Personalized Intelligent Information Systems

Workshop Report
Breckenridge, Colorado
January 11-14, 1987

Editors: Gerhard Fischer and Helga Nieper

Technical Report No. 87-9

INSTITUTE OF COGNITIVE SCIENCE
TECHNICAL REPORT 87-9

Gerhard Fischer and Helga Nieper (Editors):

Personalized Intelligent Information Systems

Report on a Workshop
Breckenridge, Colorado
Sunday, January 11th - Wednesday, January 14th, 1987

List of Participants (a list of their addresses appears in the Appendix):

Participants from Outside Institutions:

- John Carroll (IBM)
- Allen Cypher (Intellicorp)
- Kate Ehrlich (Symbolics)
- Robin Jeffries (Hewlett Packard)
- Erich Neuhold (Gesellschaft fuer Mathematik and Datenverarbeitung)
- Brian Phillips (Tektronix)
- Stephen Weyer (Apple)

Participants from the University of Colorado:

- Brigham Bell
- Hal Eden
- Gerhard Fischer
- Walter Kintsch
- Andreas Lemke
- Clayton Lewis
- Suzanne Mannes
- Helga Nieper
- Peter Polson
- Christian Rathke
- Paul Smolensky
- Althea Turner
- Emilie Young

Remark: Only a minimal amount of editing has been done on the individual contributions to publish this technical report as quickly as possible. We hope that readers are sufficiently interested in the contents of this report that they are not disturbed by the different styles used in the individual contributions.

This workshop was supported by grant No. MDA903-86-C0143 from the Army Research Institute.

Table of Contents

Part 1 -- Gerhard Fischer: Objectives of the Workshop 2

1. Goals of the Research 2

2. Problems 3
- 2.1 High Functionality Results in Low Usability 3
- 2.2 Reality is not User-Friendly -- or: There Will Always Be a Need for Complex Systems 4
- 2.3 Personalized Information Systems Are not Simply Databases 4
- 2.4 Design Constraints Should Be Based on Human Nature and the Real World 5

3. Ideas for Solutions 6
- 3.1 Intelligent Support Systems 6
- 3.2 Cooperative Problem Solving Systems 8
- 3.3 Situation Models versus System Models 9
- 3.4 Reusability 10
- 3.5 From "Human Computer Communication" to "Human Problem-Domain Communication" 10

4. Open Issues 11
- 4.1 End-User Modifiability -- Is it a Good or a Bad Thing? 11
- 4.2 Reducing the Need to Learn and Increasing Usability 12
- 4.3 Structure and Organization 13
- 4.4 The Need to Focus on Specific Systems 13

Part 2 -- Individual Presentations 16

5. Kate Ehrlich and Janet Walker: High Functionality, Information Retrieval, and the Document Examiner 16
- 5.1 Does High Functionality Imply Low Usability? 16
- 5.2 The Document Examiner as an Information Retrieval System 18

6. Emilie Young: Using the Full Power of Computers to Learn the Full Power of Computers 22
- 6.1 Interaction with Documentation Code Examples 22
- 6.2 STUMPER 22
- 6.3 Document Examiner Analysis 23

7. Steven Weyer: Dynamic Books and Computer-Supported Encyclopedias 24

8. Brian Phillips: Machine-Supported Information Management for Collaborative Work 26
- 8.1 Introduction 26
- 8.2 Intelligent Project Support Environment 26
- 8.3 Conclusion 32

9. Erich Neuhold: Databases and Information Bases, Architecture and Integration Issues 34
- 9.1 Introduction 34
- 9.2 Dynamic Integration of Heterogeneous Databases 37

10. Walter Kintsch: Knowledge Assessment and Knowledge Organization 40
- 10.1 Generating Knowledge Structures from an Associative Memory 40
- 10.2 Relation of this Work to Specific Design Projects 41
- 10.3 Generating and Updating Knowledge Structures from Text 42

11. Suzanne Mannes: Modeling the Generation of Knowledge Structures: The Basics 43

12. Peter Polson: Cognitive Complexity Theory and Transfer of Skills 44
- 12.1 Introduction 44
- 12.2 Models of Human Computer Interaction 44
- 12.3 Theoretical Models Transfer 46

13. Clayton Lewis: NoPumpG, EXPL, and Spatial Thought Dumper 50
- 13.1 NoPumpG: A System for Rapid Implementation and Prototyping of Advanced User Interfaces 50

13.2 EXPL: A Model of Causal Analysis and Generalization for Human-Computer Interaction 52
13.3 The Spatial Thought Dumper and the Explicitness Bug 52

14. Paul Smolensky and Brigham Bell: EUCLID, a System for Argumentation Support 54

15. John Carroll: Consultant Dialogues 55

16. Robin Jeffries: Task Scripts -- An Approach to Increased Access to Functionality 56

17. Christian Rathke and Hal Eden: Reorganization of Knowledge Structures 59

18. Andreas Lemke: Support for Design and Redesign 61
18.1 The Problems 61
18.2 Ways to Go 62
18.3 Construction Sets and Design Kits 63
18.4 Problems of Design Kits 64

19. Helga Nieper: Information Retrieval by Reformulation: From ARGON to HELGON 65
19.1 Introduction 65
19.2 Two Case Studies: Bibliography Data and Mail Messages in ARGON 66
19.3 Extensions to ARGON 67
19.4 HELGON 68

