Tools for Implementing Groupware: Survey and Evaluation

by

Tore Urnes and Roy Nejabi

Technical Report No. CS-94-03

1994

Department of Computer Science

York University

North York, Ontario

Canada M3J 1P3

Contents

1 Introduction

This report presents the results of an effort to evaluate a set of existing tools for developing groupware. Our evaluation approach has been to put ourselves in the position of groupware developers and actually use the tools to develop groupware. The evaluation is based on a set of criteria which helped us measure the level of support each tool provided for important development issues and activities.

The topic selected for our project was motivated by a perceived need for a more in-depth analysis of and comparison between existing groupware development tools. In addition, we both hoped to be able to do work in the area of groupware tools and felt that hands-on experience using existing tools would be valuable.

This section starts by defining what groupware is. The major part of this introductory section attempts to motivate the need for tools to develop groupware by presenting an overview of technical issues involved in groupware applications. A short survey of the groupware tools chosen as the subjects of our project follows next. This introductory section will end with a discussion of related work.

The rest of this report is organized as follows: Section 2 will describe our approach to evaluating the selected groupware tools. The criteria used throughout the evaluation will be motivated and presented. Section 3 then presents our actual evaluation results for each of the tools. Finally, section 4 will summarize the results and discuss conclusions that we have drawn from our work.

1.1 Defining Groupware

Groupware falls into a relatively new research field called computer-supported cooperative work (CSCW). Dix et al. [22, Ch. 13] explain that CSCW "is about groups of users – how to design systems to support their work *as a group* and how to understand the effect of technology on their work patterns". A major area within CSCW is the provision of computer systems to support group work. Such systems are called *groupware*.

In his book of readings [2, Intro. Part I], Baecker cites a number of definitions of the term "groupware". A good definition is attributed to Malone: "information technology used to help people work together more effectively". Some authors emphasize the point that there is more to groupware than technical issues like distribution and communication technology. Baecker attributes the following quotation to Dyeson:

> More than a way of coding or building applications, groupware is a way to define,

structure, and link applications, data and the people who use them.

To this date, two main classes of groupware have been identified: *asynchronous* and *synchronous*. The former class, consisting of for example e-mail and organizational memory systems, is clearly the most successful. However, the tools considered in this report all provide development support for synchronous groupware. Synchronous groupware is often called *desktop conferencing* applications; examples include collaborative writing/drawing/design tools, group decision support systems, and games. In this report, the term "groupware" primarily refers to synchronous groupware.

It should also be noted that we are mostly concerned with *collaboration aware* [43] groupware in this report. Collaboration aware groupware applications are "aware" that they are being used by a group of users, i.e. they contain functionality for handling issues that emerge when a group of users interact through networked computers. A class of groupware tools called *shared window systems* [30] is used to automatically generate *collaboration transparent* groupware applications from existing window-based single-user applications. Collaboration transparent applications are unaware that they are being used by multiple, simultaneous users.

1.2 Why Groupware Development is Difficult

The following overview of technical issues that are related to groupware aims at motivating the need for tools to develop groupware and at familiarizing the reader with terminology and issues appearing throughout the remainder of this report.

Compared to the task of developing applications for a single user, developing groupware is more difficult. For a start, groupware developers face all the problems experienced by single-user application developers. Due to the nature of the type of groupware applications we consider here, these problems are similar to those inherent in human-computer interface development [48]. Groupware developers must also solve additional problems. First, they must handle input from and output to multiple users plus coordination and collaboration among them. Secondly, groupware systems are inherently distributed systems posing such non-trivial problems as synchronization and network latency. Finally, the lack of "production quality" groupware tools means that developers have to settle with single-user tools (e.g. graphics toolkits), resulting in developers trying to work themselves around single-user biases of the tools.

Several authors have attempted to organize the technical issues that serve as a foundation for groupware applications. Ellis et al. [24] suggest that groupware relies on the approaches and contributions of the following four disciplines (in addition to social theory): distributed systems (operating systems and data bases), network communication (bandwidth, connectivity, and multimedia

protocols), human-computer interaction, and artificial intelligence (intelligent agents). Baecker [2, intro. part I] mentions human-computer interaction, networking and communication, operating systems and data base systems, windowing systems and environments, audio and video technology, and artificial intelligence as the areas of computer science within which groupware developers require expertise. In his dissertation, Rüdebusch [64, ch. 2] organizes groupware-specific technical issues into four levels of abstraction (in decreasing order of abstraciton level): human-computer interaction, concurrency control, distribution, and communication.

In this short overview of technologies underlying synchronous groupware, we have chosen to only use three broad categories:

- *Interaction technology.* This category includes all technologies that are involved in human-human and human-computer interaction. Typical examples are window systems, I/O devices, discrete (graphics, text) and continuous (audio, video) media, window coupling, and workspace management.
- *Distribution technology.* This is where we consider different architectures and approaches to consistency management (i.e. access and concurrency control).
- *Network technology.* This category deals with transportation of data between different hardware components. Capacity, connectivity, and protocols are issues of concern here.

1.2.1 Interaction Technology

Groupware supports collaboration by allowing computers to mediate interactions (i.e. communication) between a group of users. Bowers and Rodden [6] describe how this computer-mediated interaction (or multi-user interfaces) evolved from traditional HCI, how its role is completely different from that of HCI, and how the physical work environment has great impact on it.

Many early groupware systems where realized by using shared window systems [30] to turn existing window-based single-user applications into multi-user applications. Shared window systems often rely on server-based window systems like the X window system. In a server-based window system input and output events travel between the window server and the client applications, providing an entry point for multiplexing of output and demultiplexing of input. Kernel-based systems are more problematic because they require a replicated architecture in order to be suitable for shared window systems [11].

Groupware could benefit from changes being made to current window system technology; Lauwers and Lantz [43] suggest a number of improvements (e.g. support for migrating window state,

telepointers, and transparent windows for annotation). Freeman [26] looks at how current window systems are single-user biased and presents his view on how window systems might evolve in the future, including how device drivers might better support groupware.

Continuous, digital media like audio and video have great potential for improving the effectiveness of groupware [69]. One distinguished feature of continuous media in a groupware setting is the fact that different media streams must be synchronized. Media synchronization is the enforcement of temporal relationships [32]. Two media streams may be temporally related in thirteen different ways [74]. There have been some proposals for collaborative multimedia models. Vin et al. [73] present a hierarchical model consisting of the three layers stream, session, and conference. Temporal relationships (or constraints) are defined by setting stream attributes. SuiteSound [56][19, "Multimedia Support" sidebar] is a realization of an object-based model where multimedia is incorporated by creating flow and filter objects. Rhyne and Wolf [55] suggest that shared event histories may serve as a framework for multimedia synchronization. Hill [35] advocates that declarativeness is a desirable property of any approach to specifying mixed continuous media in groupware.

Interaction coupling is an important issue in groupware. Coupling refers to the degree that events observed by a user are the results of actions performed by other users [16]. Strict *What-You-See-Is-What-I-See* (WYSIWIS) [68] coupling is the most extreme version where every action performed by any user is immediately observable by the other users. Strict WYSIWIS is often too restrictive [67], and much work has been put into coming up with interaction infrastructures that support more customizable interaction coupling (e.g. [51, 17, 37, 3]). The dialogue independence model (i.e. logically separating the functional core from the user interface) that is the foundation of user interface management systems is often used in order to simplify flexible interaction coupling schemes.

Groupware applications normally present users with a set of windows, many of which are shared. Workspace management is the provision of facilities to take care of issues like distinguishing shared windows from private ones, identify all windows belonging to the same application, among other things [43]. Two metaphors have been proposed to ease workspace management [67]: one is based on stampsheets, the other extends the rooms metaphor.

1.2.2 Distribution Technology

Responsiveness is a major concern in groupware systems where users are geographically distributed. The response times are influenced by the choice of protocols for communication, consistency, and synchronization [47], which are in turn influenced by the choice of architecture. The simplest architecture alternative is the centralized one. It generally provides poor performance. Shared window system researchers [11, 44] have proposed to use a fully replicated architecture (i.e. reduce net-

work communication to an absolute minimum) in order to achieve minimized response times. This approach gives good performance but at the cost of considerable consistency and synchronization problems. Other researchers [28, 17, 3] suggest that a semi-replicated architecture is a good solution which manages to keep both communication, consistency, and synchronization costs at a low level. Others [58, 65] advocate that full-fledged distributed systems support should be incorporated into groupware tools. It is also interesting to note that the distributed systems community, e.g. the ISIS toolkit [5], starts to take groups into account.

Issues related to the choice of architecture are scalability and robustness. Ahuja et al. [1] compare the performance of centralized and replicated architectures. Graham and Urnes [28] report that performance measurements have shown that a semi-replicated architecture scales much better than a centralized one.

Distribution entails that multiple users are given the opportunity to interact simultaneously. This creates a potential for race conditions, i.e. attempts to make conflicting updates to shared data (i.e. potentially bringing the groupware application out of synch). Concurrency or access control are the solutions to this problem. There are two types of concurrency control approaches: pessimistic and optimistic.

Floor control is the most common pessimistic approach. It uses the notion of a token to restrict the number of simultaneous users that can perform actions. Floor control assumes that all shared data objects are "owned" by one permission granting body. Greenberg et al. [31] extends this notion in the GroupDraw program by letting the different users "own" different objects and be in charge of granting access permissions. That is, concurrency control is made more distributed and non-conflicting actions are allowed to take place simultaneously. Another distributed concurrency control algorithm is the dOPT algorithm [23]. This is an efficient algorithm for replicated architectures. It is based on state vectors that are passed between replica as actions are performed.

An optimistic concurrency control algorithm is described in [50].

1.2.3 Network Technology

Cerf [8] gives a good overview of general network communication issues. Discussions of how current network technology satisfies the communication requirements of groupware applications can be found in [47] and [58]. As digital, continuous media are beginning to be incorporated into groupware applications, networks will face tougher bandwidth requirements. Fox [25] explains what these requirements are and how compression techniques can help reduce them.

One issue that is of particular concern to groupware is network latency. Big latency means that

network feedthrough is poor, i.e. transportation delays exhibit considerable, unpredictable variance. Dix et al. [22, ch. 13] briefly explain why latency should be reduced as much as possible. The source of the problem is that many of today's data transport protocols (e.g. TCP) are focused on providing optimal reliability. To better accommodate transportation of continuous media data over wide areas, new protocols that sacrifice reliability to achieve predictable, short transmission delays are needed [45].

1.3 The Selected Tools

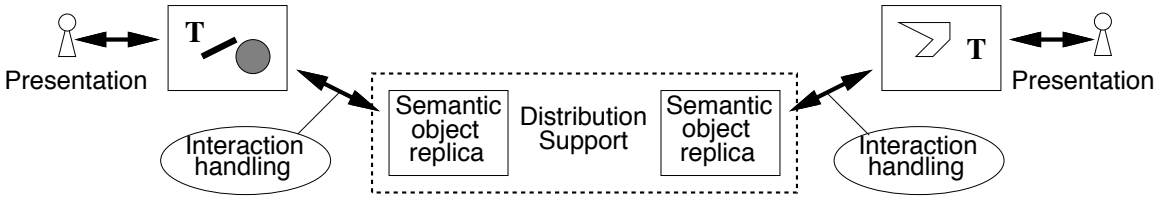
Here, we present a short survey of the groupware tools that were chosen as the subjects of our project. All of them are non-commercial systems resulting from research efforts in academia. The order of presentation is according to alphabetical ordering of tool names.

1.3.1 GroupIE

The group interaction environment (GroupIE) [64, 66, 65], is a "generic environment offering high-level development and run-time support for cooperative applications". It is a tool implemented on top of Smalltalk-80 (extended with distribution support) at the University of Karlsruhe in Germany. The development support is in the form of a library of reusable Smalltalk classes, i.e. GroupIE is object-oriented. The application domain of GroupIE is support for effective team work. GroupIE is structured according to three notions:

- Interaction. Group members work together by means of interactions. Interactions can be explicit or implicit, synchronous or asynchronous, and be either of type text or graphics (in principle also animation, audio or video). Interaction handling in GroupIE includes address resolution (according to role, sub group name, etc) and interaction adaptation (automatic, graceful degrading of interaction resource demands as dictated by available hardware).
- Coordination. In order to put restrictions on the interactions that can take place, coordination support is needed. Coordination can be basic (e.g. turn taking) or complex (relies on knowledge of task structures; checking and guiding are key factors). Coordination control takes team and task descriptions together with coordination policy constraints as input. Coordination checking is performed by intercepting interactions.
- Distribution. GroupIE is built on top of a distributed system based on a proxy model. It has support for naming and replication transparency. Both interaction handling and coordination control are implemented in a distributed way and rely on close communication with the distribution manager.

GroupIE applications have the following structure:

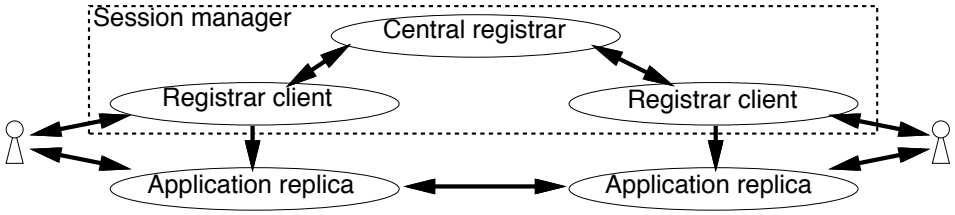


1.3.2 GroupKit

GroupKit [63, 60] is a toolkit for developing real-time work surfaces, i.e. shared visual environments. GroupKit is built on top of Tcl/Tk [52] (with Unix socket front-end extension) by researchers at the University of Calgary, Canada. Examples of groupware applications implemented by GroupKit are multi-user sketching, drawing, and brainstorming tools. Important aspects of the GroupKit toolkit are:

- Groupware applications generated by GroupKit have a replicated architecture [44]. The run-time infrastructure of GroupKit provides elaborate conference (or session) management.
- Multi-user widgets that try to address the special user interface technology needs of groupware applications.
- Flexibility through open protocols [61]. The GroupKit session manager implementation is based on open protocols which facilitates flexibility in realizing different session management policies.

