservedname> ::= Different | Equal

<name> ::= a | A |... z | Z | 0 ... 9
         | <name> <name>

<param> ::= A | ... | Z
          | <param> <param>

<paramlist> ::= <param>
              | <paramlist> , <param>

<actionrule> ::= <name> : <object> :: <action> : <permission> if <predicatelist>

<object> ::= Chip | Card

<action> ::= SelectionRequest | StaticRelease | EndMove

<permission> ::= NormalDisplay | HideDisplay |
                 NoDisplay | PermissionRefused

<predicatelist> ::= <predicate>
                  | <predicate> and <predicatelist>

<displaypermission> ::= DisplayPermission ( <name> , <name> ) if <predicatelist>

<modifyrule> ::= <name> : <modifyaction> [ <actionlist> if <predicatelist>]
                          if <predicatelist>

<modifyaction> ::= NewSession

<actionlist> ::= <action>
               | <action> and <actionlist>

<action> ::= DeleteFact {<predicate>}
           | AddFact {<predicate>}
```

Appendix A gives the set of facts and rules, written according to the preceding grammar, corresponding to the requirements seen earlier.

5.6 Principle of access to the permissions

In this section, only permission rules are considered.

5.6.1 Accessing rules from within the objects

Rules are written in a file, then parsed and called at run-time by each object each time a user tries to perform some atomic action upon it. This means that each object needs to know the address of the rule interpreter, or inference engine, to determine whether an action is permitted or not.

Rules involve a number of parameters that objects have to send to the rule server. These parameters are the name of the player who is requesting the permission, the source and destination areas, the type of action performed, and the type of object acted upon. The return value, which will determine the object's response to the user request, is restricted to one of the four values indicated above.

5.6.2 Checking rules

Once a request is received from one of the objects, parameters are extracted and the first rule corresponding to the action requested is checked. If the conditions are verified, then the answer, i.e. NormalDisplay, HideDisplay, NoDisplay or PermissionRefused, is returned to the object. As the conclusion of each permission rule consists only of the answer to the request, there is no need to update the rule base. Thus, the inference mechanism is quite simple. Although this may not appear clearly when looking at the rule structure, the system actually implements forward chaining. Indeed, the reasoning mechanism of a typical rule can be viewed as follows: if the action requested is X and if this predicate is true then the permission is Y.

5.6.3 Redisplaying the objects

If the permission is different from PermissionRefused, it is necessary to redisplay the object on all the displays. For the requestor's display, the permission is the return value from the rule interpreter (e.g. Player1 in Figure 5.2). What about the other views of the object (e.g. views of Player2, Player3 and Player4 in Figure 5.2) ? The principle is as follows. At initialization time, each view of each area receives a permission indicating the way any object located inside the area should be displayed (using DisplayPermission (...) predicates).

In other words, a view can be displayed according to two different permissions. If it has been modified by some user, then it will be redisplayed according to the permission corresponding to some rule involving the relevant atomic action. If the view has not been modified by a user directly, then it uses the default permission of the view of the area it is in.

Thus, as far as permissions are concerned, the requestor of an action is somehow privileged, as he bypasses the default permissions set on areas, using instead permissions relevant to the area, the action taken, and the object manipulated.

5.6.4 Providing explanation

Several authors emphasize the need for explanation in expert systems, pointing out that users feel more confident when they know exactly what happened and why [Buc86]. It seems natural to apply

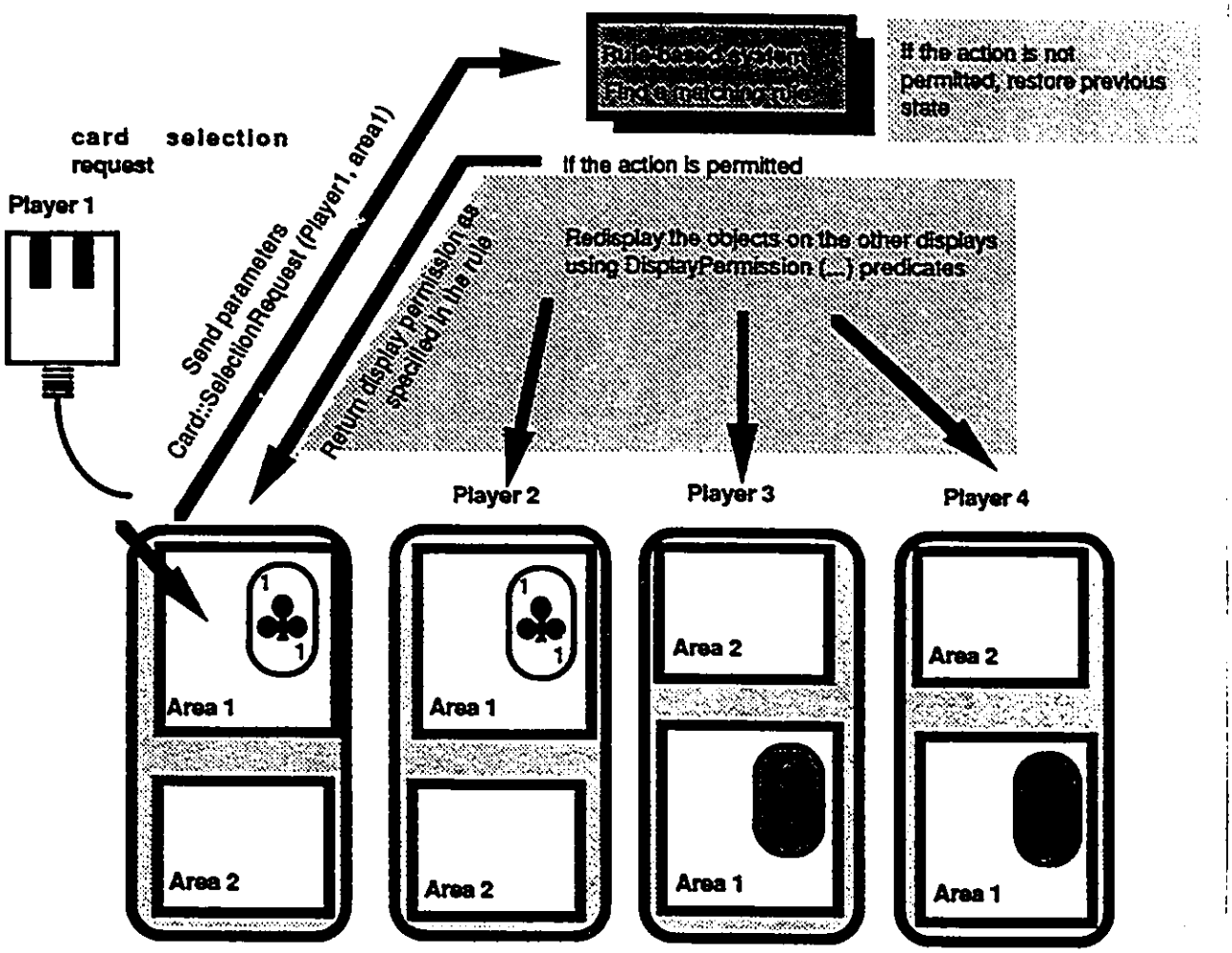

Figure 5.2: Redisplaying objects after a user request

the same principle to a graphical user interface. In our system, we can not afford lengthy explanations about why a particular action was not allowed. As a compromise, the system issues compact and non-distracting explanations. Each time one user attempts to perform an unauthorized action, a pop-up message appears on his screen (and on his screen only) with a standard text such as "Sorry ! This action is not allowed".

The motivation is that it allows users to know why their attempt was not successful. If no messages are issued, users may assume that they did not press the button correctly, or that the system is not working properly.

The salient points that we retain from this chapter, concerning display management in a game context, are as follows:

- display management must permit to define what components will be involved in a particular session: what type of objects, how many, where, what graphical attributes; how many players, their name and location. Each object can be referred to by its name.
- For each player, each type of object, each area, and each possible action, it is necessary to define what can be done or not done.
- For each player, each area, each type of object, one must define if the object should be seen, and how.
- The knowledge corresponding to the preceding points can change dynamically: we need to be able to define what facts have to be added or deleted.
- As we are in a game, an order has to be established between the users.

Chapter 6 presents how these points were implemented, and discusses the results.

Chapter 6

Multicard: a working system

6.1 Presentation of the software

6.1.1 Architecture

The centralized architecture was realized using X Window Release 11.3 as a basis. The application exchanges information with remote or local X servers using the facilities of the Xt library to create connections through a network (see Figure 6.1). The network is an Ethernet running TCP/IP [tcp] [ip85]. For experiments, PCs 386 running SCO UNIX System V/386, Motif and X11R3 were used. However, the program, called Multicard, is written in such a way that it is possible to include in a session any machine hosting an X server, whatever the display resolution may be. For instance, machines such as a Sun 3/150 and Digital DECstation 3100 were also occasionally used.

6.1.2 Main objects

In this section, we describe some of the main classes of C++ objects that were created in order to implement the system. A general view of the class hierarchy can be seen in Figure 6.2, and a description of all the classes is found in Appendix B.

GraphicObj, GameSubject, and associated classes

The GraphicObj class implements the basic functionalities for views of multi-user objects. The idea is that each instance of the class has a widget associated with it, and waits, through the use of X library event handlers, for events corresponding to each of the possible atomic actions. Thus, the GraphicObj object uses the widget as an interface to the user. When one of the events is detected, the view (i.e. GraphicObj instance) asks for the corresponding permission in the way described in Chapter 5.

When one of the views detects an atomic action, and that this action is allowed by the rule system, the view has to notify the subject (see Chapter 5 for an explanation of the subject-views idea) that something occurred. The GameSubject class implements the concept of subject for view objects of class GraphicObj. Thus, each GraphicObj bears a reference to a GameSubject instance, and each GameSubject has a list of associated GraphicObj views.

It is to be noted that GraphicObj implements only event detection and handling, permission request, and redisplay request for all the views. It does not specify how the view is to react to a given

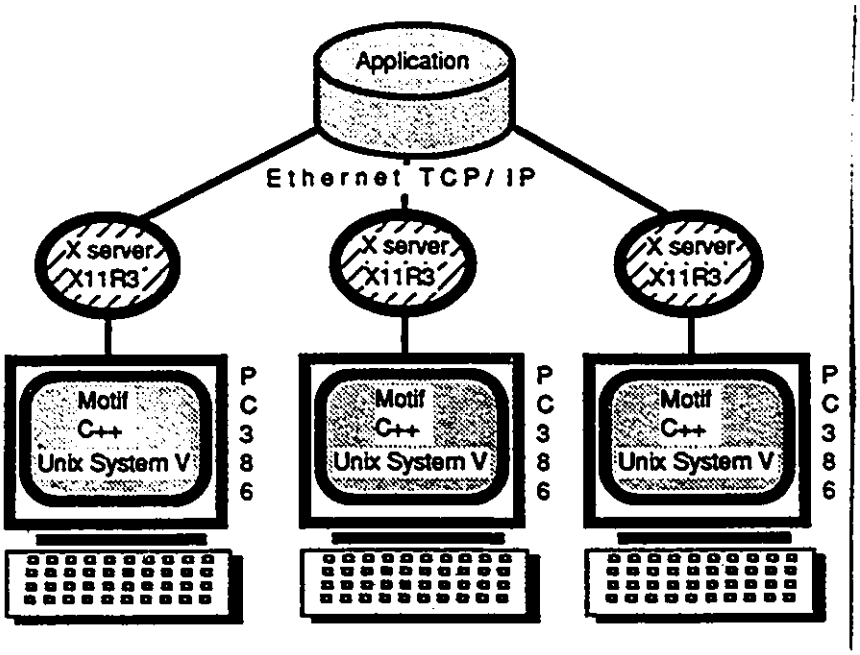

Figure 6.1: Centralized architecture using an Ethernet LAN and X Window

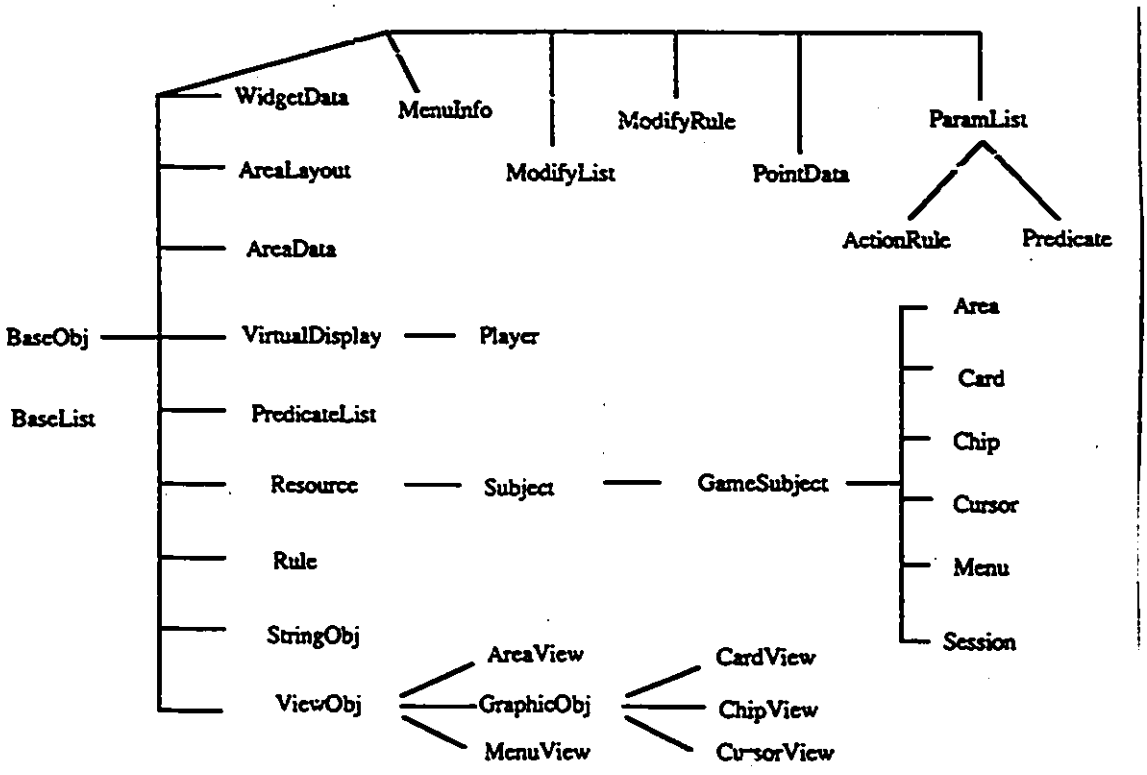

Figure 6.2: A general view of the C++ hierarchy of classes used in the system

atomic action. This is done in the classes derived from GraphicObj. Thus, adding a new type of multi-user object is straightforward, as most of the processing is already implemented in the ancestor GraphicObj class. All what has to be done is to specify what type of widgets it corresponds to, and how it should react to atomic actions. Examples of classes deriving from GraphicObj are the CardView class, that is associated with Card objects (deriving from GameSubject), and the ChipView class, associated with Chip.

The Area class is another class derived from GameSubject, and is associated with AreaView views. However, due to the static nature of areas in our implementation, AreaView derives from ViewObj, and not from GraphicObj. ViewObj is the father of GraphicObj, and implements the subject-views principle without atomic actions. If it was desirable to permit AreaView objects to be manipulated like GraphicObj objects, it would be easy to make AreaView derive from GraphicObj instead of ViewObj.

Player

The Player class actually corresponds to a connection opened on a specific X server, plus a number of parameters. In particular, each Player has an object, called AreaLayout, that permits to know in what area view (i.e. object of class AreaView) a given (x,y) point finds itself. Thus, when a view (i.e. a derived GraphicObj object) wants to know in what area it is, it has to ask the corresponding Player that will consult its AreaLayout object.

Session

The Session object has several duties, among which: read configuration files in order to initialize Player, Card, Chip, and Area objects, and build up the base of facts and rules. The Session object also contains the rule interpreter module.

BaseObj and BaseList

All the classes in the system derive from the base class BaseObj. BaseObj, associated with BaseList, implements basic list processing facilities. Indeed, a large part of the management of rules, facts, and multi-user objects is based on lists.

6.1.3 Configuration files