The GroupKit run-time system is organized as follows:



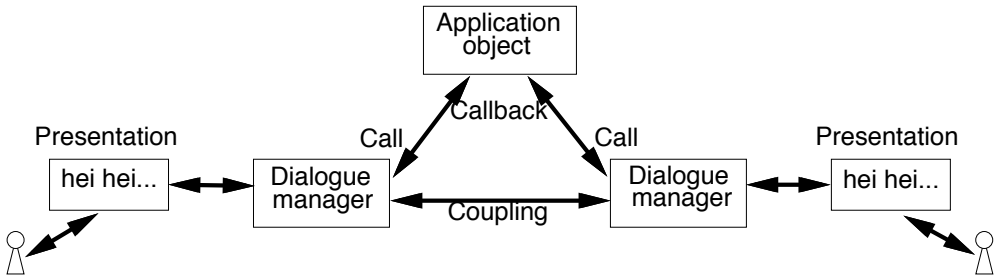
1.3.3 Suite

Researchers at Purdue University have developed a "high-level and flexible framework for supporting the construction of multi-user user interfaces" called Suite [17, 18]. The framework is based

on the generalized editing model, "which allows users to view programs as active data that can be concurrently edited by multiple users". It is structured according to the Seeheim model [29]; dialogue control is specified by (declaratively) issuing calls to dialogue managers, while application code is specified using C (programmers write callbacks which maintain consistency and take care of semantic feedback). Suite offers an object-based approach for structuring applications. The objects are heavy-weight (an object is an executable C program) and persistent. User interfaces are basically text only, i.e. the intended application domain is multi-user editing of highly structured text (no support for direct manipulation of graphical objects). Generated groupware applications have a hybrid, semi-replicated architecture. Major features of Suite include:

- Elaborate coupling model. Suite is a fairly closed system in the sense that user interface issues are specified at a very high level. This allows dialogue managers to offer generic functionality to end-users that enables them to dynamically customize the coupling of their windows with regard to other users' windows.
- A broad range of collaboration schemes can be specified with relative ease in Suite. Collaboration transparent applications can incrementally be made more collaboration aware.

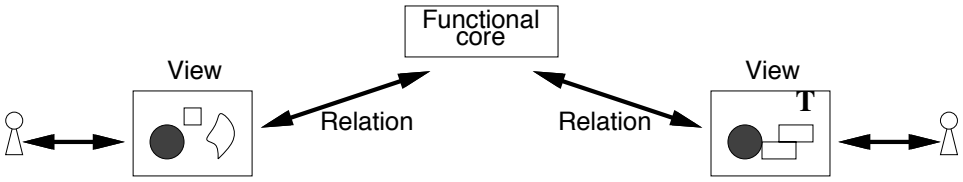
The following diagram shows how the major entities in a Suite application are related:



1.3.4 Weasel

The Weasel system [28, 72] was developed partly at Queen's University, Canada, and partly at GMD at the University of Karlsruhe in Germany. Weasel is based on a model underlying most UIMSs: the base application code is completely separated from the user interface code. In the case of Weasel, the particular instance of this model is called the relational view model. The idea is that application data structures and user interface views are linked by relations. These relations map application data onto graphical views and vice versa. A special purpose, purely functional (i.e. very high-level and declarative) language called RVL is used to specify the relations. RVL offers easy customization of views for different users and limited support for collaboration aware

constructs for realizing collaboration/coordination schemes. Applications are specified using the Turing plus language. User interfaces are generated automatically from the RVL specifications. All issues concerning distribution, communication, and synchronization are handled automatically by Weasel. Generated (groupware) applications have a semi-replicated architecture and have been proven to scale well as the number of simultaneous users increases [28]. The multi-user version of the relational view model may be sketched as follows:



1.3.5 Other tools

The four groupware development tools presented above are not the only existing ones. Here, we briefly comment on those tools which were not selected.

We had hoped to be able to obtain a copy of the Rendezvous system [38], perhaps the most impressive groupware tool developed to this date, but that could not be accomplished (Rendezvous is neither a system originating from academia nor a commercial system). Rendezvous was developed by researchers at Bellcore.

Another interesting system originating from Lancaster University, England, is the MEAD multi-user interface prototyping environment [4, sidebar 2]. Unfortunately, this could not be delivered to us within time to be considered in our project.

The ConversationBuilder [42] is a tool we decided not to consider, given our main focus on synchronous groupware. It is a tool for structuring asynchronous communication.

1.4 Related Work

To the best of our knowledge there exist no "hands-on" evaluation of recent groupware tools. A survey of groupware tools focusing on conceptual issues has been written by Dewan [16]. Here a large number of tools are categorized into being either database systems, distributed systems, message servers, shared object systems, shared window systems, multi-user toolkits, multi-user UIMSs, or multi-user user interface generators. One aspect of Dewan's evaluation is to consider a single example application (collaborative editing and testing of programming modules) and compare how this application probably could have been implemented by the tools in each category. This project

extends the work of Dewan by actually using the tools for developing groupware. Our evaluation approach is also potentially more objective because we do not use a single trial application program as the basis of our evaluation. Of the tools we have selected, GroupKit and multi-user Suite are included in Dewan's survey.

In his paper comparing the programming demands of single-user and multi-user applications, Patterson [54] proposes a list of three "dimensions" of programming complexity that are of special concern to multi-user application programmers. The three dimensions are: concurrency, abstraction, and roles. We take this work one step further by incorporating the needs of groupware programmers into a set of evaluation criteria and evaluating the subject tool relative to those criteria.

Some groupware tool authors provide short comparisons between groupware tools when they discuss related work. Hill [38] briefly compares Rendezvous to, among others, Suite, GroupKit, and Weasel. Rüdebusch considers Rendezvous, Suite, and other tools/systems in his dissertation [64].

2 Tool Evaluation Description

This section explains various aspects of the tool evaluations we have performed. The actual details concerning the evaluation results of each tool can be found in section 3.

It should be apparent from the brief introductory presentation of the tools selected for evaluation, that they are very different with respect to the kind of groupware applications they provide development support for. The tools are, in other words, biased. Therefore, we needed to pay careful attention to the evaluation criteria that provided the basis of our analysis. Specifically, we aimed at choosing criteria that were as neutral as possible in light of the different biases of the tools.

We further took the fact that the tools are biased into account, when selecting example prototype applications to implement using each of the tools. Specifically, we choose example applications from the categories that the authors of the tools had in mind when they designed the tools, e.g. for Suite, we implemented applications where users edit structured text.

The first part of this section will describe our tool evaluation approach in general terms. The remaining part will present the tool evaluation criteria.

2.1 Evaluation Design

We have performed the tool evaluation primarily by rating each tool with respect to a set of evaluation criteria. Our basis for rating the tools is obtained partly from reading existing documentation and publications about the tools and partly from experience gained after implementing simple prototypical groupware applications through tool usage.

Our evaluation approach made us make two choices:

1. What are the criteria relative to which we will rate the tools.
2. What example applications are we going to implement.

We initially came up with a large number of evaluation criteria, but chose a subset of eleven to be used in the actual evaluation. This was mainly due to the time limitation that is always inherent in a project of this kind. All the chosen criteria are what we call *programmer-centered*, i.e. the major issue is to what extent does a tool help the groupware developer in realizing groupware applications. In other words, our evaluation approach has been to put ourselves in the position of groupware developers and try measuring (relative to the criteria) the level of support each tool provided for important development activities and issues. The chosen criteria are motivated and presented in section 2.2.

Section 2.3 briefly presents criteria that where not included in the evaluation framework.

As already indicated, we have used as a general guideline when selecting example applications to implement that we should try to stay within the application domain the tool designers had in mind when designing the tools. However, whenever we perceived a lack of support for essential functionality in a tool, attempts were made to investigate such issues further.

In addition to consider example applications developed by ourselves, we have also looked at the source code of applications developed by others using the tools in questions (often implemented by the tool authors themselves).

Even though we did not implemented any groupware applications of what could be considered large size, we still feel our knowledge basis was sufficient for performing the evaluation presented in this report.

2.2 Evaluation Criteria

Before starting to consider specific criteria, we have to get an idea of what the needs of groupware developers are. These needs are typically reflected in the stated goals of existing groupware development tools. We therefore start by considering such goals.

2.2.1 Goals of Groupware tools

In [35], Hill considers the needs of programmers building user interfaces for what he calls Multi-User Multi-Media Synchronous (MUMMS) applications (a synchronous groupware application is a special case of a MUMMS application, i.e. a MUMMS application aimed at supporting real-time group interaction). A general issue considered by Hill is the programming model underlying tools for developing MUMMS applications. Properties of the programming model will inevitably need to become apparent during the course of development and it is therefore an important goal that the model is well-adapted to the implementation tasks needed to realize the target applications. Another general issue of importance is "quick and easy change" of implementations. Having small changes to the interface of a MUMMS application resulting in small changes to the implementation, is a goal. Hill further discusses a large number of "requirements" for development support for MUMMS applications. These are the principles that have been considered necessary or desirable to follow in order to achieve the goals of the Rendezvous system [38].

In section 1.3.3 we briefly introduced the Suite framework. Based on experience from working on the Suite framework, Dewan [14] proposes a long list of principles for developing multi-user user

interface development environments (UIDEs). Of more interest to us here are the "four fundamental goals of a UIDE" that he states in order to motivate his principles:

1. Domain-independence. One should support the development of general applications and not only a particular domain such as conferencing tools.
2. Automation. Low-level detail should be handled automatically and should not need to be specified explicitly.
3. Flexibility. Applications should be easy to tailor to different users.
4. Iterative design. It should be possible to develop multi-user user interfaces incrementally.

Roseman and Greenberg [60] also propose design "requirements" for groupware tools. They are the designers behind GroupKit, which was briefly described in section 1.3.2. An issue advocated by Roseman and Greenberg is that groupware should be personalizable [59] in order to help getting users to accept groupware. Though this is not stated explicitly, it is apparent that supporting personalizable groupware is a goal for groupware tools.

In addition to the above mentioned goals, some work has also been done in order to investigate what programming complexities are of special concern to groupware developers [54]. As we already mentioned in section 1.4, these complexities are perceived to go along three dimensions: concurrency, abstraction, and roles. Concurrency is needed to ensure that users are not preempted from having continuous access to an application. Abstraction means the logical separation of the code of a groupware application into the functional core and the user interface part. Development of synchronous groupware is thought to benefit from abstraction. Users take on (sometimes several different) roles in groupware settings. Access rights are often dictated by the roles of the different users. Evidently, groupware tools should strive to aid programmers in tackling such complexities.

We also found three surveys on groupware design issues [24, 51, 15] helpful when trying to get an impression of groupware developers' needs.

Finally, some results from a closely related field: Researchers working on developing tools for single-user user interfaces suggest that the following are desirable features of programming languages for implementing such systems [40]:

- Efficient runtime execution.
- Fast translation or compilation.
- Portability to, and availability on, a wide range of platforms.

- Facilities to support reuse.
- Strong typing (and other mechanisms for early error detection and prevention).

2.2.2 The Selected Criteria

We are now in a position to motivate and present the criteria relative to which we will compare the groupware tools we have selected for our investigation. The criteria have been loosely organized into four categories:

Rapid prototyping

Due to the many technical, psychological, and sociological issues that have to be taken into account when developing groupware (e.g. [24, 57]), groupware development is a highly experimental process. To aid this experimental process, support for rapid prototyping is needed. The idea is that rapid prototyping will allow groupware developers to perform usability evaluations by deploying working prototypes at early stages, and hence provide a basis for making early critical decisions about the application being developed. Based on the results of experimentation, a new iteration of the cycle of prototyping, deployment, and evaluation may be performed [9].

We have identified four criteria that allow us to gauge the extent to which groupware tools support rapid prototyping.

1. Iterative development. This criterion is central to rapid prototyping. We are interested in how well a tool supports making (major) changes to a prototype version of an application. Such major changes will typically involve iterating the stages of the design process [22, chapter 5].
2. Incremental development. We want to investigate how convenient it is to implement just a subset of an application and still be able to experiment with the partial implementation. Also, it is often the case that groupware developers need to make small changes to applications, e.g. some adjustment to the appearance of a user interface view. It is important that small changes to a groupware application result in small changes to the implementation [35].
3. Reuse. Reusing existing code can speed up the development process. We want to investigate the support for reuse a tool provides.
4. Reuse of single-user code. All the subject groupware tools are based on tools for supporting development of single-user applications with interactive user interfaces. Therefore, it is interesting to measure the support for easy inclusion of existing single-user code in groupware applications.

Underlying design paradigm

According to Hill [35], properties of the conceptual programming model of a groupware tool will inevitably become apparent during the course of development. It is therefore important that the programming model is well-adapted to the implementation task. This observation leads to in the following criterion:

5. Matching between the conceptual programming model and implementation task.

Another issue related to the underlying design paradigm of groupware tools is what kind of abstractions are provided by the programming language(s) required/offered by a tool. The abstractions provided by a programming language are intended to hide the low level details of, for example, the underlying hardware. Groupware developers have to tackle many low level issues dealing with replication and consistency maintenance [44], concurrency control [23], and managing devices involved in human-computer and human-human interaction. An important question is how well suited the programming language abstractions are for this type of low level issue:

6. Matching between language abstractions and groupware related low level issues.

Language issues

The criteria related to the programming language(s) a tool offers are:

7. Easy to learn. This criterion reflects how easy it was to learn to use a tool. Documentation is an important aspect here.

8. Declarativeness. Generally, it is desirable to have as high a level of specification as possible. In particular, we want the specification language(s) to be as declarative as possible so that we only have to worry about *what* we want and on so much *how* to achieve it.

9. Expressiveness. Normally, there is a trade-off between declarativeness and expressiveness. This criterion tries to measure whether too much expressiveness has been sacrificed in order to achieve high-level programming.

Session

Olson et al. define a session as "a period of time when two or more members of a group are working together synchronously" [51]. By session management, we mean the activities a groupware application has to perform when creating a session, allowing participants to enter and leave a

session, and tearing down a session. Some researchers advocate that session management should be an independent entity, normally provided by the groupware tool [60, 53]. In other words, groupware developers should not have to worry about session management. This leads to our next criterion:

10. Session management support.

Another important issue somewhat related to the session is the overall performance of the generated groupware applications. Even though this issue is not directly related to development support, we have chosen to include it as a criterion in our evaluation. One should, however, keep in mind that we only want to "measure" in a very subjective fashion what we think about the performance, i.e. we will not perform actual timing measurements.

11. What is the perceived performance of the generated groupware applications.

Note that the criteria in the rapid prototyping and language issues categories focus mainly on more general tool issues. The underlying design paradigm and session criteria attempt to be more groupware specific. In particular, the three dimensions of programming complexity of special concern to groupware developers discussed by Patterson [54] are covered by the two underlying design paradigm criteria.

2.3 Other Criteria

We feel that the eleven criteria we motivated and presented in the previous section provide us with a proper framework to rate the development support offered by each of the subject groupware tools. The chosen criteria are not the only possible ones, however. Here, we take a look at other criteria that where considered but left out because of time restrictions.

Two criteria which can be used to judge the quality of the generated groupware applications are *scalability* and *robustness* [22, ch. 13, pp 461–466]. The scalability criterion measures the rate at which overall performace deteriorates as either the number of interacting users increases or the level of activity increases. It should be noted that synchronous groupware applications will typically have a small number of simultaneous users (except, perhaps, for groupware that supports electronic meeting and decision rooms). Due to the nature of synchronous groupware applications (one user's actions will almost always trigger a series of corresponding actions in the displays of the other users), even a very small number of simultaneous users can pose serious performance problems [36].

It is generally more difficult to develop robust groupware applications as compared to developing robust single-user applications. The main reasons being the larger number of (often different)

hardware and software components involved, the more complex algorithms that are needed for many tasks, and the higher level of non-determinism (i.e. unforseen sequences of events) that is a consequence of having a large number of simultaneous interactions taking place. A robustness criterion would attempt to establish a perceived "crash" frequency, the extent to which "crashes" are handled gracefully, whether localized faults have localized effects, what possibilities there are for recovering from error conditions, and possibly other aspects.

The following criteria would have fit under the underlying design paradigm category above.

What support a tool offers for developing personalizable groupware, i.e. *flexibility*, is an important criterion. One of the defining properties of groupware, and also what makes groupware powerful [51], is the ability to tailor, in software, groupware to the needs of the different users. Roseman and Greenberg [61] present a number of arguments supporting the need for flexible groupware.

Another notion that is central to groupware is *coupling* [15] or linking [51] as it is also called. Groupware tools should allow programmers to easily specify how different entities should be coupled. A familiar example of tight coupling is WYSIWIS interaction, i.e. the user interface object are kept identical accross the displays of all users. Note that the ability to specify different degrees of less tight coupling is basically a prerequisite for having flexible groupware.

3 Evaluation Results

This section presents our evaluation results. We have chosen to treat each tool individually (another possibility would have been to organize the section according to the evaluation criteria). Section 4, the conclusion section, will attempt to view our results in a broader context and make comparisons between the tools.

Within the context of each of the tools being subject to our investigation, things are organized as follows: First, the brief survey in the introductory section is augmented with a more thorough presentation of issues that are novel or important about the tool in question. Then, the example applications that were developed using the tool will be presented. Next we will describe our evaluation by simply going through the criteria and give our rating. Finally, strong and weak points are summarized.

The order of presentation is the same as that used for the tool survey in section 1.3

As described in section 2.1, our evaluation has been performed by rating each tool relative to a set of evaluation criteria. A rating is given as one of the following marks:

- none. This means that a tool either does not support or performs very poorly with respect to a criterion.
- ok. The tool provides some support within the context of the criterion, but this support is minimal.
- good. We found that the criterion is supported well by the tool.
- very good. The tool made extra efforts to provide support for the criterion.

The evaluation results are summarized in figure 1.

3.1 GroupIE

GroupIE (Group Interaction Environment) [64] has been developed at the University of Karlsruhe, Germany. The main motivation behind GroupIE was to demonstrate the usefulness of a conceptual model for supporting teamwork in a distributed setting. That is, one might say that GroupIE provides development support for groupware applications that fall within a generic model of distributed teamwork.

We will now present the teamwork model that GroupIE supports.

	GroupIE	Groupkit	Suite	Weasel
Rapid Prototyping				
Iterative dev.	xx	x	x	xx
Incremental dev.	xx	xx	x	x
General reuse	xx	x	x	x
Single–user reuse	o	x	xxx	xx
Design Paradigm				
Task/model match	xx	x	xx	xxx
Task/lang. match	xx	x	x	xx
Language Issues				
Easy to learn	x	xxx	x	x
Declarativeness	xx	o	x	xx
Expressiveness	xx	xx	xx	x
Session				
Session Management	o	xxx	x	o
Performance	x	xx	xx	x

o = none x = ok xx = good xxx = very good

Figure 1: Summary of groupware tool evaluations

3.1.1 A Generic Model for Distributed Teamwork

Most groupware systems only provide *basic coordination*. The purpose of basic coordination is to keep the groupware application in question in a consistent state, typically through access/concurrency control. On the other hand, *complex coordination* aims at enhancing the effectiveness of collaborations. Practically all groupware systems that realize complex coordination are asynchronous and provide text as only interaction medium. One of the major goals of the work behind the teamwork model that is to be presented is to look at complex coordination in synchronous, multi-medial groupware applications.

The fundamental idea behind the teamwork model is that a set of users form a team. Team members can interact with each other in arbitrary ways through a shared work surface. This is called teamwork and its purpose is to perform some task. Due to the nature of teamwork, it can sometimes be executed more effectively if the team member interactions are coordinated with respect to the task at hand.

In order to specify a formal, generic model of teamwork in a distributed setting, precise models of team member interactions, team structures, task structures, and coordinations are needed. We will only present these models informally here. The full formal treatment of these issues can be found in [64, chapter 3].

The *interaction model* has an interaction as the fundamental entity. Interactions are actions performed by team members in the context of teamwork. The interaction model is object-oriented. This means that actions are actually operations performed on objects. In other words, team members interact by manipulating shared objects. This is only part of the picture, though. Every interaction must satisfy properties dictated by an *interaction context* that is associated with it. The interaction context specifies three properties or attributes of an interaction:

- *Visibility.* Who are the team members that should observe the interaction taking place.
- *Synchronism.* For each of the team members to whom the interaction is visible, there is kept a private copy of the synchronism attribute. The synchronism attribute specifies time constraints on the delay between initiating an interaction and the time it takes before other team members observe the interaction. It also specifies how often other team members should receive "updates" on interactions that stretch out in time. Finally, the synchronism attribute also specifies how an interaction should be mapped to other team members. Possible mapping approaches are: syntactical (i.e. communicate everything "as is"), semantical (i.e. only communicate the "result" of the interaction), and descriptional (i.e. only communicate that the interaction is taking place, nothing else).

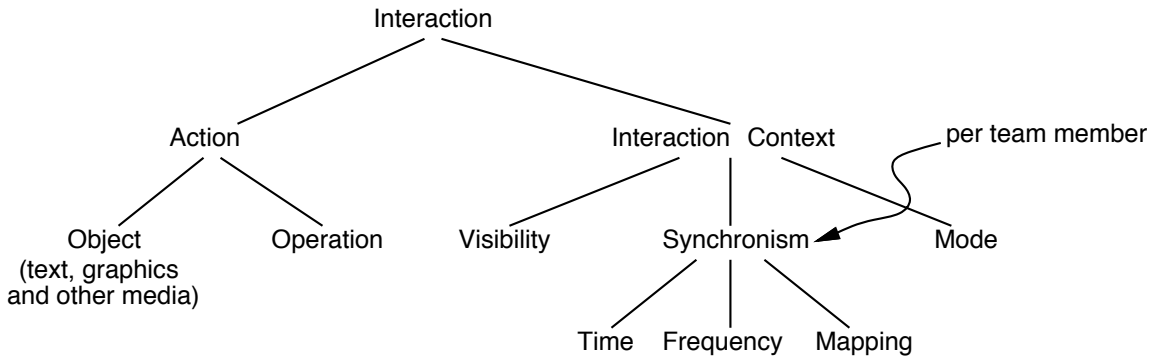


Figure 2: The interaction model upon which the teamwork model of GroupIE rests.

- *Mode.* The mode of an interaction is either implicit or explicit. Briefly, implicit interaction means that the important thing is to make team members aware of the operations that are taking place. Explicit interaction, on the other hand, focuses on bringing the object that is the subject of the interaction to the attention of team members. Roughly, implicit interaction is more work oriented, while explicit interaction is more conversation oriented.

Figure 2 summarizes the interaction model. The objects in the interaction model are called *team objects.*

The *team model* puts forward the notion that a team is an instance of a team description. The team description reflects the structure of the team in that it specifies what are the "paths" of interaction among team members. The notion of a *placeholder* is central for obtaining team description that are as general as possible. Placeholders eliminate the need for referring to explicit team members in team descriptions. A placeholder is an instance of a team role. Team descriptions therefore identify all the paths of interaction that exist between the different team roles (or placeholders). A placeholder is either a single team member or a subteam of the team in question.

The *task model* is similar in spirit to the team model in that a task is an instance of a task description. Entities that are part of a task description are the team objects that are involved in the task, the subtasks making up the task in question and the task roles that are associated with a task. The central issue in a task description is a set of task restrictions. Task restrictions are predicates (pre- and post-conditions and invariants that actions must satisfy) that are formulated over the above mentioned entities. The task restrictions reflect the structure of the task and represent the required *basic coordination.* For example, the pre-condition of an action (i.e. specifying when interaction with a team object can take place) could be the termination of a certain subtask.

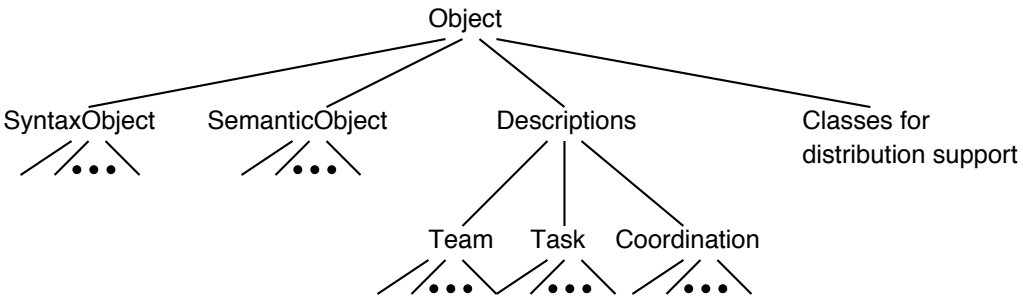


Figure 3: Overview of the current GroupIE Smalltalk class library implementation. Team objects are made up of semantic and syntactic objects. Non-basic coordination is achieved by combining objects instantiated from classes in the subtree of descriptions.

The *coordination model* integrates the team and task models that where just presented. A coordination is an instance of a coordination description. Coordination descriptions allow *complex coordinations* to be specified. Complex coordinations are either guiding or checking. Guiding coordination takes effect after an interaction is finished in order to see if suggestions can be made for following interactions. Checking coordination is activated before an interaction is allowed to commence so as to assure that any coordination policies that are peculiar to a certain team/task configuration are not violated. Another objective of the coordination description is to map team roles onto task roles.

GroupIE is one possible (distributed) instantiation of the teamwork model just presented. The current GroupIE implementation presents the programmer with a set of about fifty Smalltalk classes. The structure of the class library is depicted in figure 3.

A team object is realized as an instance of a subclass of the class `SemanticObject` which references instances (one for each team member) of subclasses of the class `SyntaxObject`. The semantic part of the team object holds the visibility and mode attributes of the interaction context, while the syntactic parts (one for each team member) hold the synchronism attributes.

The team, task, and coordination descriptions are specified using special-purpose declarative languages. These specifications are, in turn, transformed into class declarations. The generated class declarations are subclasses of the classes `Team`, `Task`, and `Coordination`.

Distribution support is provided through a proxy object extension of Smalltalk. The semantic parts of team objects are represented by proxys.

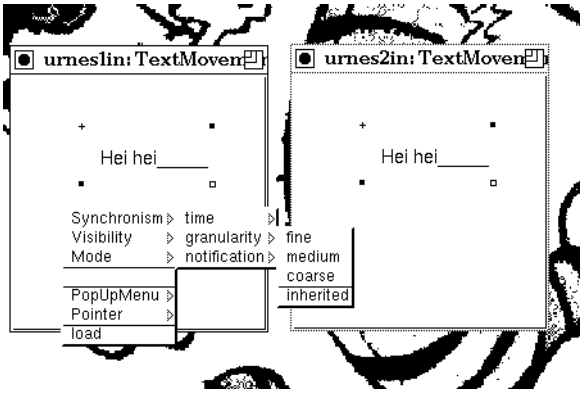


Figure 4: A simple whiteboard application. The user on the left is changing the interaction context synchronism attribute (when the user moved an object, the lag on the other user's screen was found to be too high; a coarser granularity will reduce the lag by updating the other user's views less frequently).

3.1.2 Example Applications

Due to the lack of documentation on how to actually use GroupIE to develop applications, we had to spend considerable time trying to understand how to write applications in GroupIE. Consequently, the reminder of the time that was allocated to application development in GroupIE only allowed us to implement a very simple whiteboard application.

Our example program demonstrates the interaction model in GroupIE and how easy interesting applications can be developed once one has managed to get a sufficient understanding of the tool. We would very much have liked to put the non-basic coordination facilities of GroupIE to the test but we did not find we could take the time.

Figure 4 shows a screen shot of the whiteboard application. Graphical objects (text, lines, and rectangles) can be placed on a canvas and manipulated (moved and scaled) by the users in the session. The canvas is an instance of the editor (team object) class which is made up of a replicated `SemanticEditor` object and an attached copy of a `SyntaxEditor` object for each user. The editor class has built-in support for end-user tailoring of the interaction context. That is, end-users can pop up a menu that allows them to dynamically set the values of all interaction context attributes. Note that user interface objects like pop up menus and pointers are also considered to be objects, i.e. their interaction context can also be altered. For example, a user can turn her pointer into a telepointer by simply setting the visibility attribute of the pointer interaction context.