There are four configuration files for the system (see Figure 6.3). The first one, called Config, contains the following information: number of players, their name, their location in the network, number of areas, and a list of Card and Chip objects. If other multi-user objects were used, they would be specified here too. For each Card object, one specifies what bitmap is to be used (i.e. what card it is). Card objects assume by default that they all have the same bitmap for their back side.

The second file, called Card, records graphical information concerning all the graphical objects used in the session. This file is a standard X Window configuration file, and permits to specify various attributes such as initial placement, size, and colour. In order to ease the configuration process, it is assumed that objects of class Class will be given the names class1, class2, and so on. Thus, if five areas are declared in Config, the information concerning the objects area1 to area5 will be looked at in Card by the application. The process of writing the information regarding 52 cards, 30 chips and 5

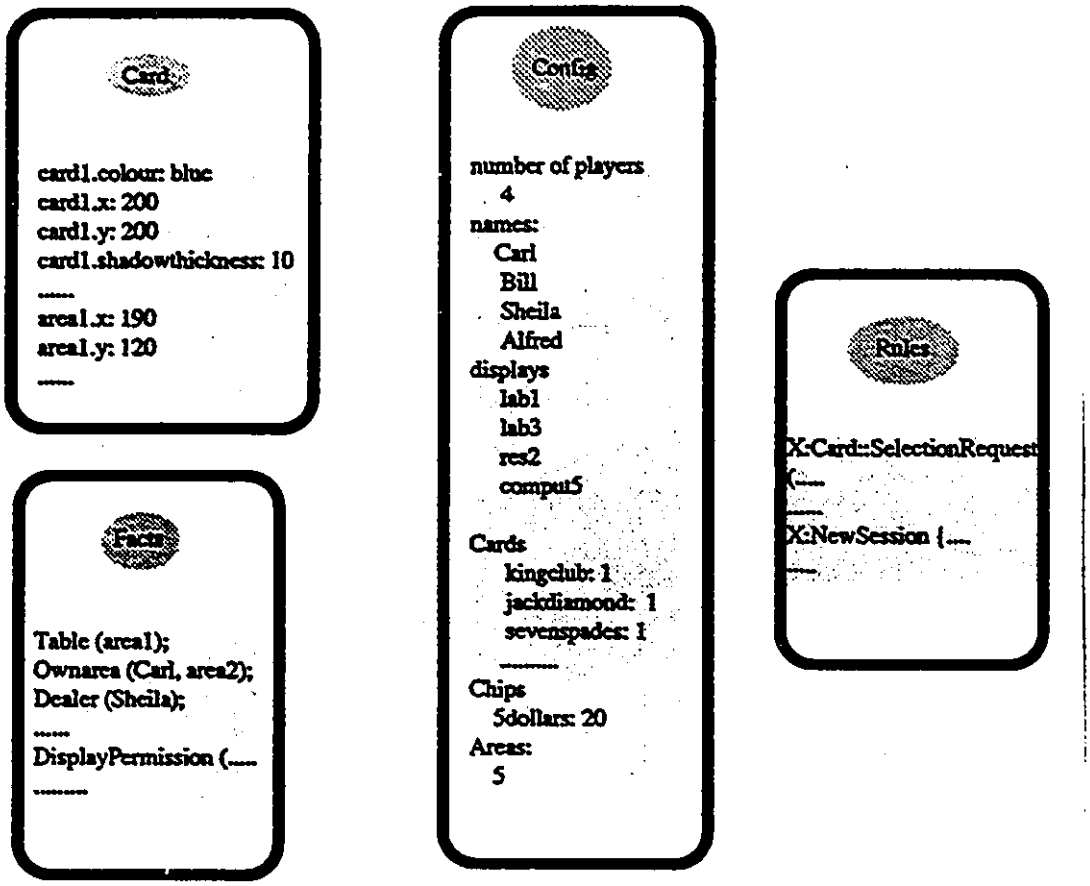

Figure 6.3: Information contained in the different configuration files

areas, for instance, can be done only once. Afterwards, it is possible to use a lesser number of objects by modifying Config only, and with minor changes to Card.

The last two files deal with the rule system. The first one, called Facts, contains information about the session in the form of user-defined predicates and display permissions (i.e. of type DisplayPermission). Constant names used in the predicates can be any of the names declared, explicitly or implicitly (e.g. by declaring 5 areas), in Config.

The last file, called Rules, contains the rules used by the Session object to determine whether a given action is allowed or not. It contains also the rules modifying the knowledge base.

Figure 6.4 summarizes the relationships between the different configuration files.

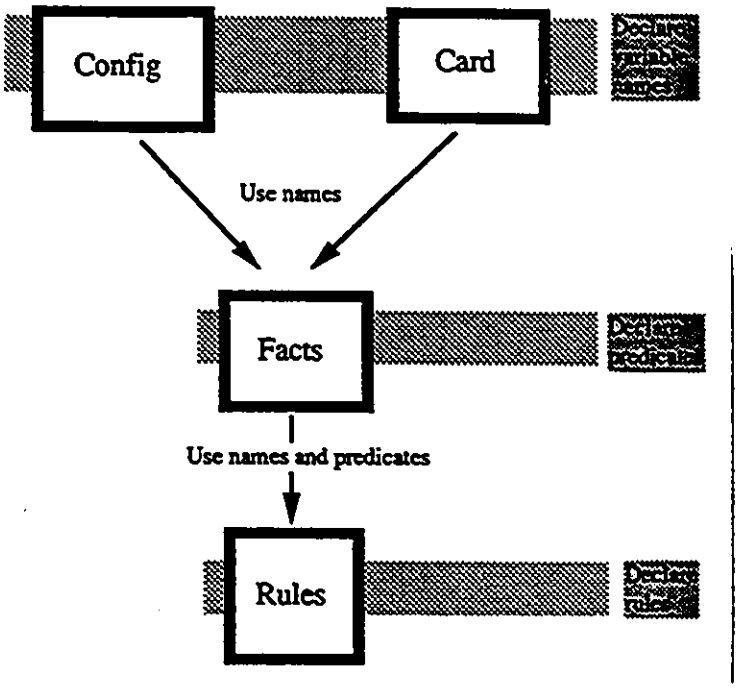

Figure 6.4: A diagram representing the relationships between the different configuration files

6.1.4 Options concerning the production system

Constraint set on the syntax of rules

One way of improving efficiency of logic programs is by restricting the power of the language [Hoa87]. In our case we assume that each predicate in the condition list of a rule can not contain more than one variable symbol that does not appear in the conclusion part of the rule. For instance, if a user-defined predicate ToTheRightOf (X, Y) appears in the condition list, either X or Y has to appear in the conclusion part (i.e. the Party: Object::Action (source, dest) : Permission part). The reason is that this permits to cut down drastically the complexity of the search procedure. Another trick is used to reduce complexity: when rules are parsed, Different predicates are appended at the end of the condition list. Thus, when the rule is processed, it is never necessary to resolve a Different predicate

containing variables, as the inference procedure as presented above ensures that these predicates contain only constant parameters when they come to be considered.

About not using not

In the system, the NOT symbol is not available as a construct to build rules. If it was allowed, it would require a special treatment, which would make processing more complex. We want to keep the system as homogeneous as possible for efficiency reasons. One has to keep in mind that the production system has to be real-time.

Why not Prolog

We defined a simple logical language in C++, instead of resorting to Prolog, that would have provided the same capabilities, and much more. However, Prolog would have been too slow for our purposes, and it forces a backward-chaining approach to reasoning. Backward reasoning involves backtracking, which practically means a lot more computations than what is required in a forward-chaining system such as the one we chose. It can be noted that those AI systems that seek speed and real-time responses are usually forward-chained implementations in some fast language such as C [Bar88].

6.1.5 Handling facts and rules

Parsing

As our grammar is extremely simple, we preferred to ensure a maximum of homogeneity by writing a parser in C++, instead of using some table-driven parser that takes a grammar as input and produces a parser for the corresponding language. An UNIX-based example of such a tool is Yacc [Joh75] that produces a C output.

Different classes were created to represent facts and rules in a convenient way (refer to Figure 6.2), in particular Rule and Fact. As an example, a Rule object is composed of an ActionRule object and of a PredicateList object. Similarly, ActionRule contains a ParamList object to represent the different parameters, and each parameter is represented as a character string. PredicateList is a list of Predicate objects, each one containing a ParamList.

Pattern-matching and searching for permission request rules

In this section, only the rules corresponding to permission requests are considered. The case of rules modifying the knowledge base will be presented later. When a request is made to the rule interpreter method of the Session object by a GraphicObj view, the following steps take place.

1. Get the parameters from the view: player's name, type of object, source and possibly destination areas, and type of atomic action.

2. Take the list of rules contained in Session. If there is no element left, return the default NormalDisplay permission. Else, take next rule. If its action and object fields match the parameters sent by the view, then go to step 3, else redo step 2.

3. Substitute the parameters received from the view in the rule.

4. Take the list of facts contained in Session. Consider the first predicate in the list of conditions of the rule. If this predicate contains no variables, then find whether it belongs to the list of facts or not. If not, this means that the rule can not be fired, so go to step 2. Else (the fact is true), redo step 4 with the list of remaining predicates (i.e. the tail of the list).

   Else (the predicate contains unresolved variables), find the list of possible substitutions for the unresolved variable. Note that there is exactly one unresolved variable, according to what was explained in the previous section. If the list is empty, the rule can not be fired, so go to step 2.

5. For each value in the list of possible substitutions found in step 4, replace it in the rule, and repeat step 4 with the list of remaining predicates (i.e. the tail of the list).

6. When all predicates have been successfully tested, return the corresponding permission to the requesting view object.

Note that if no matching rule can be found, the system returns a default permission, equal to NormalDisplay. This value corresponds to the behaviour of the system in the case there are no rules.

Pattern-matching and searching for rules modifying the knowledge base

The general principle is the same as the one used for rules concerning permission requests. The main difference is that facts have to be added or removed from the database of facts. Actually, as the inference mechanism simply finds the first matching set of parameters for a given list of predicates, only the first substitution list is considered, even if there are several possible solutions. This is sufficient for the type of rules we need, but should be improved if more complex behaviour was desirable.

6.1.6 Interface options

In this section, several options concerning the implementation of the multi-user interface are presented.

Telepointers

Each user can see what his partners are doing through telepointers reproducing their mouse movements. These telepointers are always visible, and contain the name (as specified in the Config configuration file) of the corresponding player. A special class of objects, called Cursor, implements this notion. Cursor is a child of GameSubject, like Card or Chip, and is associated with CursorView views, that derive from GraphicObj. The implementation of Cursor is however different from Card, for instance, as its views have to be displayed on N-1 displays if there are N players, and CursorView does not need to ask for permissions.

As these CursorView telepointers are controlled in software through the windowing system, unlike the mouse cursor that is hardware-controlled, it is important to keep their size small enough so that their movement remain smooth. We limit them to the minimum rectangle required to contain a string representing the name of the corresponding user. There is no problem concerning the geometrical integrity of displays, as the same interface layout is used for all users.

Areas

Areas have a fixed size and location, which we justify as follows. In card games, where users typically both interact with others and have to look after their own game at the same time, it seems necessary to ensure that every participant can have a general view of the game at any moment. Thus, it should not be permitted to let a resized or moved area hide another one. Keeping areas at a fixed size and location is the simplest way to avoid this kind of problems.

Single-user applications

Due to the implementation, it is possible to include small programs such as clocks or parallel editing sessions, but these will remain entirely private, i.e. visible by one user only. This approach is similar to the concept of "rooms" which are separate workspaces containing all the windows relevant to a given application [HC86]. In our case, rooms are implemented with toplevel-type X widgets. Keeping private applications under control is difficult, as the risk is to loose contact with the activity of the other players. However, these programs, like single-user programs visible for everybody using a virtual terminal approach, are irrelevant in our problem: we deal with true multi-user systems.

One-button mouse