3.1.3 Evaluation

As mentioned earlier, groupware developers program in the Smalltalk programming environment when using GroupIE. This automatically, gives good support for iterative and incremental development. Iterative development is facilitated through the component-based approach to programming that is entailed by object-orientation. Incremental development is supported by the interpreted environment that Smalltalk provides.

Most good object-oriented languages provide good support for reuse, and Smalltalk is no exception. A major motivation for using GroupIE is that it is easy to reuse the 50 Smalltalk classes that GroupIE provides. However, trying to convert an existing single-user Smalltalk application into a groupware application is not possible without having to basically rewrite the whole application from scratch. For example, if one wants to use a button in a multi-user setting, that button has to be an instance of the GroupIE class `SyntaxButton`.

In view of the criteria under the underlying design paradigm category, GroupIE gets a "good" rating. GroupIE supports multiple views by allowing multiple syntactic objects attached to a single semantic object. It is possible to declaratively specify properties of tasks and teams, e.g. which roles team members can take on and how those roles are related. Properties of different kinds of coordination are also specified declaratively. For example, a programmer can specify properties of a special kind of a hierarchical team. This specification can be converted into a Smalltalk class, creating a template for the desired team type. Instances of that type of team can be made by instantiating the class. Coordinated interaction is achieved by dynamically associating instances of teams and tasks with an instance of a coordination class. One should beware that programmers occasionally are forced to deal directly with the replicated implementation model of GroupIE.

GroupIE is hard to learn. Documentation is poor. The declarative specification of teams, tasks, and coordination gives GroupIE a "good" rating on declarativeness. We are not enthusiastic about the expressiveness provided by the GroupIE class library, but chose to give it a "good" rating with reservations.

Session management is not provided in GroupIE. This is a considerable problem because of the replicated implementation model. Machine names that can participate in a session are hard coded. The distribution support must be initialized (making connections to the machines participating in the session and initializing the object tables that hold proxys) sequentially for each user.

The performance of GroupIE applications is acceptable for prototyping, but not suitable for production use. In WYSIWIS applications, display updates may lag considerably.

3.1.4 Conclusion

GroupIE is an implementation of a generic model for distributed teamwork. We experimented with the classes that make up the interaction model and found them interesting, in particular the built-in support for having end-users customize the interaction coupling.

As mentioned, perhaps the most novel feature of GroupIE is the non-basic coordination facilities that are offered in the context of synchronous, multi-medial groupware. Unfortunately, we did not have time to investigate non-basic coordination in any great detail. However, judging from what we did learn, it looks promising.

There are, however, severe problems with the current version of the system. Performance is too poor. We suspect that the overhead imposed by the teamwork model is partly to blame; it could also be that Smalltalk slows things down significantly.

The distribution support is not as transparent and easy to use as we had hoped. In particular, better support for session management and team object creation should be provided.

Finally, a system of the complexity and size of GroupIE is totally dependent on documentation in order to be usable. The documentation coming with GroupIE is currently far from being acceptable.

3.2 Groupkit

GroupKit [63, 60] is a groupware toolkit developed at the University of Calgary, Canada. It extends an existing toolkit (Tcl/Tk [52]) for developing graphical user interfaces for single-user applications. The first version of GroupKit was based on Interviews [46]. We will only consider the current Tcl/Tk version here, even though the earlier version might have more elaborate multi-user widgets.

The interesting aspects of GroupKit are its session management support and the fact that it is a groupware toolkit, i.e. it provides programmers with a set of widgets that are tailored for use in groupware applications. We will now discuss these issues in some detail.

3.2.1 Session Management

Synchronous groupware requires people to work together by using the same application simultaneously. A *session* is the period of time that a groupware application is being used by a group of people to work together synchronously [51]. By session management, we mean the activities a groupware application has to perform when creating a session, allowing participants to enter and

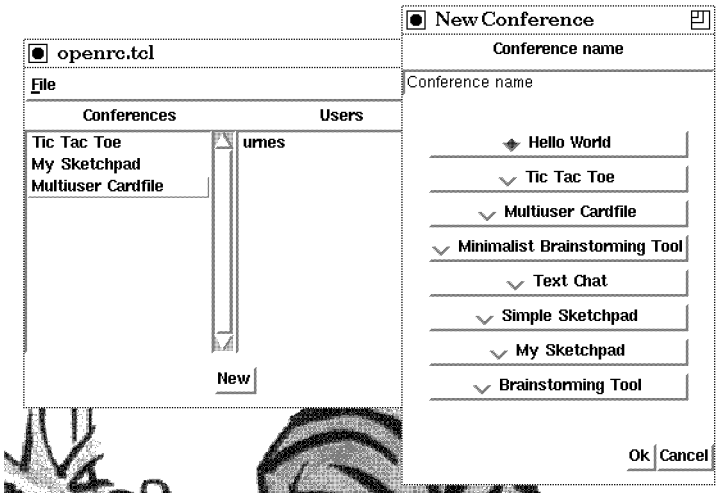


Figure 5: The user interface of a GroupKit registrar client. The `openrc.tcl` window is the main registrar client window. The "Conferences" list on the left is the list of all on-going sessions. By selecting a session (the "Multiuser Cardfile" session is selected in this example) the list of participating users in that session is shown in the "Users" list on the right. The "New" button can be pressed to pop up the list of possible sessions that can be created from this registrar client. This list is shown in the `New Conference` window on the right hand side.

leave a session, and tearing down a session.

GroupKit provides stand-alone session management that can be used by all groupware applications that are developed using the tool. The session management support consists of two entities: a *central registrar* and a set of *registrar clients*.

The central registrar is a background process that runs on a computer that is known and accessible to all users. It maintains the complete lists of all the ongoing sessions and all the users registered for each of the sessions.

A registrar client typically presents the user with three types of lists:

1. A list of on-going sessions.
2. A list of participating users for each on-going session.
3. A list of possible new sessions that may be created.

The registrar client that comes with GroupKit offers all this information through a nice graphical user interface (see figure 5).

The session management support in GroupKit is slightly more novel than what has been described so far. The session management support has been implemented by using *open protocols* [61]. This

means that the central registrar (called the controlled object in open protocol terminology) has been programmed to be as general as possible. That is, one has gone to great lengths in order to make the central registrar policy independent. The central registrar is a server in that it maintains a set of data (session and user lists) but it departs from more traditional servers by being policy independent. A key factor is the protocol that dictates what external requests the central registrar obeys. A controlled object will typically be extreme with respect to the wide range of requests it will obey.

Now, any registrar client (the controller object in the terminology of open protocols) will typically only use a small subset of the requests that the central registrar makes available. The important thing is that it is easy to write many different kinds of registrar clients without ever having to change the central registrar (it does not't even have to be re-compiled). For example one could imagine wanting to have a registrar client that has super user functionality (e.g. can throw out users from a session), or a registrar client that requires users to be "sponsored" by users already in the session in order to join it. The open protocols of the session management support of GroupKit allow many such registrar clients to be used without changing the central registrar.

3.2.2 Multi-user Widgets

GroupKit can be considered to be a user interface toolkit providing high-level abstractions, called widgets, for simplifying the realization of graphical, interactive groupware applications. The widgets that are available in the current version of GroupKit are primarily the Tk widgets, i.e. single-user widgets like buttons, scrollbars, menus, and canvases. Two multi-user widgets are offered by GroupKit [62] in addition to those provided by Tk[1].

The first multi-user widget is a "vertical remote scrollbar, a device that displays the position of a remote user's scrollbar" [62]. Each user is represented by a color in GroupKit and a remote user's scrollbar position is represented by a colored position indicator. The remote scrollbar also has a pop up menu that names of the users of the scrollbar.

The second multi-user widget is a remote cursor widget. A remote cursor is simply a label with a user's name appearing in remote users' canvas widgets.

[1]It should be noted that the InterViews version of GroupKit offers two different widgets (or glyphs, as similar objects are called in InterViews) than the Tcl/Tk version, namely a sketchpad overlay and a cursor overlay.

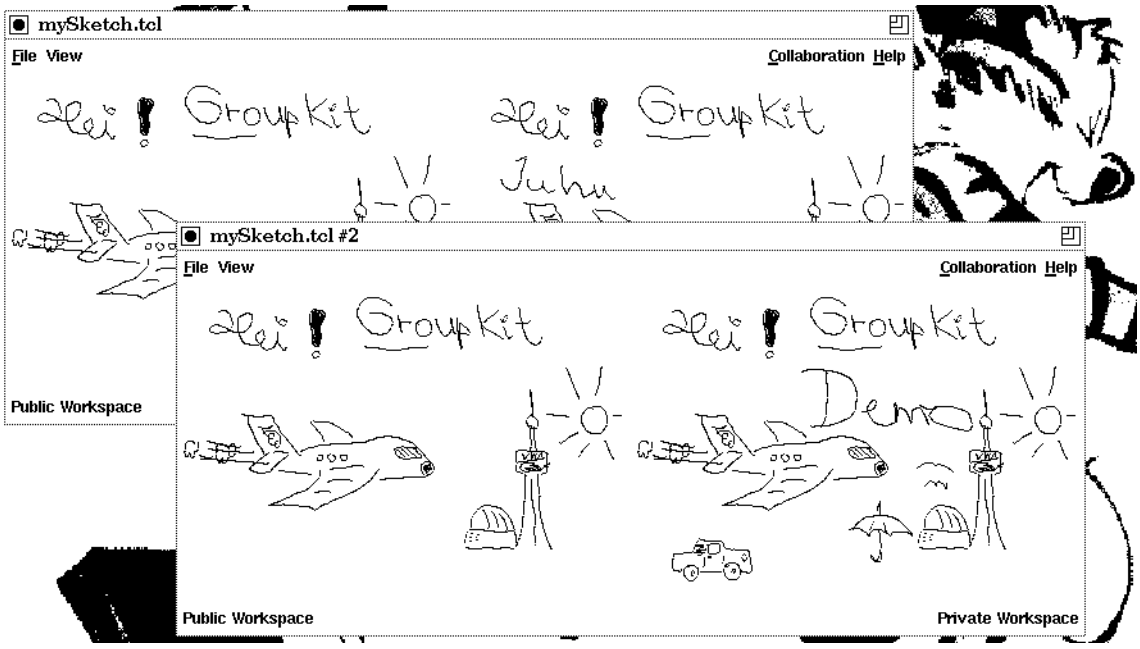


Figure 6: A simple sketching program. Each user has both a public and private view (or workspace). Everything appearing in the public view also appears in the private view (this makes it easy to privately annotate whatever appears in the public view). Any sketching taking place in a public view will automatically be broadcasted to all public and private views. Sketching made in a private view will not be broadcasted unless requested explicitly by the "owner" of the private view. Private sketchings have a different color than public ones. Deleting in a public view will perform corresponding deletions in all views; deleting in a private view will only affect the *private* sketchings in the private view in question. (Regrettably, the dividing line between the private and public views is missing in this screen dump; it had a too low gray scaling factor.)

3.2.3 Example Applications

The first example application we wrote for GroupKit was an extended version of the very simple sketching programs that comes with the ftp-able GroupKit distribution. We added a private drawing area and implemented erasing functionality. Figure 6 depicts a two-user session.

The sketching program was very easy to implement. The item tagging facility that is offered through the structured graphics functionality of the Tk widget took care of maintaining per-user data for each user's private workspace.

Our second example is a multi-user cardfile browser. A set of cards, each holding a person's name, address, and phone number information, is kept in a list. A slider consisting of the letters of the alphabet is used for selecting a card. See figure 7.

This example also attempts to investigate how customized views may be specified in GroupKit. We first implemented a WYSIWIS version of the cardfile browser, which was fairly straight forward.

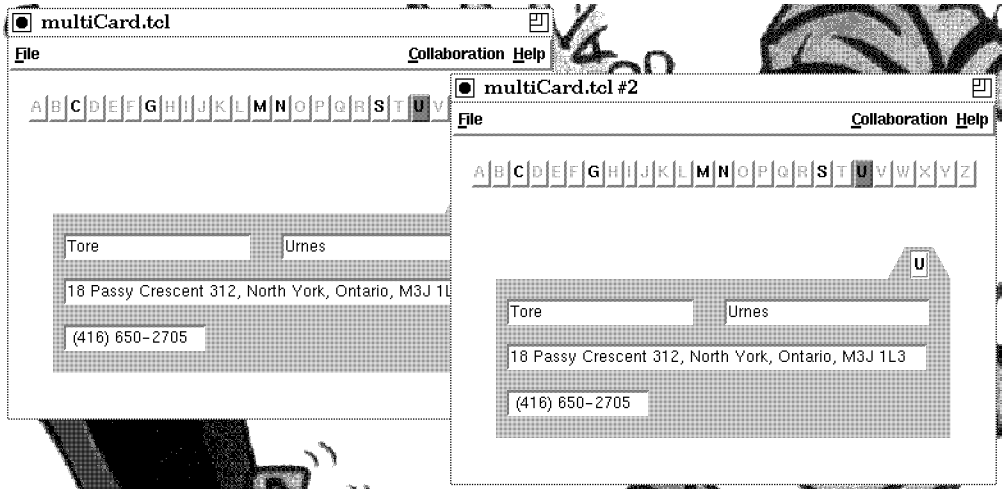


Figure 7: A cardfile browser example. A view consists of a slider and a card. The slider is used to select cards by clicking on a slider letter (the card having a last name beginning with the slider letter will be shown). The fields of a card can be edited. If the last name of a card is changed, then the slider will be updated correspondingly (i.e. it will always reflect the currently available cards by highlighting certain letters).

Next, we wanted to allow independent browsing while also keeping user interfaces consistent in response to any changes that were made to the cardfile data base. This turned out to more difficult than we had imagined. Whenever a last name is changed, the slider must be brought up to date across all users. In addition, for those users that are currently viewing the changed card, the card view must also be brought up to date. In addition to keeping views up to date, the cardfile data base itself had to be maintained across all replica and we also had to write code to bring any latecomers up to date. It should be evident that implementing a multi-user version of the cardfile browser was much more difficult than implementing a single-user version.

3.2.4 Evaluation

GroupKit programmers typically spend most of their time writing callback procedures in the Tcl scripting language. The callback procedures are activated by either Tk or GroupKit widgets in response to user interaction. Groupware applications generated by GroupKit have a replicated architecture. Programmers keep data consistent across the different replica by multicasting all update commands performed on shared data.

Support for iterative development in GroupKit is given an "ok" rating because of the lack of constructs for structuring the code. Programmers often have to resort to using global data and rely on side-effects to accomplish many tasks. This makes it difficult to make major changes to the code.

GroupKit programs are interpreted. This ensures that small refinements can be made very quickly to an application. The widgets provided by Tk and GroupKit provide sensible default values for most attributes, hence reducing the effort of programming user interfaces. These positive aspects of GroupKit are reflected in a "good" rating on incremental development.

General reuse is made more difficult by the lack of facilities for component-based programming and the reliance on global data. No special efforts have been made in order to ease reuse of existing single-user Tcl/Tk applications.

In GroupKit, the conceptual programming model is identical to the replicated implementation model. This means that programmers get the feeling that they are programming a distributed system. One must constantly worry about keeping shared data consistent, and bringing latecomers up to date is often non-trivial. Though the policy dependent widgets in Tk and GroupKit relieve the programmer from having to think about low level user interface details, there is a lack of support for specifying issues related to sharing of data, such as consistency maintenance and access control. We have therefore only given an "ok" rating on the underlying design paradigm criteria.

When it comes to ease of learning, GroupKit is in a class of its own. The documentation is excellent. Tcl/Tk is a simple language which is fun to learn. Support is also available through a mailing list and an internet news group on Tcl. There is no question about the validity of a "very good" rating.

Tcl/Tk is an imperative language, resulting in a "none" rating on declarativeness. The expressiveness of the language is "good". An interesting feature is the possibility of extending the GroupKit interpreter with arbitrary C code libraries that can be easily accessed by creating new Tcl commands.

GroupKit has very good session management. It is easy to get an overview of existing session, to join an ongoing session, or to create a new session.

GroupKit applications run quickly, despite the fact that they are based on an interpretive language. We give a "good" performance rating.

3.2.5 Conclusion

GroupKit is the only tool among the ones evaluated in this report that is robust, expressive, properly documented, and offers good session management. It is therefore the only tool that is suitable for actually building more than toy applications.

Unfortunately, it is difficult to use GroupKit to implement applications that have non-trivial coupling needs; i.e. coupling that is neither strict WYSIWIS nor completely independent. We suspect

that this problem can only be rectified by changing the underlying design paradigm of the tool. We would in particular have liked to see a more centralized conceptual programming model. Since GroupKit is so heavily based on Tcl/Tk, it is unclear how this might be accomplished, though.

3.3 Suite

This section gives a detailed presentation of *Suite* (a System of Uniform Interactive Type-directed Editors) [18, 17, 21, 12]. Work on Suite started in 1987 at Purdue University. An overview of the project is given in [20]; it goes through the history of the project and contains an annotated bibliography recounting all publications related to Suite.

Suite provides development support for groupware applications where users interact through a shared work surface by editing structured text. That is, groupware applications are viewed as generalized, multi-user data structure editors. Issues related to the use of generalized multi-user editing as interaction model in a groupware context are discussed in [15].

We will now look at two distinctive features of Suite: user interface coupling and automatic deployment of single-user Suite code in a multi-user setting.

3.3.1 Flexible Coupling

A novel feature of Suite is that the user interfaces of different users can be easily coupled in a flexible, dynamic manner. Coupling refers to the kind of sharing that is needed among windows displaying a shared workspace. The most common form of coupling is WYSIWIS coupling [68], found in many early groupware systems. Experimentation with groupware applications has verified the need for more flexible coupling [67]. Before we explain coupling support in Suite in more detail, we will take a closer look at the model that underlies Suite.

Suite is based on the Seeheim model [29] and *dialogue managers* play an important role in the system. Briefly, dialogue managers take care of user interface issues. The other major entity in the model is the functional core that we will call the *application*.

There are two major abstractions in Suite: *active values* and *interaction variables*. Active values represent the shared data structures of generated groupware applications. Interaction variables represent the editable, textual versions of those data structure items. The interaction variables reside in dialogue managers and are roughly copies of the active values with added interaction attributes. Users interact by editing (changing) the interaction variable presentations. The active values, corresponding to the changed interaction variables, and other users' interaction variables

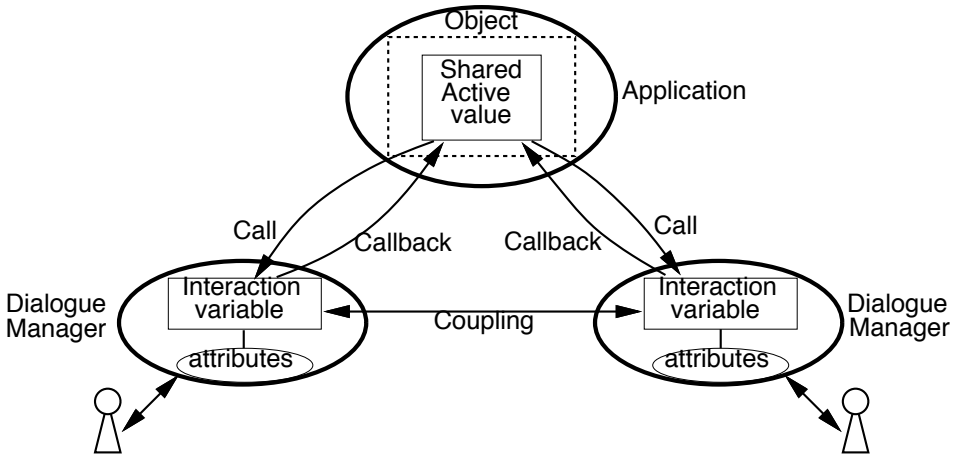


Figure 8: A Suite application consisting of one object containing one editable active value. Suite creates a set of interaction entities (interaction variables and associated attributes) for each user. Normally, one interaction entity is created for each active value in the application for each user. Interaction entities reside in dialogue managers. This figure is a simplified version of figure 3 in [17].

may be updated by committing the edited changes.

From a programmer's view, applications are object-based[2]. The objects are heavy-weight since they are basically stand-alone, executable C programs. Objects can send messages to each other (implemented on top of a remote procedure call mechanism). Within an object, active values can be specified as global C variables of (in principle) any type (but there are some restrictions in the current implementation).

What makes Suite a groupware tool is the fact that an object can have several dialogue managers "attached" to it, one for each user. Also, objects and dialogue managers do not have to execute on the same machines. Figure 8 shows how an object containing a single active value has two dialogue managers attached to it; i.e., two users may interact with it. The application C code specifies a set of *callbacks* that may be activated by dialogue managers. Callbacks are needed to keep data consistent and to provide semantic feedback to users. The application instructs dialogue managers about properties of the desired user interface by issuing *calls*.

From an architectural view point, one should note that there is always only one copy of each object and each user will normally have attached no more than one dialogue manager to any object. An application can consist of an arbitrary number of objects, each being a separate heavy-weight process. One might view the dialogue managers as being replica of a shared dialogue manager. We

[2]Suite is not object-oriented since it offers no code sharing mechanisms like inheritance or delegation.

call the Suite architecture semi-replicated.

We are now in a position to describe the coupling support in Suite.

One interesting feature of the coupling model in Suite is the fact that both application programmers and end-users can specify how each user's user interface should be coupled with those of other users.

Programmers (explicitly) specify coupling by coding callbacks – a fairly traditional and ad hoc approach, i.e. input "events" are sent to some agent (callback function) that determines which windows to bring up to date.

The provision for end-user specification of coupling is more interesting. The idea is to identify a set of *coupling attributes* and associate them with interaction variables in dialogue managers. When end-users edit the presentation of interaction variables, dialogue managers broadcast the changes that are made to the other dialogue managers according to the current setting of the coupling attributes. This activity is depicted by the "Coupling" arrow in figure 8. Each dialogue manager provides the user with a standard *coupling window* that provides facilities for changing the coupling attributes dynamically. Figure 9 shows an example.

As can be seen from figure 9, coupling attributes can be divided into two types: those specifying *what* to couple and those specifying *when* coupled data items should be broadcast to other dialogue managers (or when other dialogue managers should poll for broadcasted data). The twelve buttons in the middle of the coupling window are used to specify what should be coupled. The lower part of the coupling window offers provisions for setting the eight transmit/listen coupling attributes.

Setting the value of a coupling attribute involves specifying the value of the attribute in question (e.g. "True" for the `ValueCoupled` attribute), the name of the text fields in the user interface window the setting applies to, and the set of other dialogue managers that should receive notification of changes. The upper part of the coupling window is used for specifying these latter two parameters.

The coupling windows supplied with Suite dialogue managers offer many interesting opportunities for dynamically changing the coupling among windows in groupware applications generated by Suite. A video demonstration of the Suite coupling windows is given in [13].

3.3.2 Single-User Code Reuse

One of the design goals of Suite was to allow programs written for an earlier single-user version of Suite to be executed in a multi-user setting under the current (multi-user) Suite system. Shared window systems [30] already allow existing window-based single-user applications to be used by

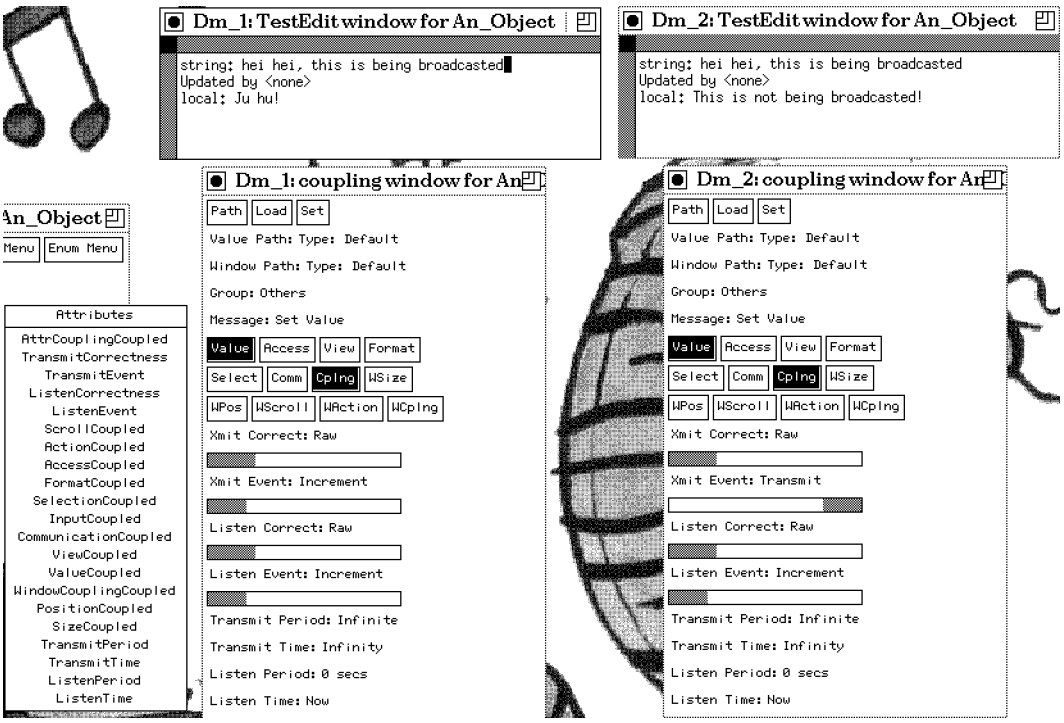


Figure 9: This example shows a simple example application (`An Object`) with two dialogue managers attached (`Dm 1` and `Dm 2`) to it. The "coupling window" provided by each dialogue manager is shown. On the left is a popup menu listing all the coupling attributes that are currently supported in Suite (the popup menu is part of the standard "attribute window" of Suite dialogue managers). This simple application allows users to edit a 'string'. If a user commits edited changes, then a message says who did the commit. The string 'local' can be edited privately. The user having dialogue manager `Dm 1` has changed the standard setting of the coupling attributes (the `TransmitEvent` (Xmit Event) attribute has been set to `increment`, meaning that every key stroke is broadcasted to the other dialogue managers).

multiple simultaneous users. One property of groupware applications generated by shared window systems is that they exhibit strict WYSIWIS coupling. This is a consequence of the fact that such applications are completely collaboration transparent (the applications are not aware that they are executed in a multi-user setting).

The fact that Suite dialogue managers are in effect collaboration aware (e.g. offering generic facilities to allow end-users to tailor the coupling between different users' user interfaces) takes Suite a step beyond shared window systems in that collaboration transparent applications can be made more collaboration aware without changing the actual code. Note that this approach has many similarities with what is offered in the MMConf [11] system, where limited, generic collaboration awareness functionality can be linked into existing single-user applications. However,

whereas the collaboration aware primitives in MMConf are primarily intended to help overcome problems with using replicated architectures in shared window systems, Suite is targeted at making collaboration transparent applications more collaboration aware with as little cost as possible. Note that both Suite and MMConf require single-user applications to be re-compiled. Shared window systems do not have this drawback.

As with shared window systems, rules must be established in order to define "new" semantics of single-user operations in a multi-user setting. In shared window systems, the end-users themselves have to enforce these rules, and hence the strict floor control protocols that identify shared window system groupware. This floor control overhead is not imposed on end-users when executing single-user Suite applications in a multi-user setting. In stead, Suite enforces a default set of rules that allows simultaneous interactions to take place in a proper manner.

The first rules are concerned with *access control*. For callbacks, things are simple, i.e. everybody has equal access to update application objects. For calls, things are more complicated. The different types of calls are categorized into five categories: response, request, retrieval, action, and update. All the different call categories have different rules regarding whether one (response), any (retrieval) or all (request, action, update) dialogue managers are targets for the call in question.

The second subset of rules is concerned with *coupling*. When a user commits a change made to an interaction variable, all dialogue managers get their corresponding interaction variables updated. Interaction attribute changes are not broadcasted.

The *concurrency control* rule is that once a user starts editing an interaction variable, all other corresponding interaction variables at other dialogue managers are locked.