All interactions take place via mouse action. Moreover, only one button of the mouse is used. These choices reflect a concern for maximum simplicity of use. Examples such as the interface of Apple's MacIntosh proved that it is possible to perform complex operations with one-button mouses, provided suitable constructs such as menus exist. In the current implementation, no keyboard shortcuts are provided, because none is necessary. Our idea of a good interface is that it is both extremely easy to understand and efficient, so that no shortcuts are required. This seems to be possible in the case of card games applications, where the complexity of the operations performed by the users remains low enough.

Floor control

In an attempt to provide a system as close as possible to real-world card games, there is absolutely no floor control on the multi-user objects. If two users try to move the same object at the same time, they will be able to do it, with the same consequences as if they were trying to grab the same real-world card simultaneously. In other words, it is assumed that participants possess elementary notions of politeness. There has been no problem related to this matter during the tests. It may be interesting to note that the use of rules preventing users from accessing some objects is equivalent to the implementation of a changeable floor control. The only case where there is truly no floor control is that of the public workspaces.

Adding or removing participants

If one wishes to modify the list of participants in a session, it is necessary to stop the game, change the data (e.g. in Config and Facts files), and rerun the application. There is no special interface to do this in a more convenient way. This, however, could be done without stopping the application, due to the centralized architecture (see Chapter 2). Such an improvement was not considered as necessary

in a card game context, where the rules of the game set constraints on arrivals and departures. If someone leaves or enters, a new game is started, which corresponds to changing the configuration.

6.2 Tests with poker game

We now provide details on how the system actually works.

6.2.1 General user feedback

For the tests, four workstations were used, with the rules corresponding to poker. The system was tested by ten users, nine of whom were very familiar with computers, and one who had no computer experience. For each of these, it can be said that one of the goals has been achieved: the system is easy to use and self-understandable.

The response time of the system, i.e. the time delay between a mouse request and the actual change on the requestor's display, is variable. For move actions, in the case of four workstations, it is never significant. However, flip actions can require up to 1 second to be performed. The fact that it takes longer to flip a card than to move it is explained by the greater number of rules (one to select, and one to release) to be scanned in the case of card flipping. It is to be noted that the response time depends largely on the overhead created by the rule system. When the system is run without rules, the response time for flipping is always smaller than .5 second, which yields a .5 second maximum overhead for 6 rules and 10 facts. A former set of runs was done with 24 rules, leading to an extra overhead of 1 second maximum. The figures given assume that no heavy duty processes, such as compilers, are running on one of the workstations simultaneously. For the number of rules involved, the system can be said to be real-time, which is a requirement for the type of application considered. It is interesting to note that only six rules are required to model a poker game. This is due to the fact that many rules are implicit, taking advantage of the default NormalDisplay permission. Thus, it is only necessary to specify what is not allowed, and not what is allowed.

Another parameter is the notification time, i.e. the delay between different workstations when the same object has to be redrawn on each of these. This delay depends largely on the underlying Ethernet network used, and on the load of the different workstations. For four workstations, it ranges from not significant to almost 1 second. There is no significant variation when the stations are distributed on different floors. In all cases, this delay is short enough to permit smooth interactions without damages to the cohesiveness of the group.

One of the problems raised was that the redraw times for the graphics, when objects are moved, were a little bit too slow. In particular, telepointers were not at all synchronized with their associated cursors. This, however, is due only to the poor quality of the graphic boards used on the PCs 386. Indeed, excellent results were achieved on the Digital DECstation 3100.

Another remark was that it would be much more convenient to have the dealing being done automatically, instead of doing it by hand. This could be done by adding a new option in the main menu.

6.2.2 Screen space

As can be seen on the figures inserted in this chapter (for instance Figure 6.5), each screen contains five areas plus the general menu. On small displays (about 14 in), as each area has to receive about five cards, or even thirty-two in the case of the table, this basic layout fills up most of the screen space. For other applications, in particular if other card games were to be implemented with more than four players, screen space would become a real problem. Solutions are discussed in Chapter 7.

6.2.3 Telepointers

They appear to be really useful, and, for games at least, we believe they should be displayed permanently. In our implementation, telepointers are passive, in that they simply follow one player's mouse movements. It would be possible to let them convey more information by modifying their colour, for instance, according to whether the corresponding user is active (i.e. is manipulating objects) or not (i.e. idle or moving his mouse around without grabbing objects).

6.2.4 Information to the users

The system displays two kinds of information: warnings corresponding to non-permitted actions, and general information about the session.

In the first case, each time an user tries to perform an unauthorized action, a pop-up message is issued warning that a prohibited action was attempted (see Figure 6.5). The text for this message is always the same, and can be set through the Card configuration file. We do not believe that it would be very helpful to explain why an action failed by listing the sequence of predicates and rules that led to this conclusion. Indeed, such a level of information is not needed for card players, and would distract the players from the game.

General information about the session is accessible via a main menu visible to every user. We chose not to implement this menu as a pop-up menu for several reasons. Popup menus require some adaptation time for users to learn how to use. Moreover, it is desirable that all users know if someone is using the menu or not, but everybody may not want to see a menu errupt into his screen at any moment. Indeed, experiments showed the importance of smooth transitions in multi-user interfaces [LMB91].

As every participant does not want information at the same time, the corresponding menus are visible only for the requestor, even though the main menu itself is visible to every participant (see Figure 6.6).

6.2.5 Starting a new session

In the main menu described previously, there is an option permitting to start a new session, provided the requestor has the permission to do so (this is determined by the rule for the NewSession atomic action). When this action is requested, a confirmation message is issued (see Figure 6.7). If the requestor confirms his choice, a warning message is displayed on all the workstations, except the requestor's, specifying what is going to happen (see Figure 6.8). Note that Figure 6.7 and Figure 6.8 represent two screens of the same session, illustrating the idea of user-specific views for the objects contained in the top and bottom areas.

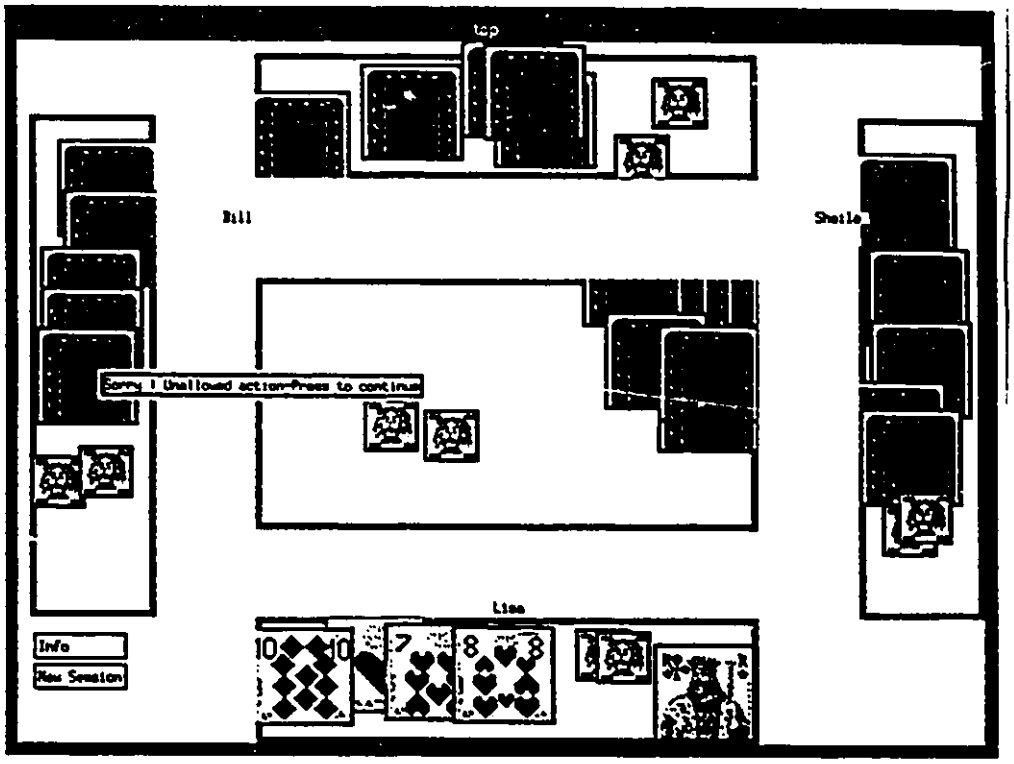

Figure 6.5: A sample session with a pop-up warning message

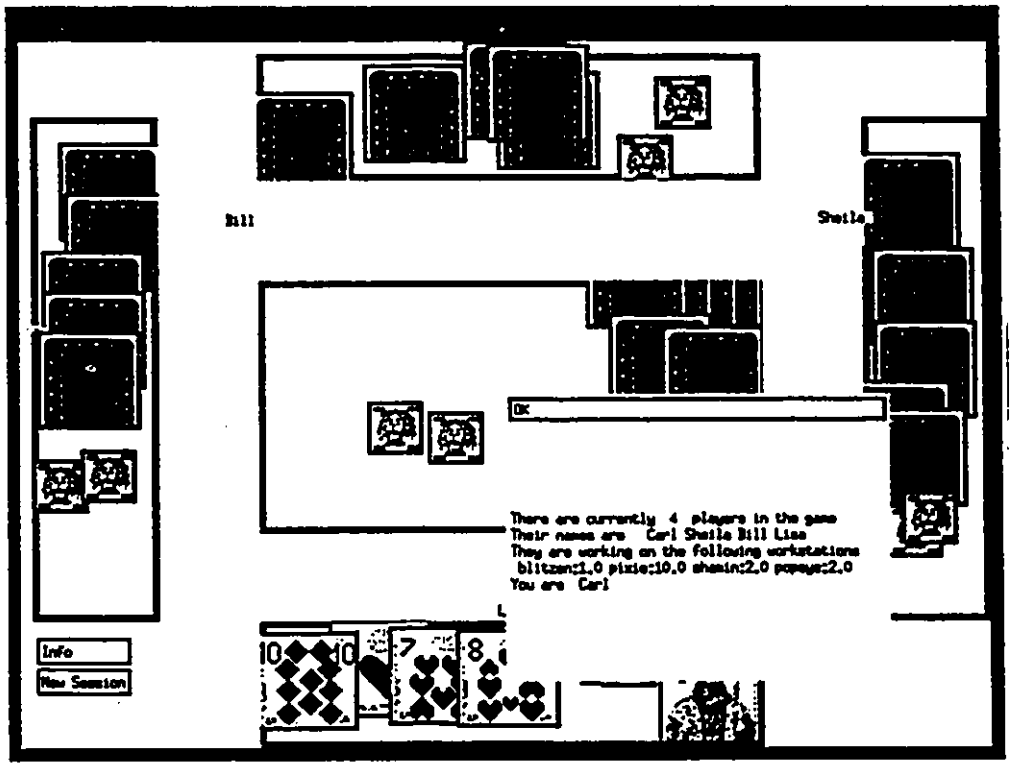

Figure 6.6: A sample session showing a private information menu selected from a menu visible to every player

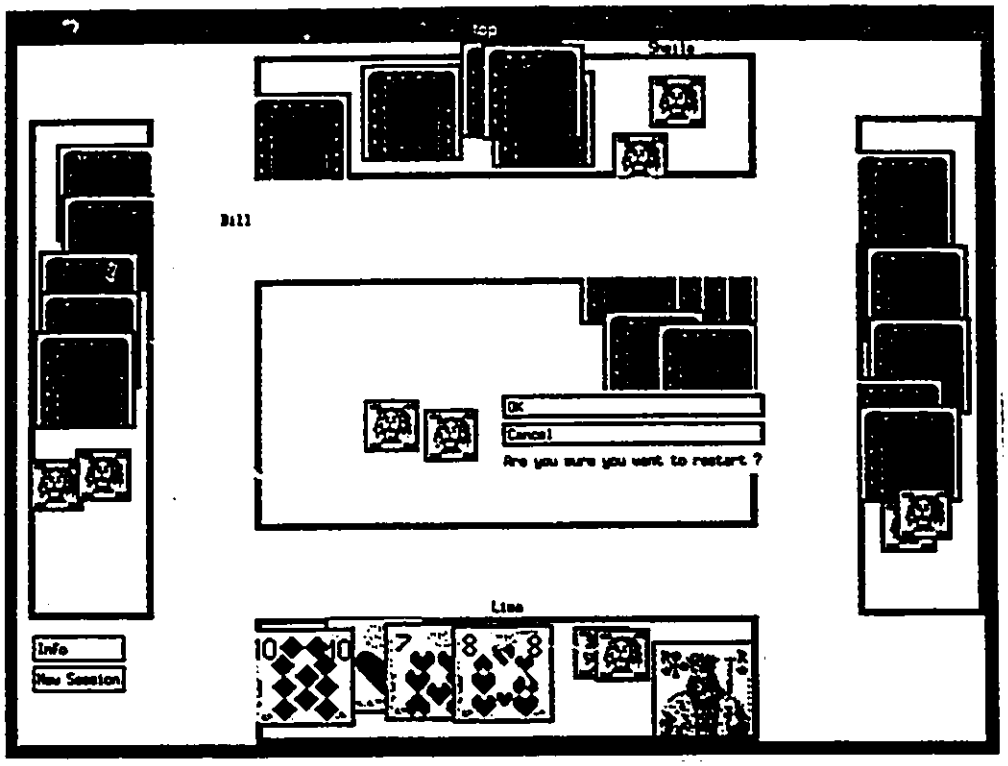

Figure 6.7: Confirmation message issued to the requestor of the NewSession option

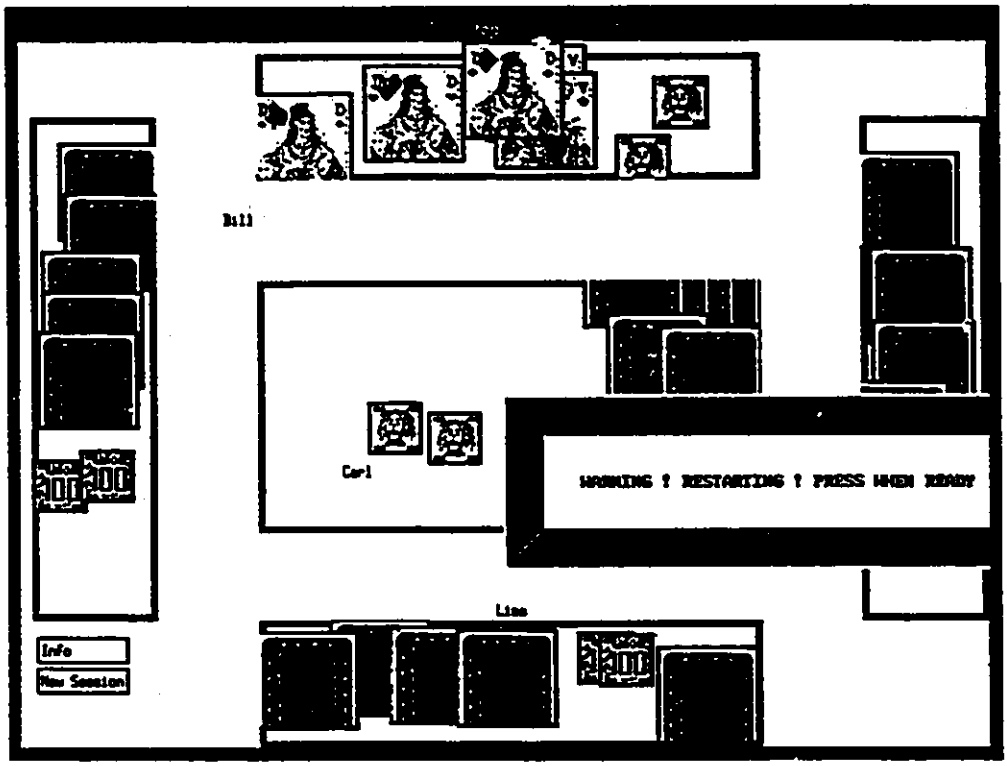

Figure 6.8: A view of the warning message issued to all workstations - but that belonging to the requestor - when the NewSession option is activated

6.2.6 Information between users

As some of the testing was done with the four players located in the same room, whereas some other was done with different rooms, it is possible to decide whether users feel the need for exchanging spoken information. Theoretically, they should not, if every one knows the rule of the game. However, they obviously need to speak. This makes the introduction of some inter-users communication facility necessary, such as a voice transmission capacity or a system of mailing.

6.3 Defining other applications

In this section, we review the possible problems that one could meet when trying to define other applications according to the display management principles presented above.

6.3.1 Expressiveness of the system

The first question to answer is: what type of relationships can be modelled with the existing rule system ? And what can not be done ? At first sight, as we use first-order logic, it is possible to model a great variety of problems. The limitations will be similar to those of logic (see Chapter 3).

However, there is a number of things that can not be done without modifying the system. In particular, the lack of metarules makes the modelling of some problems very awkward.

From that point of view, the case of bridge is very interesting. We take the opportunity to present an actual example of modifying the system, assuming that it is initially configured for poker.

The first difference deals with the number of cards used. Our poker game uses 32 cards (some poker games also use 52 cards [Ank85]), whereas bridge needs 52. The list of cards has just to be modified in Config. The Card file already contains graphical defaults for 52 cards, so it is not necessary to modify it.

Then, the Chips field in Config has to be set to empty, as in bridge there is no concept of betting.

We now come to the critical part. Let us first recall the general principles of bridge playing. Each game can be divided into two distinct phases: bidding, and actual card playing. During the bidding phase, each team of two players tries to agree upon a contract they will have to satisfy in the playing phase. At the end of the bidding phase, one of the teams will try to make their contract, and the other one will try to prevent them from succeeding. In any case, one of the players of the first team, the declarer's partner, displays his cards to everybody. These cards will be played by the declarer, and can not be used by the other team.

Thus, bridge is an excellent example of a game that would be particularly suited to a decomposition into different phases, with different sets of rules. Solving this problem in the current system is awkward. Indeed, even if a new menu option is defined to be called at the end of the bidding phase, changing facts would not be enough: one needs also to modify the rules themselves. To be more specific, the area owned by the declarer's partner should now be made visible to everybody, and only the declarer should have access to it. However, it must be noted that using metarules does not completely solve the problem: how does the system know who is going to be the declarer's partner ? This is still another issue that has to be tackled. A simple solution is to enter the name of the declarer's partner in the system once the button indicating the end of the bidding phase has been pressed.

The example of bridge highlights the main limitation of the system: it is not suited for problems that can not be controlled by a single set of rules.

6.3.2 Examples of possible applications

Obviously, the system is suited to implement a variety of card game applications, provided these games can be modelled with one set of rules only, like poker.

Another family of applications, for which it would be extremely easy to define new classes, is that of games. In particular, scrabble, chess, Monopoly or others would be easily implemented. With non-card games applications, it is necessary to create new classes of multi-user objects. As was explained previously, this process is quite straightforward. For instance, scrabble would only require a set of letters to replace cards and chips. Indeed, almost the same rules as poker's can be directly reused without modification.

Among non-game applications, examples of situations involving rich inter-user relationships include brainstorming or educative programs.

Chapter 7

Conclusion and suggestions for future work

After reviewing the fields of multi-user systems, knowledge representation and object-oriented methodology, we focused on the example of poker to list the requirements for display management in a multi-user game system. The solution proposed is based on a production system featuring two types of rules. Rules are written in a simplified logic language coded in C++. Access permissions were set by giving a semantic to the application, allowing to define different roles for the participants (e.g. Dealer) and to characterize objects on an individual basis (e.g. Table). Tests proved that response times were good, and that the application was convenient to use. They also highlighted several limitations, the most important being that the scope of possible applications is limited to those that can be modelled with a single set of rules.

In a more general way, we can now determine what options proved to be justified, and which ones should be modified. Several principles seem to have worked correctly: ease of use for the interface, the no-floor control approach and multiple telepointers that appear to be essential in highly interactive applications, choice of a rule-based system, reduced complexity of the rule syntax to preserve real-time responses, and the use of X Window for communications. Problems include the absence of metarules and the poor management of screen space.

The introduction of metarules would broaden the scope of possible applications. Each session would be divided in several distinct phases, as explained in Chapter 5. For now, the rules controlling user actions have no influence on the rule-based system. It would be possible to define a set of actions to be taken after a specific user request. For instance, instead of having only two possible actions (i.e. AddFact and DeleteFact), one could count the number of occurrences of some actions, and trigger some procedure when a condition is verified. Other actions would consist in moving from one game phase to another.

Concerning screen space, the solution seems to consist in allowing area resizing, or in displaying the same data at different scales. The difference between these two solutions is that, by different scales, we mean that some objects would be displayed at a larger scale than some others, but it would not be possible to resize them. In the case of resizing, it is possible to use a system of constraints ensuring that no area would overlap another one (see [Mye90] for an example of UIMS for single-user interfaces based on constraints). In order to ensure that users have a complete view of the group's activity at every moment, window resizing would also provoke the resizing of the graphical objects

located inside the window.

Area resizing or rescaling is related to the question of user-centered screen layouts. In the implementation of the system, the same interface layout is used for all users. One of the conclusions from the tests is that differing displays would be useful. By differing we mean that all the displays share the same general topology, and can be seen as the result of rotations, scalings or translations applied to a base configuration. In the case of rotations, as there are only four sides in a display, there are only four possible differing configurations. Starting from a base configuration, the other three are obtained through rotations of angles equal to 90, 180 or 270 degrees. At creation time, each view of each area could then transform its coordinates according to the layout information (i.e. the value of the angle). The center of each rotation is the center of the minimal bounding rectangle for the base configuration. The values of the angles ensure that transformed coordinates are always integer values. In some cases, however (e.g. menus), it may be desirable to declare some objects as invariant, i.e. appearing at the same location on every screen. As the same area is to be displayed with different orientations on different screens, it is necessary to rotate the views of the objects (e.g. cards) contained in its different views. The preceding procedure would allow to extend the user-centered aspect of the system to geometrical questions. In particular, each player could see his cards at the bottom of his display. If the system is to be extended with desktop video using direct eye-contact principles such as presented in [BM90], the geometrical layout of the hands of the players on any screen would match the view each user has of the others on the same video display device. In such a system, each participant would have a realistic feeling of where the others are spatially located.

Another problem deals with the fact that there is no explicit facility to represent time in the system. Such a facility would present several advantages. Firstly, new rules could be defined, that would include time constraints in their precondition lists. Secondly, synchronization problems could be tackled effectively. Indeed, in the current system, synchronization of workstations is based on the optimistic assumption that events will be reported simultaneously. Synchronization of multi-media documents could be introduced as well. In particular, the poker session itself could be viewed as a multimedia document. Time information would permit to replay it at different speeds, or to restart it from a given point. The notion of atomic action makes it easy to record a session by keeping trace of all the events occurring.

Future work could fruitfully introduce these new concepts. The combination of object-oriented techniques and production rules ensures a greater flexibility and adaptivity which should make modifications easy to carry out.

Also interesting would be to add a voice conferencing facility, or, at least, a mailing feature. One may think of a system of permissions that would apply on vocal interactions in the same way as they are defined for graphics. In particular, new rules could be written to decide who can talk to whom, and whether their conversation should be accessible by the others. Communications would have to be easy to set up, to preserve the naturalness of the system. The same ideas could be applied in the case of a mail system.

Another aspect that would be interesting to develop deals with the integration of single-user programs in multi-user environments, as it is likely that the former will still occupy the largest place in future computer systems. A virtual terminal protocol based on X Window could be added to the system. However, screen space problems will emerge. In particular, screen switching can be used [Sak90], although it should be implemented with a mechanism permitting to watch for possible changes on the screen displaying the multi-user application, to keep every user aware of what may be

happening in the group.

Even though this work deals primarily with games, it is expected that it could find applications in other fields. In particular, as it is likely that multi-user systems will appear soon in everyday office environment, or in computer aided teaching, there will probably be a need for a management of shared workspaces, involving different views of the same objects and permissions on actions according to the identity of the requestor. Our feeling is that artificial intelligence will be helpful to resolve the problems raised by these systems, provided implementations are fast enough.

Appendix A

Sample set of rules and facts for poker

A.1 Facts file

Below is the Facts file, as actually written in the current implementation.