A final interesting aspect of incorporating collaboration transparent (i.e. single-user) code into Suite is the provision of collaboration aware versions of most calls occurring in single-user Suite programs. This allows programmers to make the code itself more collaboration aware in an incremental fashion. For example, one can direct calls to specific groups of users.

3.3.3 Example Applications

There are quite a number of example programs coming with the Suite distribution. When executing the example programs, it struck us that all of them were basically WYSIWIS applications in terms of the contents of the views. That is, different users saw exactly the same text fields in their views. We therefore decided to investigate how highly customizable (relaxing WYSIWIS in terms of view contents) groupware applications could be specified in Suite.

Our example program was a cardfile browser of the kind that was developed using GroupKit. So,

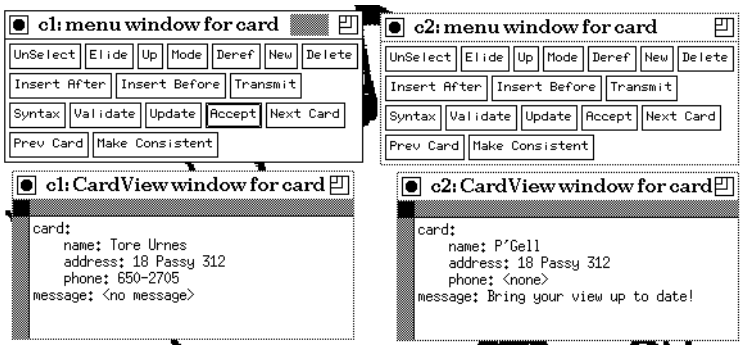


Figure 10: A simple cardfile browser. Users can simultaneously view different cards. A card is selected by pressing the "Next Card" and "Prev Card" buttons. The "Make Consistent" button is needed to solve a problem that arises when changes to a card are committed. A card consists of a name field, an address field, and a phone field. The message field tells users when they have to press the "Make Consistent" button. The two users in the example are viewing different cards. The user on the left just committed changes made to the "address" field, making the view of the user on the right inconsistent. The user on the right must now press the "Make Consistent" button.

we have a set of cards and we want different users to be able to view different cards simultaneously. The contents of each card can be changed and consistency should be maintained. Figure 10 shows the Suite version of the cardfile browser.

During the development of our example program we made some surprising discoveries. First of all, Suite has a centralized application and we therefore did not think that we had to take latecomers into account. This turned out not to be completely true. In Suite, explicit (though simple) actions have to be taken to ensure that latecomers are brought up to date in a proper manner.

Secondly, it is very difficult to properly update the views of different users when views are not WYSIWIS in terms of content. The commit command always updates *all* views regardless of whether users are viewing different data or not. Therefore, when a user changes a field in a card, the corresponding field is updated with the new value in all views even when other users are viewing different cards. The only way of handling this is to have an update callback re-setting all the users views to their correct state after a commit. In order to do that, the update callback must know which cards are currently being viewed by each user (i.e. one must maintain a per-user data structure with the current card of each user). Then it must instruct the dialogue manager of each user to redraw its view with the right card information. We use a compromise where we broadcast a message to all users saying that a commit was just performed an that the users should manually restore the correctness of their views (it is relatively simple to provide such functionality).

It is very easy to specify the content of a view as long as one is satisfied with the automatic layout. Basically, one tells the dialogue manager the type and address of a data structure entity that should be displayed. The dialogue manager goes through the data structure (there are restrictions on what data structures it can handle) and displays all (sub)entities of basic type (string or integer) it can

find in consecutive order. A card is a structure containing three strings. Two simple calls, a submit and an engage, are sufficient to display it.

We discovered an interesting way of maintaining per-user data (e.g. the current card viewed by a user). By submitting data to the dialogue manager but not engaging it, the dialogue manager does in effect serve as a per-user depository for arbitrary data. The data can be accessed in callbacks, activated by the user in question, by issuing `Dm GetView` calls. We could not find any evidence that this approach had been used before.

There is automatic support for persistent data in Suite.

3.3.4 Evaluation

Even though programmers are largely relieved from having to worry about user interface presentation and coupling issues, we have only given an "ok" rating on support for iterative and incremental development. The reason is that programmers are forced to resort to global variables for implementing semantic feedback, and the support for modularizing the code is not fine grained enough (the only kind of module or "object" is a C file, called a heavy-weight object, that is compiled into a stand-alone executable program) Also, a slow compilation process substantially hinders incremental development.

Reuse of existing single-user code is exceptional in Suite. The build-in coupling support in the dialogue managers makes it possible to develop groupware by simply developing a single-user application. Reuse in a more general sense is only "ok".

Suite is distinguished by its elaborate, well-designed coupling model. We found that having a dialogue manager taking care of all presentation and many coupling issues was a benefit. However, no work has been done to make it less cumbersome to maintain state across the editable objects, i.e. programmers are forced to manipulate global variables with ad hoc C code in order to provide semantic feedback. We give a weak "good" rating on the conceptual model and a strong "ok" rating on the language abstractions.

An "ok" rating is given to ease of learning since there is no tutorial on how to develop applications in Suite (there is some documentation for single-user Suite, but the current Suite system has changed substantially after that documentation was written). Writing make files is non-trivial in Suite. Although Suite is largely based on the imperative C language, the presence of a declarative dialogue manager interface gives Suite an "ok" rating on declarativeness. Expressiveness is "good" with considerable reservations, and we emphasize that Suite cannot be used to develop groupware based on the graphical, direct manipulation style of user interface that many users expect today.

Suite offers some session management support, but users can not get an overview of on-going sessions. Performance is "good", but keep in mind that Suite has a textual user interface.

3.3.5 Conclusion

The developers behind Suite have put much effort into allowing end-users to tailor many aspects of the interaction coupling. We suspect that the Suite coupling model might be a bit too elaborate, especially in view of the rather restricted (text-only) interactions that can currently take place.

In view of the expectations of today's end-users, the interaction styles supported by Suite (i.e. only editing of text) are clearly to restrictive.

3.4 Weasel

The Weasel system [28, 72] was developed partly at Queen's University in Canada and partly at the University of Karlsruhe in Germany[3]. In a way, the motivation behind Weasel is to demonstrate that the *relational view model* is suitable for developing graphical, highly interactive, multi-user user interfaces. The relational view model was inspired by an early version of Weasel which demonstrated how a declarative (pure functional) language, called the graphical view language or GVL, was practical for specifying algorithm animations involving arbitrary abstract data types [27, 10].

We first give a description of the relational view model and its realization in Weasel. Then, we present some example applications developed using Weasel. Finally, our evaluation is presented.

3.4.1 The Relational View Model

Weasel is an example system implementing the relational view model, and is the only system employing it to support development of multi-user user interfaces. The Trip2 system [70] is an example of an instantiation of the relational view model for supporting development of single-user user interfaces.

The principle underlying the relational view model is that of using the specification of a relation to facilitate bi-directional mapping between an abstract and a pictorial form of the same data. The abstract form of the data is typically represented as abstract data types, i.e. part of the functional core of an application (abstraction) . The pictorial form is represented as graphical views. Note that both the abstraction and the views can be manipulated, the former by the application program;

[3]Weasel was developed by T. C. N. Graham, J. R. Cordy, and the first author of this report.

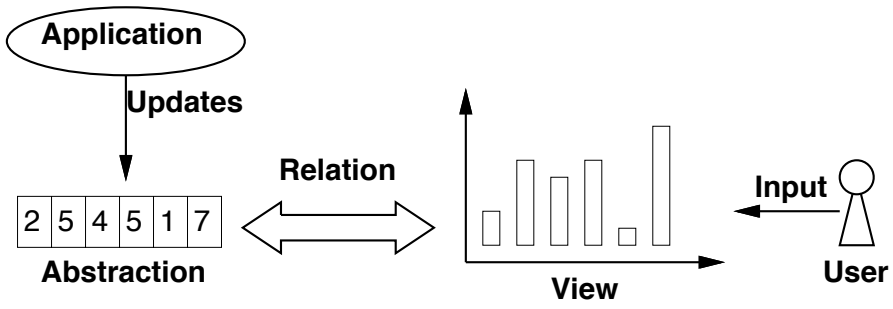


Figure 11: The relational view model, an example. A relation provides bi-directional mapping between an abstraction consisting of an array of six numbers and a view consisting of a bar graph where a bar's height reflects the value of a number. The array can be updated by an application program and a user can directly manipulate the (e.g.) height of the bars.

the latter by an end user. By continuously maintaining the relation, updates made to either the abstraction or the views can be mapped onto corresponding updates to the views or the abstraction, respectively. See figure 11 for a simple example of the relational view model.

The most interesting aspect of the relational view model is how relations are specified. In Weasel, relations are specified using the *Relational View Language* (RVL). RVL is a pure functional language [39], i.e. relations are specified as a set of side-effect free functions. RVL is based on the GVL language mentioned above. The fundamental entities of the language are graphical primitives like lines, circles, and boxes. An RVL function will always yield a picture when it is evaluated. Arbitrary pictures can easily be specified by combining functions in the usual manner of functional languages. Note that the fact that RVL is a pure functional language entails that RVL specifications are declarative (even in a strict interpretation of the term).

Specifying pictures will not bring us very far towards realizing multi-user user interfaces, however. First, we need to explain how the bi-directional mapping is accomplished. Basically, it depends on two notions: bindings and interactors.

RVL functions take parameters and a parameter can be *bound* to virtually any data structure (e.g. variables of basic types, arrays, lists, graphs, etc.) Conditional constructs and recursion provide simple and powerful support for RVL specifications to traverse any of these data structures. In other words, we can bind arbitrary data to RVL specifications and easily use those data to generate pictures. This takes care of the mapping from abstraction onto views.

When a user manipulates some part of a picture in a view we need to register this and according to the "semantics" of the picture make some update to the abstraction. For example, in figure

11, clicking on a bar might cause the corresponding number in the abstraction to be incremented. RVL provides Garnet-style *interactors* [49] for this purpose. An interactor is an agent that can be associated with a picture object of the screen; whenever that picture object is manipulated by the user it causes some semantic operation to take place. A mouse click interactor, for example, can have the semantics that some entity in the abstraction (which is bound to the picture object in question) is assigned the value true.

Remember that each part of a picture is generated by calling a function. In RVL, if a function call has an interactor attribute, then the resulting picture will be sensitive to input. Furthermore, the part of the abstraction being bound to the parameters of the called function is the target of whatever are the semantic operations of the interactor. RVL has many predefined interactors, e.g. text entry interactors, mouse click interactors, counter interactors, etc.

This explains the bi-directional mapping. However, there are still some notions that are needed in order to be able to effectively specify multi-user applications with graphical, direct manipulation user interfaces (of which desktop conferencing applications is a special case).

First of all, users expect to see familiar interaction metaphors like menus and scroll bars on the screen. These can of course be specified easily (even with customized look an feel) using RVL. Unfortunately, there is one problem: data state used exclusively for operating user interface metaphors has to be kept together with the abstraction. This is not compatible with the dialogue independence principle [33] underlying the majority of modern user interface development systems. Therefore, data related to the state of the user interface metaphors are kept separate from the abstraction in the UI state. Now, interacting with a scroll bar will update the UI state. Programmers are allowed to write simple, one-way constraints to trigger updates to the abstraction if any UI state updates warrant such actions.

Figure 12 summarizes the discussion this far. We now have a system which is practical and useful for developing single-user applications with graphical, direct manipulation user interfaces.

We are now in a position to discuss the multi-user features of Weasel and RVL. Clearly, by simply taking the single-user Weasel system and distributing the views one would get instant collaboration transparent (i.e. WYSIWIS [68]) desktop conferencing applications. Note that this would not add any new concurrency control problems since single-user Weasel was designed to manage multi-threaded dialogues [34] and the application can do computations on its own, independent of what the user does. So, we would stand apart from shared window systems [30] in that floor control would not be needed (or, indeed, would not be possible).

We will now show that it is relatively straight forward to have easy specification of collaboration aware applications in Weasel, using RVL. Collaboration awareness is desirable since it makes it

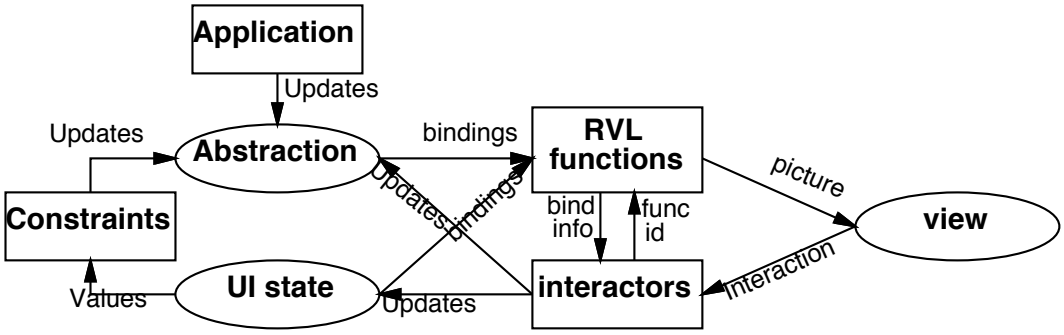


Figure 12: The single-user Weasel system. Rectangles show active entities, ovals are passive entities. Note that all the active entities creates a concurrency control problem. This is taken fully care of (automatically) by the Weasel run-time system, so programmers do not have to worry about that. Note that this is an implementation model, not a conceptual programming model.

possible to relax the pure WYSIWIS interaction style of collaboration transparent applications [67]. In Weasel, making an applications collaboration aware is called customization with respect to the different users.

We define the abstraction and the UI state as the *context* of the application. The notions of *global* and *local* contexts is fundamental to how customization is achieved in Weasel. There is always one global context for each application. The global context contains everything that is shared between the users. Both parts of abstraction and UI state may be shared. Every user has his/her own local context. Now, customization is achieved by moving abstraction parts and UI state parts from the global context to the local context of each user. For example, if the UI state of a scroll bar is stored in the global context, then that scroll bar will provide WYSIWIS scrolling. However, if the UI state of the scroll bar is in the local context, then a user will not be able to scroll other users' views by using that scroll bar. Similarly, only updates made to abstraction entities in the global context will be noticeable by other users.