```
   // Note that it is possible to write comments like this line

   // First tell Carl is the dealer
Dealer (Carl);

   // Then define Ownareas relationships
Ownarea (Carl, area1);
Ownarea (Sheila, area2);
Ownarea (Lisa, area3);
Ownarea (Bill, area4);

   // Where is the table
Table (area5);

   // Turn-taking relationships
SuccessorOf (Carl, Sheila);
SuccessorOf (Sheila, Lisa);
SuccessorOf (Lisa, Bill);
SuccessorOf (Bill, Carl);

   %% // This sign indicates the separation between facts and display permissions which are actually
rules

   // Display permissions ; DisplayPermission is a reserved keyword


DisplayPermission (X, Y, NormalDisplay) if
                    Table (Y);
```

```
DisplayPermission (X, Y, HideDisplay) if
                Ownarea (U, Y) and Different (U, X) and Table (Z) and
                Different (Y, Z);


DisplayPermission (X, Y, NormalDisplay) if
                Ownarea (X, Y);


   //end of Facts file
```

A.2 Rules file

```
// every player is allowed to handle objects in his own area
//no rules necessary: implicit
// no player can select or flip objects in another player's own area
X:Card::SelectionRequest (Y):PermissionRefused if Different (X, Z) and Ownarea (Z, Y);
X:Chip::SelectionRequest (Y):PermissionRefused if Different (X, Z) and Ownarea (Z, Y);
// every player can handle objects inside the table area
// no rules necessary: implicit
// every player can move objects from his own area to the table area
// no rules necessary: implicit
// only the dealer can move cards from the table to someone's own area


X:Card::EndMove (Y, Z):PermissionRefused if Ownarea (W, Z) and Table (Y)
                                        and Dealer (T) and Different (T, X);


X:Card::EndMove (Y, Z):HideDisplay if Ownarea (W, Z) and Table (Y)
                                    and Different (W, X) and Dealer (X);


// every player can move chips from the table area to his own area
// no rules necessary: implicit
// no player can move objects from another player's own area to his
// already implicit in the rules preventing users from selecting objects
// in others own areas
// every player can move objects from his own area to another player's

X:Card::EndMove (Y, Z):HideDisplay if Ownarea (W, Z) and Ownarea (X, Y)
                                    and Different (X, W);
```

```
%%
// now special rule to restart the session

Z:NewSession [DeleteFact {Dealer (X)} and AddFact {Dealer (Y)}
                        if Dealer (X) and SuccessorOf (X, Y)]
            if Dealer (Z);


// end of Rules specification file
```

Appendix B

Overview of the class hierarchy

In this appendix, we do not reproduce exactly the header files used in the system. To make things clearer, several unimportant lines, irrelevant here, have been skipped. Each section corresponds to one of the actual header files.

B.1 Area

Area is a subject connected to several AreaView views.

```
class Area : public GameSubject {
        public:
                Area (Session*);
                ~ Area ();
                char* GetName ();
        private:
                friend class AreaView;
                char name[64];
                static int index;
};
```

B.2 AreaLayout

Keeps trace of the repartition of the various areas on a given display (ie. Player).

```
class AreaData : public BaseObj {
                public:
                        Position x;
                        Position y;
                        Dimension width;
                        Dimension height;
                        AreaView* areaview;
};
```

```
class PointData : public BaseObj {
                public:
                        Position x;
};

class WidgetData : public BaseObj {
                 public:
                         Widget w;
};

class AreaLayout : public BaseObj {
                 public:
                         AreaLayout (BaseList*);
                         AreaView* GetAreaView (Position, Position);
                         ~ AreaLayout ();
                         void InitCover ();
                         Player* GetPlayer ();
                         void RaiseCovers ();
                 private:
                         Player* player;
                         BaseList* viewlist;
                         BaseList* areadatalist;
                         BaseList* SortHorizontal ();
                         BaseList* SortVertical ();
                         void CreateCovers (BaseList*, BaseList*);
                         void CreateBox (Position, Position, Position, Position);
                         BaseList* widgetlist;
                         int cnt;
                         Arg wargs[8];
};
```

B.3 AreaView

Player and VirtualDisplay specific view of an Area.

```
class AreaView : public ViewObj {
               public:
                       AreaView (GameSubject*, Player*);
                       void Update ();
                       long GetPermission ();
                       void SetPermission (long);
               private:
                       long permission;
};
```

B.4 BaseObj

Base class for most objects. Implements basic list operations.

```
class BaseList {
        friend class BaseObj;
        public:
                BaseObj* element;
                BaseList* next;
};

class BaseObj {
        public:
                BaseList* AddItem (BaseObj*, BaseList*);
                BaseList* RemItem (BaseObj*, BaseList*);
                BaseList* AppendList (BaseList*, BaseList*);
                int ListLength (BaseList*);
                void DeleteList (BaseList*);
                BaseObj ();
                virtual ~ BaseObj ();
                virtual void Update ();
                virtual void Redraw (GameSubject*);
                virtual void Release (GameSubject*);
                virtual void Raise ();
                virtual char* GetClass ();
};
```

B.5 Card

A subject connected to several CardView views.

```
class Card : public GameSubject {
        public:
                Card (Session*, char*);
                ~ Card ();
                char* GetName ();
        private:
                int width;
                int height;
                friend class CardView;
                char name[64];
                static int index;
        protected:
                char bitmapname[64];
```

```
                char backbitmapname[64];
};
```

B.6 CardView

Player and VirtualDisplay specific view of a Card.

```
class CardView : public GraphicObj {
            public:
                CardView (GameSubject*, Player*);
                ~ CardView ();
                void Redraw (GameSubject*);
                void Release (GameSubject*);
                void Raise ();
                char* GetClass ();
            private:
                Pixmap pixmap;
};
```

B.7 Chip

Similar to Card, but for chips.

```
class Chip : public GameSubject {
        public:
            Chip (Session*, char*);
            ~ Chip ();
            char* GetName ();
        private:
            int width;
            int height;
            friend class ChipView;
            char name[64];
            static int index;
        protected:
            char bitmapname[64];
            char backbitmapname[64];
};
```

B.8 ChipView

```
class ChipView : public GraphicObj {
            public:
```

```
                ChipView (GameSubject*, Player*);
                ~ ChipView ();
                void Redraw (GameSubject*);
                void Release (GameSubject*);
                void Raise ();
                char* GetClass ();
        private:
                Pixmap pixmap;
};
```

B.9 Cursor

Cursor object : subject consisting of several CursorView views.

```
class Cursor : public GameSubject {
        public:
                Cursor (Session*, Player*);
                ~ Cursor ();
                void MoveCursor (Widget, caddr_t, XEvent*);
                void RaiseCursorViews ();
                char* GetName ();
        private:
                friend class CursorView;
                char cursorname[64];
};
```

B.10 CursorView

```
class CursorView : public GraphicObj {
        public:
                CursorView (GameSubject*, Player*);
                ~ CursorView ();
                void Update ();
};
```

B.11 GameSubject

Base Class for objects used in a card game : Cards, Chips, Cursor...

```
class GameSubject : public Subject {
        friend class GraphicObj;
        friend class CardView;
        friend class CursorView;
```

```
        friend class ChipView;
        public:
                GameSubject ();
                ~ GameSubject ();
                virtual void AddPlayer (BaseObj*);
                virtual void RemPlayer (BaseObj*);
                virtual char* GetName ();
                Session* GetSession ();
                BaseList* GetPlayerList ();
        protected:
                Position delta_x, delta_y, last_x, last_y;
                Position widget_posx, widget_posy;
                // widget_ownx = position of the cursor relative to the view widget
                Position widget_ownx, widget_owny;
                // x_save = initial widget_posx at SelectRequest
                Position x_save, y_save;
                BaseList* playerlist;
                Session* session;
                Boolean requestorhidden; // used to determine the state of
                // the requestor
};
```

B.12 GraphicObj

Base class for graphics. The idea is in the widget.

```
class GraphicObj : public ViewObj {
        public:
                void SelectRequest (Widget, caddr_t, XEvent*);
                void EndMove (Widget, caddr_t, XEvent*);
                void Move (Widget, caddr_t, XEvent*);
                GraphicObj ();
        protected:
                void ProcessMove ();
                void NotifyRedraw (GameSubject*);
                void NotifyRelease (GameSubject*);
                void NotifySelect ();
};
```

B.13 Menu

Menu object : subject consisting of several MenuView views.

```
class Menu : public GameSubject {
```

```
        public:
                Menu (Session*, BaseList*);
                ~ Menu ();
                char* GetName ();
                void RaiseMenuViews ();
        private:
                friend class MenuView;
                char name[64];
                static int index;
};
```

B.14 MenuView

Player and VirtualDisplay specific view of a Menu.

```
class MenuInfo : public BaseObj {
        public:
                MenuInfo (char*, XtCallbackProc);
                char* str;
                XtCallbackProc func;
                void GiveInformation (Widget, caddr_t, XEvent*);
};

class MenuView : public ViewObj {
        public:
                MenuView (GameSubject*, Player*, BaseList*);
                void Update ();
                void NewSession (Widget, caddr_t, caddr_t);
        private:
                Widget button[5];
};
```

B.15 Player

Player is actually a VirtualDisplay with a BulletinBoard widget.

```
class Player : public VirtualDisplay {
        public:
                Player (char*, XtAppContext, int*, char**);
                ~ Player ();
                Widget GetFrame ();
                Widget GetUnderFrame ();
                Widget GetPopup ();
                Widget GetEmerg ();
```

```
        Widget GetQuestionDia ();
        void CallError (Widget, caddr_t, caddr_t);
        void CallEmerg (Widget, caddr_t, caddr_t);
        void CallCancel (Widget, caddr_t, caddr_t);
        void CallOk (Widget, caddr_t, caddr_t);
        void ShowMenu (XEvent*, Position, Position);
        void ShowEmerg (XEvent*);
        void ShowQuestionDia (XEvent*);
        char* GetName ();
        char* GetCursorName ();
        void SetCursorName (char*);
        void SetCursor (Cursor*);
        Cursor* GetCursor ();
        void InitAreaLayout (BaseList*);
        AreaLayout* GetAreaLayout ();
        void GiveInformation (XEvent*);
private:
        Widget frame;
        Widget underframe;
        Widget popup;
        Widget errmesg;
        Widget emergpopup;
        Widget emerg;
        Widget questiondia;
        Widget bulle;
        Widget textbulle;
        static int index;
        char name[64];
        char cursname[64];
        Cursor* cursor;
        ModifyRule* mfrule;
        char* curcard[7];
        XmString xmstr[7];
        char* plname;
        char* plstat;
protected:
        AreaLayout* arealayout;
};
```

B.16 Resource

Resources are shared objects.

```
class Resource : public BaseObj {
```

```
        public:
                Resource();
                virtual ~ Resource();
                void Reference();
                void Unreference();
        private:
                friend void Unref(Resource*);
                unsigned refcount;
};
```

B.17 Rule

Rule object.

ParamList is a list of StringObj, used to represent predicates or functions.

```
class ParamList : public BaseObj {
        public:
                ParamList (BaseList*);
                ParamList ();
                ~ ParamList ();
                BaseList* GetParamList ();
                void Substitute (char*, char*);
                char* GetNthParam (int);
                void SetNthParam (int, char*);
                BaseList* Duplicate ();
                void Display ();
        protected:
                BaseList* paramlist;
};
```

PredicateList is a list of predicates.

```
class PredicateList : public BaseObj {
        public:
                PredicateList (BaseList*);
                PredicateList ();
                ~ PredicateList ();
                BaseList* GetPredicateList ();
                void Substitute (char*, char*);
                PredicateList* Duplicate ();
                PredicateList* GetCdr ();
                void Display ();
        protected:
                BaseList* predicatelist;
};
```

```
class Predicate : public ParamList {
                public:
                        Predicate (char*, BaseList*);
                        ~ Predicate ();
                        char* GetName ();
                        Boolean Resolve (BaseList*);
                        void Display ();
                private:
                        char* name;
};

class ActionRule : public ParamList {
                 public:
                        ActionRule (char*, char*, BaseList*, long);
                        ActionRule (ifstream&, streampos&, streampos&);
                        ~ ActionRule ();
                        char* GetParty ();
                        char* GetMethod ();
                        long GetPermission ();
                        void SetParty (char*);
                        void Display ();
                 private:
                        char* party;
                        char* method;
                        long permission;
};

class ModifyList : public BaseObj {
                public:
                        ModifyList (char*, char*, PredicateList*, PredicateList*,
                        PredicateList*);
                        ModifyList (ifstream&, streampos&, streampos&);
                        ~ ModifyList ();
                        void Substitute (char*, char*);
                        char* GetParty ();
                        PredicateList* GetDeleteList ();
                        PredicateList* GetAddList ();
                        PredicateList* GetModifCondList ();
                        char* GetMethod ();
                        void SetParty (char*);
                        void Display ();
                        Predicate* GetFact (ifstream&, streampos&, streampos&);
                private:
                        char* party;
                        char* method;
```

```
                    PredicateList* deletelist;
                    PredicateList* addlist;
                    PredicateList* modifcondlist;
};

class Rule : public BaseObj {
        public:
                Rule (ActionRule*, PredicateList*);
                ~ Rule ();
                PredicateList* GetPremiseList ();
                void SetPremiseList (PredicateList*);
                ActionRule* GetActionRule ();
                void Substitute (char*, char*);
                Rule* Duplicate ();
                void Display (); // Display what is in the rule
        private:
                ActionRule* actionrule;
                PredicateList* premiselist; // list of predicates
};

class ModifyRule : public BaseObj {
                public:
                        ModifyRule (ModifyList*, PredicateList*);
                        ~ ModifyRule ();
                        PredicateList* GetPremiseList ();
                        void SetPremiseList (PredicateList*);
                        ModifyList* GetModifyList ();
                        void Substitute (char*, char*);
                        ModifyRule* Duplicate ();
                        void Display (); // Display what is in the rule
                private:
                        ModifyList* modifylist;
                        PredicateList* premiselist; // list of predicates
};
```

B.18 Session

A session runs the whole game, and includes the inference engine for the production system.