The binding facilities of RVL provide easy, declarative specification of which parts of the abstraction and the UI state that go to global context and which go to the local context (the default is that everything goes to the global context, the programmer must specify which parts go to the local context). Hence, we can easily specify what is customizable and what is shared. Figure 13 pictures the notion of global and local context.

Users using a groupware application often have different *roles*. The role of a user can have wide implications as to how the groupware application should respond to that particular user (or role). In Weasel we have the concept of a *signature*. Each user in a groupware session is given a unique

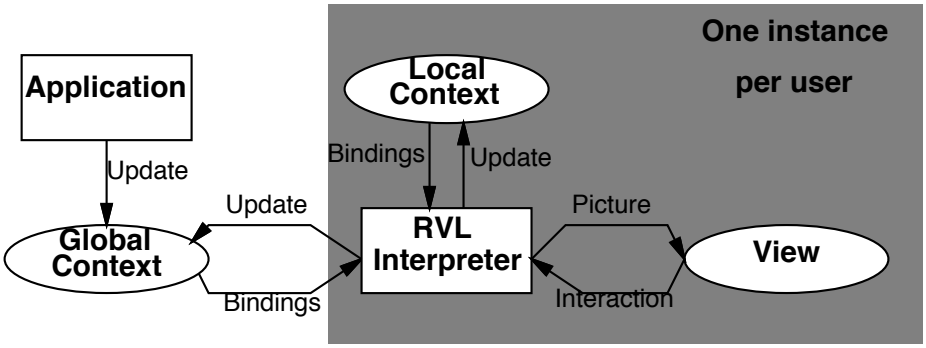


Figure 13: In the Weasel system, the global context is used to store shared abstraction and UI state. The local context provides easy customization. Note that the figure is simplified in that constraints have been omitted. The RVL interpreter is the realization of the bi-directional mapping between pictorial and abstract forms of data.

signature. Limited signature support has been built into the RVL language. This makes it easy to specify basic coordination like turn-taking or floor control. Or, more generally, the signature support in RVL makes it possible to customize differently to different users.

This section has discussed the relational view model and its implementation in the Weasel system. Though the implementation model may seem complicated, the conceptual programming model is simple; i.e. a central application bound to a set of views through relations. We repeat that the motivation behind the Weasel system is to demonstrate that the relational view model is useful and practical as development support for multi-user applications (e.g. desktop conferencing applications) having a graphical, interactive user interfaces. Weasel is one possible instantiation of the relational view model.

3.4.2 Example Applications

Our first example groupware application developed using Weasel is an inventory data base browser. A set of cities has inventories containing various products. A city's products may be ordered by other cities. Figure 14 shows an example session where two users are independently browsing through the inventory data.

The inventory data base example illustrates how view customization is achieved in Weasel by using RVL to declaratively specify what goes into the global and local contexts. Here, all the inventory data themselves are placed in the global context. Consequently, if two users are viewing the same data and one of the users makes changes to a data item, then both users will observe the change.

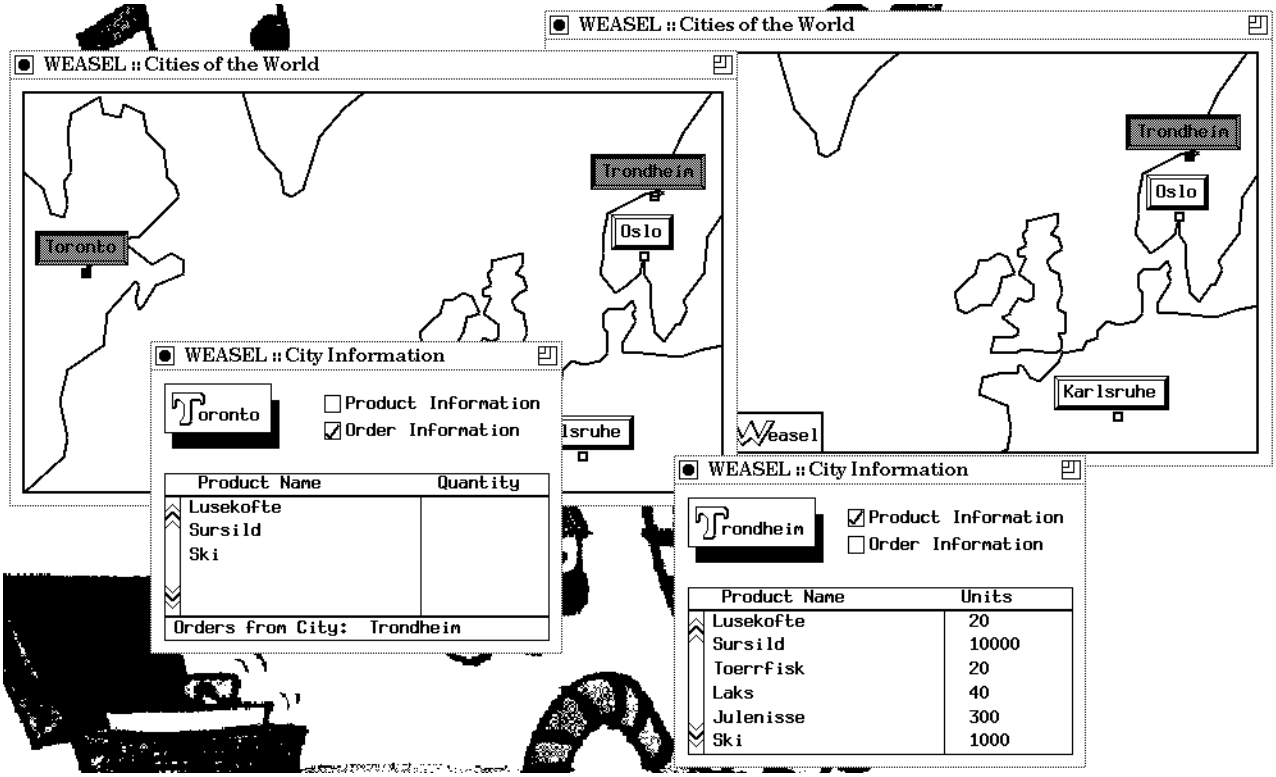


Figure 14: An inventory data browser. There are two users, each having a "Cities of the World" and a "City Information" view. The user on the left is looking at Toronto and what orders that city has on products from Trondheim. The user on the right is scrolling through the products that are available in the city Trondheim.

There is no global UI state in this example, so users can browse completely independent of each other. Browsing is done by clicking on a city on the map (large view) and use a scrollbar to go through either product or order information (small view). Note that it is easy to make the views more tightly coupled by changing the RVL specification (or, more presicely, the binding of the RVL specification).

One nice thing is that programmers never have to worry about updating views in the relational view model. The run-time system automatically detects any updates made to the abstraction (either the local or global context) and instantly figures out which actions need to be taken. Programmers are also totally unaware of the underlying implementation model (e.g. that the local contexts are actually residing in different address spaces, typically on the remote client machines on which the users are sitting).

The second example is a tic-tac-toe game. An example session is shown in figure 15. Two users take turn making moves by clicking on the tic-tac-toe board. This example demonstrates the use of signature constructs in RVL for specifying turn-taking and view customization according to the

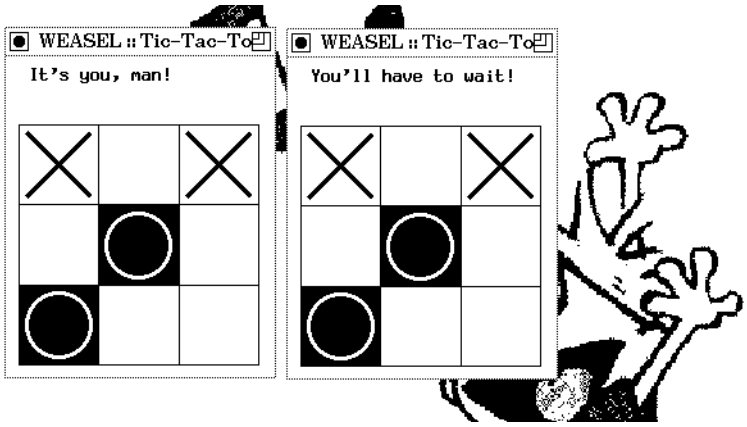


Figure 15: A tic-tac-toe example. It is the first (left) user's turn and he/she must click in an empty square to place an inverted O.

current role of a user.

Signature constructs in RVL allow this example application to be customized in three ways:

1. The user whose turn it is sees a different message in the top part of the view than the other.
2. Only the game board of the user whose turn it is, is sensitive to input.
3. The first user has inverted O's has his/her symbol, the second user has X's.

The complete RVL specification is about 90 lines of code (specifies the board and the appearance of different board items, specifies turn-taking and customization, prints a message saying whose turn it is, and prints a message congratulating the winner). The functional core, written in Turing plus, is trivial.

3.4.3 Evaluation

In Weasel, functional core of the application is specified in the Turing Plus language, which is well-suited to modular programming. User interface views may be specified separately for each module. Data state that is necessary for operating user interface metaphors such as buttons and scroll bars is kept separately from the data state of the functional core. All this provision for separating different part of an application into components is the main reason for the "good" iterative development rating.

Weasel does in many ways provide good support for incremental development. The relation specifications written in RVL offer programmers a way to easily specify sharing, view customization, and simple basic coordination in a declarative way. However, the functional core of an application must

be compiled every time it is modified. This is a big drawback. Even though RVL specifications are interpreted, we have chosen to give only an "ok" rating on support for incremental development.

It is very easy to reuse existing single-user Weasel applications and employ them in a multi-user setting. In fact, the normal development scenario when using Weasel is to first develop a single-user application that implements the functional core and some prototypical user interface views. Then, one switches to multi-user Weasel and incorporates sharing, view customizations, and coordination into the RVL specifications. We give a "good" rating for single-user reuse.

As we just argued, it is almost as easy to implement multi-user applications as to implement single-user applications in Weasel. The main reason for this is the conceptual programming model. The programmer sees a centralized model, i.e. the relational view model, and does not have to worry about distributed system issues. By requiring the use of special purpose languages for the different parts of applications, it is possible to provide well adapted language abstractions for the different programming tasks. Turing Plus is a nice language for programming the functional core, while RVL makes it easy to program views, customization, and sharing. The conceptual model criterion gets a "very good" rating, while the language abstraction criterion gets a "good" rating.

Using multiple languages is a problem when it comes to ease of learning. Also, RVL is a pure functional language and employs notions which may be foreign to many programmers. Weasel only rates "ok" on ease of learning. We give a "good" rating for declarativeness, mainly because of RVL, since Turing Plus is an imperative language.

Expressiveness is a problem in Weasel. Extensive use of separation of concerns and declarativeness provide many advantages but we have to pay an expressiveness penalty. In the current Weasel system, this is mostly due to unnecessary restrictiveness in the RVL language, and not so much a problem with the conceptual model. We only give an "ok" rating on expressiveness. There is no session management in Weasel.

Weasel's performance is acceptable for prototyping purposes, but not fast enough for production use. It should be noted, however, that the performance of Weasel applications degrades very slowly as large numbers of users are added to a session.

3.4.4 Conclusion

Weasel provides groupware developers with an interesting conceptual model and a special purpose language for specifying synchronous groupware applications.

Another interesting aspect is the semi-replicated implementation model of Weasel. Scalability measurements have shown that the semi-replicated architecture implementation of Weasel does

not experience severe performance degradation (as a centralized implementation would) when the number of users increases[4].

One should note that the functional core in Weasel applications can do computation on its own. This could be useful for implementing intelligent agents.

The biggest problem with the current version of Weasel is the expressiveness of the RVL language. The RVL syntax is also somewhat clumsy.

[4]Note that the performance measurements cited in [28] refer to an old version of Weasel. The latest version has significantly improved performance.

4 Conclusion

This section first briefly compares the evaluated tools at a more pragmatic level. Then, we propose a set of lessons that we hope will be of interest to groupware tool designers.

4.1 Tool Comparison

It should be clear from the work presented in this report that GroupKit is the tool of choice for those wanting to use a tool to develop "real" groupware. GroupKit is in a class of its own when it comes to session management and ease of learning. Unfortunately, it is also the poorest tool when considering the underlying design paradigm.

One thing that makes GroupIE and Suite interesting is the fact that they offer generic functionality that allows end-users to tailor the coupling of generated applications.

Suite and Weasel are the only tools that have been designed completely from scratch by their creators (though they were both single-user tools in the beginning). Consequently, these two tools offer interesting test beds for experimentation with conceptual models and special-purpose specification language constructs.

We think that one of the conclusions one can draw from this work is that it is still more difficult to develop groupware than to develop single-user applications. We will now, in section 4.2, take a look at how groupware tools can be improved to reduce the complexity of groupware development.

4.2 Lessons for Tool Designers

Here, we analyze the evaluation results. We first propose some lessons that we feel will suggest improvements to the development support provided by all the subject tools. Then, we discuss some additional issues that should be taken into account before one sets out to design new groupware tools.

We propose the following lessons from our tool evaluation:

1. Rapid prototyping support is greatly enhanced by providing an interpreted environment. GroupKit is a good example. One important observation of particular interest to groupware tool designers is that the combination of an interpreted language together with good session management proved an invaluable tool for quick testing of ideas in a multi-user setting. In GroupIE, the other tool with proper interpreter support, the lack of session management

made it much more troublesome to deploy prototypes in a multi-user setting.

2. The conceptual programming model that is exposed to the programmer should be higher level than the implementation model of the tool. Weasel is a good example of how a high-level conceptual model makes it easier to develop groupware. A high-level conceptual model also makes it easier to provide good programming languages.

3. Declarative programming language constructs can greatly simplify the task of tackling low-level issues that groupware developers often run into. The dependencies on global variables and callbacks in GroupKit and Suite do not cause serious problems in small applications, but in larger projects could cause difficulties in maintenance and iterative refinement. Declarative techniques, as used in Weasel and GroupIE allow issues of replication, concurrency control and consistency maintenance to be largely hidden from the programmer.

It is interesting to note that each of the four groupware tools considered here are extensions of tools for developing single-user user interfaces. This is reflected in the type of groupware applications the tools provide best support for, namely desktop conferencing applications where users interact exclusively through manipulating objects on a shared workspace. That is, the generated applications provide only a task space, not a person space [7].

Important groupware issues like supporting both synchronous and asynchronous communication processes [71] and better integration of task and person spaces [41] should be taken into account when designing new groupware tools. The latter issue will require continuous digital media like audio and video to be integrated into groupware tools. That, in turn, will force groupware tool designers to consider real-time media requirements in relation to hardware and software platforms and temporal relationship issues in relation to specification of mixed media [35].

4.3 Acknowledgments

We would like to thank Professor Ronald Baecker and Professor Hiroshi Ishii for supervising the project work that led to this report.

Professor T. C. Nicholas Graham provided many helpful comments and suggestions throughout the project in additions to proof reading the final draft of this report. His contributions are greatly appreciated.

We would also like to thank Prasun Dewan, Saul Greenberg, and Tom Rüdebusch for making their tools available to us.

This work was funded in part by a scholarship from the Royal Norwegian Research Council, a tuition fee waiver from York University, and a grant from NSERC.

References

[1] S. R. Ahuja, J. R. Ensor, and S. E. Lucco. A Comparison of Application Sharing Mechanisms in Real-time Desktop Conferencing Systems. In *Proceedings of the Conference of Office Information Systems*, pages 238–248. ACM, 1990.

[2] R. M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work, Assisting Human-Human Collaboration.* Morgan Kaufmann Publishers, ISBN 1-55860-241-0, 1993.

[3] R. Bentley, T. Rodden, P. Sawyer, and I. Sommerville. An architecture for tailoring cooperative multi-user displays. In J. Turner and R. Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 187–194. ACM Press, November 1992.

[4] R. Bentley, T. Rodden, P. Sawyer, and I. Sommerville. Architectural Support for Cooperative Multi-user Interfaces. *IEEE Computer*, May 1994.

[5] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53,103, December 1993.

[6] John Bowers and Tom Rodden. Exploding the Interface: Experiences of a CSCW Network. In *Proceedings of INTERCHI*, pages 255–262, (Amsterdam, The Netherlands, 24 April – 29 April, 1993), 1993. ACM, New York.

[7] W. A. S. Buxton. Telepresence: Integrating Shared Task and Person Spaces. In *Proceedings of Graphic Interface 92* (also in [2]), pages 123–129. Morgan kaufmann Publishers, 1992.

[8] V. G. Cerf. Networks. *Scientific America* (also in [2]), 265(3):72–81, September 1991.

[9] C. Cool, R. S. Fish, R. E. Kraut, and C. M. Lowery. Iterative Design of Video Communication Systems. In J. Turner and R. Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 25–32. ACM Press, November 1992.

[10] J. R. Cordy and T. C. N. Graham. Gvl: Visual specification of graphical output. *Journal of Visual Languages and Computing*, 1992.

[11] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson. MMConf: An Infrastructure for Building Shared Multimedia Applications. In F. Halasz, editor, *Proceedings of the Third Conference on Computer-Supported Cooperative Work (Los Angeles, Ca., Oct. 7–10)* (also in [2]), pages 329–342. ACM Press, 1990.

[12] P. Dewan. A Guide to Suite. Technical Report SERC-TR-60-P, Software Engineering Research Center, Purdue University, February 1990.

[13] P. Dewan. Coupling the User Interface of a Multiuser Program. In *ACM SIGGRAPH Video Review, Issue 87.* Originally part of the Video Program of ACM CSCW '92, November 1992.

[14] P. Dewan. Principles for Designing Multi-User Interface Development Environments. In J. Larson and C. Unger, editors, *Proceedings of the 5th IFIP Working Conference on Engineering for HCI (Ellivaari, Finland)*, pages 35–48, August 1992.

[15] P. Dewan. An Editing-Based Characterization of the Design Space of Collaborative Applicaitons. In *Proceedings of the 4th Conference on Organizational Computing, Coordination, and Collaboration*, March 1993.

[16] P. Dewan. Tools for Implementing Multiuser User Interfaces. In L. Bass and P. Dewan, editors, *User Interface Software*, chapter 8. John Wiley & Sons, ISBN 0 471 93784 3, 1993.

[17] P. Dewan and R. Choudhary. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[18] P. Dewan and R. Choudhary. Coupling the user Interfaces of a Multiuser Program. *ACM Transactions on Information Systems*, to appear.

[19] P. Dewan and J. Riedl. Toward Computer-Supported Concurrent Software Engineering. *IEEE Computer*, 26(1):17–27, January 1993.

[20] P. Dewan, J. Riedl, R. Choudhary, V. Mashayekhi, and H. Shen. An Overview of the Suite Collaborative Infrastructure and Applications. Unpublished note available by ftp (`ftp.cs.purdue.edu`, `pub/rxc/papers/sum.ps`), 1993.

[21] P. Dewan and E. Vasilik. An Object Model for Conventional Operating Systems. *Usenix Computing Systems*, 3(4):517–549, December 1990.

[22] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, ISBN 0-13-458266-7, 1993.

[23] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data* (Seattle Wash. May 2–4 1989), pages 399–407, New York, 1989. ACM.

[24] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some issues and experiences. *Communication of the ACM* (also in [2]), 34(1):38–58, January 1991.

[25] E. A. Fox. Advances in interactive digital multimedia systems. *IEEE Computer* (also in [2]), 24(10):9–21, October 1991.

[26] S. Freeman. De-constructing the workstation: Window systems, distribution and cscw. Position Paper, CSCW'92 Workshop on Tools and Technologies (Toronto), October 1992.

[27] T. C. N. Graham. Conceptual views of data structures as a programming aid. Master's thesis, Queens University at Kingston, Canada, August 1988.

[28] T. C. N. Graham and T. Urnes. Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work (Toronto, Oct. 1992)*, 1992.

[29] M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, April 1986.

[30] S. Greenberg. Sharing Views and Interactions with Single-User Applications. In *Proceedings of the Conference on Office Information Systems*, pages 227–237. ACM, April 1990.

[31] S. Greenberg, M. Roseman, D. Webster, and R. Bohnet. Issues and Experiences Designing and Implementing Two Group Drawing Tools. In *Proceedings of the 25th Annual Hawaii International Conference on the System Sciences* (also in [2]), volume IV, pages 139–150. IEEE Computer Society Press, January 1992.

[32] N. M. Guimaraes, N. M. Correia, and T. A. Carmo. Programming Time in Multimedia User Interfaces. In *Proceedings of the Fifth Annual Symposium on User Interface Software and Technology, (Monterey, California, Nov. 15–18)*, pages 125–134. acm press, 1992.

[33] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[34] R. D. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction – the sassafras uims. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[35] R. D. Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In B. A. Myers, editor, *Languages for Developing User Interfaces* (also in [2]), chapter 9, pages 125–143. Jones & Bartlett Publishers, 1992.

[36] R. D. Hill. Synchronization vs. responsiveness in distributed conversations. Position Paper, CSCW'92 Workshop on Tools and Technologies (Toronto), October 1992.

[37] R. D. Hill. The abstraction-link-view Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI 92*. ACM, acm press, May 1992.

[38] R. D. Hill, T. Brinck, S. L. Rohall, J. F. Patterson, and W. Wilner. The Rendezvous Language and Architecture for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, To appear, 1994.

[39] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[40] S. Hudson. How Programming Languages Might Better Support User Interface Tools. In B. A. Myers, editor, *Languages for Programming User Interfaces*, chapter 7. Jones and Barlett, 1992.

[41] H. Ishii and M. Kobayashi. Clearboard: A Seamless Medium for Shared Drawing and Conversation with Eye Contact. In *Proceedings of CHI'92* (also in [2]), pages 525–532. ACM, 1992.

[42] S. M. Kaplan, W. J. Tolone, D. P. Bogia, and C. Bignoli. Flexible, Active Support for Collaborative Work with ConversationBuilder. In J. Turner and R. Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 378–385. ACM Press, November 1992.

[43] J. C. Lauwers and K. A. Lantz. Collaboration Awareness in Support of Collaboration Transparency: Requirements for the next Generation of Shared Window Systems. In *Proceedings of CHI'90* (also in [2]), pages 303–311. ACM, 1990.

[44] J. C. Lauwers, K. A. Lantz, and A. L. Romanow. Replicated Architectures for Shared Window Systems: A Critique. In *Proceedings of the Conference on Office Information Systems, Cambridge, MA* (also in [2]), pages 249–260. ACM Press, April 1990.

[45] T. M. Levergood, A. C. Payne, J. Gettys, G. W. Treese, and L. C. Stewart. AudioFile: A Network-Transparent System for Distributed Audio Applications. In *Proceedings of the USENIX Summer Conference*, June 1993.

[46] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[47] Silvano Maffeis. Technologische grundlagen des computer supported cooperative work (cscw). Technical Report IFI TR 93.29, Institut für Informatik der Universität Zürich (in German), July 1993.

[48] B. A. Myers. Why are Human–Computer Interfaces Difficult to Design and Implement? Technical Report CMU–CS–93–183, Computer Science Department Carnegie Mellon University, Pitsburgh, July 1993.

[49] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. Vander Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[50] K. Narayanaswamy and Neil Goldman. Lazy Consistency: A Basis for Cooperative Software Development. In Jon Turner and Robert Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 257–264. ACM, acm press, November 1992.

[51] G. M. Olson, L. J. McGuffin, E. Kuwana, and J. S. Olson. Designing Software for a Group's Needs: A Functional Analysis of Synchronous Groupware. In L. Bass and P. Dewan, editors, *User Interface Software*, chapter 7. John Wiley & Sons, 1993.

[52] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, ISBN 0-201-63337-X, 1994.

[53] J. F. Patterson. Session Services and Synchronous Groupware. Position Paper at the ACM CSCW '92 Workshop on Tools and Technologies, November 1992.

[54] John F. Patterson. Comparing the Programming Demands of Single–User and Multi–User Applications. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology (Hilton Head, Carolina, Nov. 11–13*, pages 87–94. ACM, acm press, 1991.

[55] J. R. Rhyne and C. G. Wolf. Collaboration through shared event histories: A position paper. Position Paper, CSCW'92 Workshop on Tools and Technologies (Toronto), October 1992.

[56] J. Riedl and V. Mashayekhi. Suitesound: Collaborative multimedia. Position Paper, CSCW'92 Workshop on Tools and Technologies (Toronto), October 1992.

[57] M. Robinson. Computer Supported Co–operative Work: Cases and Concepts. In *Proceedings of Groupware '91* (also in [2]), pages 59–75, P.O. Box 424, 3500 AK Utrecht, the Netherlands, 1991. Software Engineering Research Centre.

[58] T. Rodden, J. A. Mariani, and G. Blair. Supporting cooperative applications. *Computer Supported Cooperative Work (CSCW)*, 1(1):41–67, 1992.

[59] M. Roseman. Tcl/Tk as a Basis for Groupware. In *Proceedings of the Tcl/TK '93 Workshop (Berkeley, California)*, June 1993.

[60] M. Roseman and S. Greenberg. GroupKit: A Groupware Toolkit for Building Real–Time Conferencing Applications. In J. Turner and R. Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 43–50. ACM, acm press, November 1992.

[61] M. Roseman and S. Greenberg. Building Flexible Groupware Through Open Protocols. In *Proceedings of the Conference on Organizational Computing Systems (Milpitas, Ca.)*. ACM Press, November 1993.

[62] M. Roseman, S. Yitbarek, and S. Greenberg. GROUPKIT REFERENCE MANUAL: A Guide to its Architecture, Interprocess Communication, and Programs. Included in the public domain Groupkit distribution, available by anonymous ftp from `ftp.cpsc.ucalgary.ca` under `pub/grouplab/software`, December 1993.

[63] M. Roseman, S. Yitbarek, and S. Greenberg. GROUPKIT TUTORIAL. Included in the public domain Groupkit distribution, available by anonymous ftp from `ftp.cpsc.ucalgary.ca` under `pub/grouplab/software`, December 1993.

[64] T. Rüdebusch. *CSCW: Generische Unterstützung von Teamarbeit in verteilten DV-Systemen.* Doctoral dissertation, Deutscher Universitäts-Verlag GmbH, Wiesbaden, ISBN 3-8244-2043-0, University of Karlsruhe, Germany (in German), 1993.

[65] T. D. Rüdebusch. Development and Runtime Support for Collaborative Applications. In H.-J. Bullinger, editor, *Proceedings of the Fourth International Conference on Human-Computer Interaction, Stuttgart, Germany*, Human Aspects of Computing, pages 1128–1132, Amsterdam, 1991. Elsevier Science Publishers.

[66] T. D. Rüdebusch. Supporting Interaction within Distributed Teams. In K. Gorling and C. Sattler, editors, *International Workshop on CSCW, Berlin, Germany*, pages 17–33, 1991.

[67] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *Transactions on Office Information Systems* (also in [2]), 5(2):147–167, 1987.

[68] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchmann. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[69] S. M. Stevens. Multimedia Computing: Applications, Designs and Human Factors. In L. Bass and P. Dewan, editors, *User Interface Software*, chapter 9. John Wiley & Sons, 1993.

[70] S. Takahashi, S. Matsuoka, A. Yonezawa, and T. Kamada. A General Framework for Bi-Directional Translation between Abstract and Pictorial Data. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology (Hilton Head, Carolina, Nov. 11–13)*, pages 165–174. ACM Press, 1991.

[71] M. Turoff. Computer-Mediated Communication Requirements for Group Support. *Journal of Organizational Computing* (also in [2]), (1):85;94–113, 1991.

[72] T. Urnes. A Relational Model for Programming Concurrent and Distributed User Interfaces. Master's thesis, Norwegian Institute of Technology, University of Trondheim, Norway (also available as Arbeitspapiere der GMD 643, Germany), April 1992.

[73] H. M. Vin, P. V. Rangan, and M. Chen. System Support for Computer Mediated Multimedia Collaborations. In J. Turner and R. Kraut, editors, *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 203–209. ACM Press, 1992.

[74] George M. White. A formal method for specifying temporal properties of the multi-user interface. In S. Gibbs and A. A. Verrijn-Stuart, editors, *Multi-User Interfaces and Applications*, pages 49–59. Elsevier Science Publishers B. V. (North-Holland), 1990.