```
class Session : public GameSubject {
            public:
                Session (char*, XtAppContext, int*, char**);
                ~ Session ();
                BaseList* FindAreaViews (Player*);
```

```
        void InitPlayers (ifstream&, XtAppContext, int*, char**);
        void InitCards (ifstream&);
        void InitChips (ifstream&);
        void InitAreas (ifstream&);
        void InitFacts (ifstream&);
        void InitRules (ifstream&);
        void InitMenus ();
        long GetPermission (char*, char*, char*, char*);
        long GetPermission (char*, char*, char*);
        BaseObj* FindObject (char*);
        void InitPermissions (Rule*);
        Rule* GetRule (ifstream&, streampos&, streampos&);
        ModifyRule* GetModifyRule (ifstream&, streampos&, streampos&);
        Rule* GetDisplayPermission (ifstream&, streampos&, streampos&);
        int IsAVariable (char*);
        Boolean Unify (Rule**, Predicate*);
        BaseList* FindSubstitutions (Predicate*, int);
        Boolean ProcessRule (PredicateList*);
        void ProceedDestroy (ModifyRule*);
        void ProcessNewSession (char*);
        Boolean ProcessModif (PredicateList*, PredicateList*,
        PredicateList*);
        Predicate* GetFact (ifstream&, streampos&, streampos&);
    private:
        static char VariableNames[ ];
        BaseList* arealist;
        BaseList* cardlist;
        BaseList* chiplist;
        BaseList* factlist;
        BaseList* rulelist;
        BaseList* menulist;
        BaseList* modifyrulelist;
};
```

B.19 StringObj

Convenience class for strings.

```
class StringObj : public BaseObj {
        public:
            StringObj (char*);
            ~StringObj ();
            char* GetValue ();
            void SetValue (char*);
```

```
        protected:
                char* value;
};
```

B.20 Subject

A subject is an object that has one or more views that it wishes to notify when it changes.

```
class Subject : public Resource {
        public:
                Subject();
                ~ Subject();
                virtual void Attach(BaseObj*);
                virtual void Detach(BaseObj*);
                virtual void Notify();
                Boolean IsView(BaseObj*);
                BaseList* GetViews ();
        protected:
                BaseList* views;
};
```

B.21 VirtualDisplay

A VirtualDisplay is a virtual screen area. Future extensions could permit to have a VirtualDisplay distributed on several displays.

```
class VirtualDisplay : public BaseObj {
                public:
                        VirtualDisplay (char*, XtAppContext, int*, char**);
                        ~ VirtualDisplay ();
                        Widget GetToplevel ();
                        Widget GetCursorShell ();
                        Display* GetDisplay ();
                private:
                        Display* display;
                protected:
                        Widget toplevel;
};
```

B.22 ViewObj

Base class for views.

```
class ViewObj : public BaseObj {
        public:
                ViewObj ();
                Widget GetWidget ();
                Player* GetPlayer ();
                GameSubject* GetSubject ();
        protected:
                Widget widget;
                GameSubject* subject;
                int cnt;
                Arg wargs[8];
                Player* player;
                char* source; // source area for an action
                char* dest; // destination area to an action
                Boolean requestor; // true if Player is the requestor
                // of an action
                long permission; // permission for the current action
                // as returned by Session::GetPermission
                Boolean hidden; // true if the object is displayed hidden, e.g.
                // back side up for a card
};
```

Bibliography

[AHE88] S.R. Ahuja, D.N. Horn, and J.R. Ensor. Networking requirements of the Rapport multimedia conferencing system. In *Proceedings of IEEE Infcom '88*, pages 746–751, 1988.

[AKN86] L.M. Applegate, B.R. Konsynski, and J.F. Nunamaker. A group decision support system for idea generation and issue analysis in organization planning. In *Proceedings of the First Conference on Computer Supported Cooperative Work, Austin, Tex., Dec.3-5*, pages 16–34. ACM, New York, 1986.

[All83] J.F. Allen. Maintaining knowledge about temporal events. *Communications of the ACM*, 26(11):832–843, November 1983.

[AMY88] R.M. Akscyn, D.L. McCracken, and E.A. Yoder. KMS: A distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[Ank85] Nesmith C. Ankeny. *Le poker: règles, stratégies, exercices.* Rocher, Monaco, 1985.

[AW90] H.M. Abdel-Wahab. Multiuser tools architecture for group collaboration in computer networks. *Computer Communications*, 13(3):165–169, April 1990.

[AWGN88] Hussein M. Abdel-Wahab, Sheng-Uei Guan, and Jay Nievergelt. Shared workspaces for group collaboration : an experiment using Internet and UNIX interprocess communications. *IEEE Communications Magazine*, pages 10–16, November 1988.

[B+89] R. Breitl et al. The GemStone data management system. In W. Kim and F. Lochovsky, editors, *Object-oriented concepts, databases, and applications*, pages 283–308. Addison-Wesley, Reading, Mass., 1989.

[Bak89] Ron Baker. Videoconferencing: an update. *Telecommunications*, 23(12):31–34, December 1989.

[Bar88] F. Barachini. How production systems can survive in real-time process control. In *Avignon '88, 8th international workshop on expert systems and their applications, Avignon, France, May 30 - June 3*, pages 217–234. Gerfau, Paris, 1988.

[Bar90] T. Barzilai. A multicast communication setup protocol for multimedia applications. In *Proceedings of Multimedia '90, Bordeaux, France, 3rd IEEE ComSoc International Workshop on Multimedia Communications, Nov. 14-17*, 1990.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming expert systems in OPS5.* Addison-Wesley, Boston, MA, 1985.

[BG81] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[Blo79] T. Bloom. Evaluating synchronization mechanisms. In *7th Symposium on Operating Systems Principles, Pacific Grove, Calif., December 10-12*, pages 24–32. ACM, 1979.

[BM90] Bill Buxton and Tom Moran. EUROPARC's integrated interactive intermedia facility (IIIF) : Early experiences. In S. Gibbs and A.A. Verrijn-Stuart, editors, *Multi-user interfaces and applications*, pages 11–34. Elsevier Science Publishers B.V. (North Holland), 1990.

[BS90] C. Bertin and Y. Surzur. Protocols for multimedia interactive applications. In *Proceedings of Multimedia '90, Bordeaux, France, 3rd IEEE ComSoc International Workshop on Multimedia Communications, Nov. 14-17*, 1990.

[BTW91] R. Brennan, K. Thompson, and R. Wilder. Mapping the X Window onto Open Systems Interconnection standards. *IEEE Network Magazine*, pages 32–40, May 1991.

[Buc86] B.G. Buchanan. Expert systems: working systems and the research literature. *Expert Systems*, 3(1):32–51, January 1986.

[CF89] Terrence Crowley and Harry Forsdick. MMConf: The Diamond multimedia conferencing system. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[Che89] J.Bradley Chen. A toolkit for multi user window management. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[Cho56] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2:113–124, 1956.

[Cla91] M.A. Clarkson. An easier interface. *Byte*, pages 277–282, February 1991.

[CMB+90] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson. MMConf: an infrastructure for building shared multimedia applications. In *Proceedings of the 3rd Conference on Computer Supported Cooperative Work, Los Angeles, Calif., Oct.8-10.* ACM, New York, 1990.

[Cou85] J. Coutaz. Abstractions for user interface design. *IEEE Computer*, pages 21–34, September 1985.

[D+90] O. Deux et al. The story of O2. *IEEE transactions on knowledge and data engineering*, 2(1):91–108, March 1990.

[Dav80] R. Davis. Metarules: reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[DC91] P. Dewan and R. Choudhary. Flexible user interface coupling in a collaborative system. In *CHI'91 Conference Proceedings, New orleans, Louisiana, April 27 - May 2*, pages 41–48. Addison Wesley, 1991.

[DMN70] O.J. Dahl, B. Myhrhaug, and K. Nygaard. *The Simula67 common base language.* Publication S22, Norwegian Computing Centre, Oslo, 1970.

[doa88] Information processing systems - text and office systems - distributed office applications model, April 1988. ISO/IEC JTC1/SC18/WG4/ DP 10031.

[DS84] M. Doyle and D. Straus. *How to make meetings work.* Berkeley publishing group, New York, 1984.

[EE68] D.C. Engelbart and W.K. English. A research center for augmenting human intellect. In *Fall Joint Computing Conference, AFIPS Proceedings Volume 33. Part 1, Reston, Va.*, pages 395–410, 1968.

[EGR91] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware : Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[Els73] Mark Elson. *Concepts of programming languages.* SRA Science Research Associates, Inc., 1973.

[ES90] M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[F+85] K. Futatsugi et al. Principles of OBJ2. In *Proceedings of the 12th symposium on principles of programming languages, ACM*, pages 52–66, 1985.

[F+87] D.H. Fishman et al. IRIS: an object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[For85] H.C. Forsdick. Exploration into real-time multimedia conferencing. In *Proceedings Second International Symposium on Computer Message Systems*, pages 299–315, September 1985.

[Fou90] Open Software Foundation, editor. *OSF/Motif Programmer's Guide, Revision 1.0.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[GLACL89] J.J. Garcia-Luna-Aceves, E.J. Craighill, and R. Lang. Floor management and control for multimedia computer conferencing. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[GR83] A. Goldberg and D. Robson. *SMALLTALK-80, the language and its implementation.* Addison-Wesley, Reading, Mass., 1983.

[Gru91] J. Grudin. CSCW: the convergence of two development contexts. In *CHI'91 Conference Proceedings, New orleans, Louisiana, April 27 - May 2*, pages 91–97. Addison Wesley, 1991.

[GS90] P. Gasparotti and C. Simone. A user defined environment for handling conversations. In S. Gibbs and A.A. Verrijn-Stuart, editors, *Multi-user interfaces and applications*, pages 271–289. Elsevier Science Publishers B.V. (North Holland), 1990.

[GSW86] I. Greif, R. Seliger, and W. Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the 13th annual symposium on principles of programming languages, St Petersbourg, Fla.*, pages 160–172, January 1986.

[Gus88] Philip Gust. SharedX: X in a distributed group work environment. In *Presentation at the 2nd Annual X Technical Conference, MIT*, January 1988.

[Gus89] Philip Gust. Multi-user interface project at Hewlett-Packard laboratories. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[HC86] D.A. Henderson and S.K. Card. Rooms: the use of multiple virtual workspaces to reduce space contentions in a window-based graphical user interface. *ACM Transactions on Graphics*, 5(3):211–243, July 1986.

[Hef84] Gordon Heffron. Teleconferencing comes of age. *IEEE Spectrum*, pages 61–66, October 1984.

[Hil91] Raplh D. Hill. A 2-D graphics system for multi-user interactive graphics based on objects and constraints. In *CSCW : new perspectives - University of Toronto, February 8*, 1991.

[HKN89] R. Hunter, P. Kaijser, and F. Nielsen. ODA: a document architecture for open systems. *Computer Communications*, 12(2):69–79, April 1989.

[Hoa87] C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, pages 85–91, September 1987.

[Hod89] P. Hodges. A relational successor ? *Datamation*, pages 47–50, November 1989.

[Hu89] David Hu. Building knowledge representation language structures. In *C/C++ for expert systems*, chapter 7. MIS Press, 1989.

[Int87] Interactive Software Engineering Inc. *Eiffel user's manual*, 1987. TR-EI-5/UM.

[IO90] H. Ishii and M. Ohkubo. Design of TeamWorkStation: a realtime shared workspace fusing desktops and computer screens. In S. Gibbs and A.A. Verrijn-Stuart, editors, *Multi-user interfaces and applications*, pages 131–141. Elsevier Science Publishers B.V. (North Holland), 1990.

[ip85] *DDN Protocol Handbook, Vol.1, DCA, Internet protocol*, December 1985. MIL-STD 1777.

[Ish89] Hirishi Ishii. COOKBOOK. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[Joh75] S.C. Johnson. Yacc – yet another compiler-compiler. Technical report, Computer Sci. Techn. Report 32, Bell Labs, Murray Hill, N.J., 1975.

[Lan88] K.A. Lantz. An experiment in integrated multimedia conferencing. In Irene Greif, editor, *Computer Supported Cooperative Work : a Book of Readings*, chapter 19, pages 533–552. Morgan Kaufmann Publishers, Inc., 1988.

[Lee90] Jeffrey J. Lee. Xsketch : a multi-user sketching tool for X11. In *Proceedings for the Conference on Office Information Systems, Cambridge, Mass., April 25-27*, pages 169–173. ACM Press, 1990.

[Lel88] W. Leler. *Constraint programming languages: their specification and generation.* Addison-Wesley, New York, 1988.

[LH88] B.T. Lewis and J.D. Hodges. Shared Books: collaborative publication management for an office information system. In *Proceedings of the conference on office information systems, Palo Alto, Calif.*, pages 197–204, March 1988.

[LL90] J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *CHI '90 Proceedings*, pages 303–311, April 1990.

[LLA+89] Keith A. Lantz, Chris Lauwers, Barry Arons, et al. Collaboration technology research at Olivetti research centre. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[LMB91] Gifford Louie, Marylin Mantei, and Bill Buxton. Making contact in a multi-media environment. *HCI Consortium on Computer Supported Cooperative Work, Ann Arbor, MI, Feb. 2-5*, 1991.

[LVC89] M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, February 1989.

[MAH90] R. Mantyla, J. Alasuvanto, and H. Hammainen. PAGES: a testbed for groupware applications. In S. Gibbs and A.A. Verrijn-Stuart, editors, *Multi-user interfaces and applications*, pages 37–47. Elsevier Science Publishers B.V. (North Holland), 1990.

[McD80] J. McDermott. R1: an expert in the computer systems domain. In *Proc. American Association for Artificial Intelligence, Vol.1*, pages 269–271, 1980.

[MGT+87] T.W. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987.

[Min75] M. Minsky. A framework for representing knowledge. In Patrick Winston, editor, *The psychology of computer vision*, pages 211–277. McGraw-Hill, 1975.

[MMM91] J.C. McCarthy, V.C. Miles, and A.F. Mink. An experimental study of common ground in text-based communication. In *CHI'91 Conference Proceedings, New orleans, Louisiana, April 27 - May 2*, pages 209–215. Addison Wesley, 1991.

[Moo89] D.A. Moon. The common Lisp object-oriented programming language standard. In W. Kim and F. Lochovsky, editors, *Object-oriented concepts, databases, and applications*, pages 49–78. Addison-Wesley, Reading, Mass., 1989.

[Mye90] B.A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, April 1990.

[Nau60] P. Naur, editor. *Revised report on the algorithmic language Algol60.* International Federation of Information Processing, 1960.

[Nic90] Cosmos Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, April 1990.

[Nie90] Jacob Nielsen. *Hypertext and hypermedia.* Academic Press, 1990.

[NTH+90] K. Nakamura, M. Takahashi, N. Hamada, T. Inuzuka, N. Yanai, S. Maruo, and K. Kaneda. Personal multimedia teleconferencing terminal. In *IEEE International Conference on Communications, Alberta, Georgia April 16-19*, pages 123–127, 1990.

[OBS85] G.M. O'Hare and D.A. Bell. The coexistence approach to knowldege representation. *Expert systems*, 2(4):230–238, January 1985.

[Ont89] Ontologic, Inc., Billerica, MA. *Client library reference*, 1989.

[OSKQ90] F. Oguet, C. Schwartz, F. Kretz, and M. Quere. RAVI, a proposed standard for the interchange of audio/visual interactive applications. *IEEE Journal on Selected Areas in Communications*, 8(3):428–436, April 1990.

[P+88] Palay et al. The Andrew toolkit: an overview. In *Proceedings of the USENIX Winter Conference, Dallas, TX, February 9-12*, pages 9–21, 1988.

[Pat90] John F. Patterson. The implications of window sharing for a virtual terminal protocol. In *IEEE International Conference on Communications ICC '90 Supercomm '90, Atlanta, GA, USA, 16-19 April*, volume 1, pages 66–70, 1990.

[Pau86] L.F. Pau. Survey of expert systems for fault detection, test generation and maintenance. *Expert systems*, 3(2):100–111, April 1986.

[PGLAC+85] A. Poggio, J.J. Garcia Luna Aceves, E.J. Craighill, D. Moran, L. Aguilar, D. Worthington, and J. Hight. CCWS : A computer-based, multimedia information system. *IEEE Computer*, pages 92–103, October 1985.

[PHMR91] J. F. Patterson, R. D. Hill, W.S. Meeks, and S. L. Rohall. Rendezvous : an architecture for synchronous multi-user applications. In *CSCW : new perspectives - University of Toronto, February 8*, 1991.

[PL89] M.D. Palmer Leland. Accomodating individuals' preferences in groupware. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California.* IFIP working group 8.4, 1989.

[Qui68] M.R. Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, pages 216–270. MIT Press, Cambridge, Ma, 1968.

[Rei89] Gail L. Rein. Statement of interest. In *Groupware Technology Workshop - August 24 and 25, Palo Alto, California*. IFIP working group 8.4, 1989.

[RIIM+88] J. Rosenberg, R. Hill, J. Miller, A. Schubert, and D. Shewmake. UIMSs: threat or menace ? In *CHI'88 Conference Proceedings, Washington, D.C., May 15-19*, pages 197–200. Addison Wesley, 1988.

[Rob65] J.A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RSSS90] J. Ruckert, H. Schmutz, B. Schoner, and R. Steimetz. A distributed multimedia environment for advanced CSCW applications. In *Proceedings of Multimedia '90, Bordeaux, France, 3rd IEEE ComSoc International Workshop on Multimedia Communications, Nov. 14-17*, 1990.

[SA75] R.C. Schank and R.P. Abelson. Scripts, plans and knowledge. In *Proceedings 4th International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, 3-8 Sept.*, volume 1, pages 151–157, 1975.

[Sak90] Shiro Sakata. Development and evaluation of an in-house multimedia desktop conference system. *IEEE Journal on Selected Areas in Communications*, 8(3):340–347, April 1990.

[SBF+87] M. Stefik, D.G. Bobrow, G. Foster, S. Lanning, and D. Tartar. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Trans. Off. Info. Sys*, 5(2):147–186, April 1987.

[SBK+87] M. Stefik, D.G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[Sch86] K. Schmucker. *Object-oriented programming for the Macintosh.* Hayden Book Co., Hasbrouck Heights, N.J., 1986.

[Scr90] C.J. Screenan. Synchronized retrieval of multi-media data. In *First international workshop on network and operating system support for digital audio and video, Nov.8-9, Berkeley, Calif.*, 1990.

[SG86] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SG88] Sunil Sarin and Irene Greif. Computer-based real-time conferencing systems. In Irene Greif, editor, *Computer Supported Cooperative Work : a Book of Readings*, chapter 15, pages 397–421. Morgan Kaufmann Publishers, Inc., 1988.

[Sim85] G.L. Simons. *Expert systems and micros.* NCC Publications, Manchester, England, 1985.

[Son88] D.H. Sonnenwald. Applying artificial intelligence techniques to human-computer interfaces. *IEEE Communications Magazine*, 26(3):14–20, March 1988.

[Sou90] A. Southerton. Many paths to X Window programming. *Unix World*, pages 83–87, May 1990.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of Postgres. *IEEE Transactions on knowledge and data engineering*, 2(1):125–142, March 1990.

[SS86] L. Sterling and E. Shapiro. *The art of Prolog.* MIT Press, Cambridge, Ma, 1986.

[Ste90] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE Journal on Selected Areas in Communications*, 8(3):401–412, April 1990.

[Sun87] Sun Microsystems. *NeWS Manual, 800-1632-10, Revision A, 29 March*, 1987.

[Sut63] I.E. Sutherland. SketchPad: a man-machine graphical communication system. In *AFIPS Spring joint computer conference*, pages 329–346. AFIPS Press, Reston, Va., 1963.

[Swi87] D.C. Swinehart. Telephone management in the Etherphone system. In *IEEE Glocom 87, Tokyo, Japan, Nov.15-18*, volume 2, pages 1176–1180, 1987.

[Tan87] S. Tanimoto. *The elements of Artificial Intelligence: an introduction using LISP.* Computer Science Press, Rockville, MD, 1987.

[Taz88] J. Tazelaar. In depth: groupware. *BYTE*, December 1988.

[tcp] *DDN Protocol, Transmission Control Protocol.* MIL-STD 1778.

[Tes85] L Tesler. *An introduction to MacApp 0.1.* Apple Computer Inc., Cupertino, Calif., Feb.14, 1985.

[TFC+88] Robert H. Thomas, Harry C. Forsdick, Terrence R. Crowley, Richard W. Schaaf, Raymond S. Tomlinson, Virginia M. Travers, and George G. Robertson. Diamond : A multimedia message system built on a distributed architecture. In Irene Greif, editor, *Computer Supported Cooperative Work : a Book of Readings*, chapter 18, pages 509–532. Morgan Kaufmann Publishers, Inc., 1988.

[Tic87] W.T. Tichy. What can software engineers learn from artificial intelligence ? *IEEE Computer*, pages 43–54, November 1987.

[VM79] W.J. Van Melle. MYCIN: a knowledge-based consultation program for infectious disease diagnosis. *International Journal for man-machine studies*, 10:313–322, October 1979.

[VM81] W.J. Van Melle. *EMYCIN: a domain-independent system that aids in constructing knowledge-based consultation programs.* Pergamon-Infotech, New York, 1981.

[Wal88] J.H. Walker. Supporting document development with Concordia. *IEEE Computer*, pages 48–59, January 1988.

[Web89] B.F. Webster. *The NeXT Book.* Addison-Wesley Publishing Company, Inc., 1989.

[WEK90] A.L. Winblad, S.D. Edwards, and D.R. King. *Object-oriented software.* Addison-Wesley, 1990.

[WFA84] J.M. Wright, M.S. Fox, and D. Adam. *SRL/1.5 User's Manual.* Carnegie-Mellon University, Robotics Institute, 1984.

[Whi89] The Whitewater Group, Inc., Evansta, Illinois. *Actor language manual*, 1989.

[Wil87] R.H. Willmott. X400 progress and prospects. In *Proceedings of International Open Systems, Pinner, UK*, pages 173–182. Online Publications, 1987.

[ZW90] Chaim Ziegler and Gerald Weiss. Multimedia conferencing in local area networks. *IEEE Computer*, pages 52–61, September 1990.