20. Allen Cypher: Thought-Dumping 69
20.1 Postponed Structuring 69
20.2 Minimal Structuring 69
20.3 Tools versus Intelligence 70
20.4 Memory Organization and Retrieval 70
20.5 Hypertext 71
20.6 Actual Usage 72

References 73

Appendix: List of Participants 79

List of Figures

Figure 2-1: Quantitative Analysis of Some Systems 3
Figure 2-2: Databases versus Personalized Information Systems 5
Figure 3-1: An Architecture for Intelligent Support Systems 7
Figure 3-2: Basic Components to Support Cooperative Problem Solving Processes 9
Figure 4-1: Reducing Learning and Increasing Usability Through New Presentations 14
Figure 8-1: Project Encyclopedia 27
Figure 8-2: Project Browser 28
Figure 8-3: Image Stored in Encyclopedia 29
Figure 8-4: Calendar Interface 30
Figure 8-5: Model of Task-Oriented Communication 30
Figure 8-6: Mail Interface 32
Figure 9-1: Reference Architecture for Advanced Information Systems 35
Figure 9-2: An Advanced Computing Architecture 36
Figure 9-3: Information System Architecture for Database Integration 38
Figure 18-1: Monolithic Systems do not Support Redesign 63
Figure 18-2: A Structured System (like the EMACS Editor) Supports Redesign 63
Figure 18-3: An Architecture for Design Kits 64
Figure 19-1: ARGON 65
Figure 19-2: KANDOR 66
Figure 19-3: KANDOR Individuals of a Bibliography and a Mail Database 67
Figure 19-4: HELGON 68
Figure 20-1: Filing a Note in a Bin 70
Figure 20-2: A Fisheye View 71

Gerhard Fischer and Walter Kintsch: Introduction

This workshop was organized by the research project "Theories, Methods, and Tools for the Design of User-Centered Computer Systems," which is funded by the *Basic Research Program of the Army Research Institute (ARI)* and which is carried out at the Department of Computer Science, Department of Psychology, and Institute of Cognitive Science at the University of Colorado, Boulder (principal investigators: Gerhard Fischer and Walter Kintsch).

This project is a long-term research effort with a strong interdisciplinary component and was globally defined in the proposal as follows:

> The goal of this research is to develop *theories, methods, and tools for the design of user-centered computer systems* and test them by designing, implementing, and evaluating a customizable *Personalized Intelligent Information System*. Our research effort is based on the hypothesis that user-centered system design cannot be done and understood without trying to test and extend existing systems and design new ones. At the same time, user-centered system design cannot be understood by just doing it; the system building efforts must be based on a deep understanding of the theoretical and methodological issues behind them, derived from Artificial Intelligence, Cognitive Science, Cognitive Engineering, and Cognitive Ergonomics (Human Factors). The software systems developed will be significant not only as technical achievements in computer science, but also because these systems are based upon analyses of how humans can best be helped to cope with complex information systems. The *results* of this research effort will make substantial contributions to a number of significant scientific and practical problems in the design and comprehension of complex user-centered systems.

The **objectives of the workshop** were less to present answers and final results, but

1. to bring together researchers who work on similar goals, problems, and questions;
2. to make jointly the goals of the research more precise and more specific;
3. to generate plans for action to tackle the goals;
4. to identify and enumerate a set of problems and questions;
5. to characterize the most promising ways to deal with these issues;
6. to foster communication and cooperation between these researchers.

We are planning to organize **future workshops** in the research area described by this report. If you are interested in participating, please send us a short notice.

Acknowledgements

We would like to thank the participants of the workshop who accepted our invitation and who contributed to this technical report.

Special thanks to the Basic Research Program of the Army Research Institute (ARI), especially to our program director Michael Drillings, for the support of our research. Without this support the workshop would not have taken place.

Part 1 -- Gerhard Fischer: Objectives of the Workshop

This part of the report is based on the content of the project proposal, the objectives of the workshop, the contributions of the participants (which are contained in the second part of the report), and the discussions carried out during the workshop.

1. Goals of the Research

The goals of this research were briefly mentioned in the above introduction. They can also be described by enumerating the *expected results.* We hope to make substantial contributions to a number of significant scientific and practical problems:

1. the theory of design and comprehension of complex user-centered systems;
2. the theory of knowledge use and knowledge retrieval for designers at all levels;
3. the definition, exploration, and instantiation of new methodologies (e.g., evolutionary design methodologies, coevolution between specifications and implementations to cope with design instabilities, strategies to incrementally derive well-structured problems from ill-structured ones, end-user control over systems);
4. the design, implementation, and evaluation of a customizable personalized intelligent information system;
5. a set of well-tested tools and metatools organized in a personalized intelligent information system;
6. guidelines for constructing user-centered systems based on the above theories, methodologies, tools, and metatools;
7. quantitative and qualitative measures to be used by designers and Human Factors specialists to evaluate the design of user-centered systems.

The human has to be the "fixpoint" for the design of many future human-computer systems. Theories and methodologies from Cognitive Science (being an interdisciplinary research area whose goal is to create a better understanding of a broad range of cognitive abilities like design, understanding, learning, remembering, and thinking) are an important source of knowledge and insight towards the construction of user-centered systems, as described in the project proposal [Fischer, Kintsch 86]. (For a good overview of this topic, see [Norman, Draper 86] for a discussion of approaches toward this design goal.)

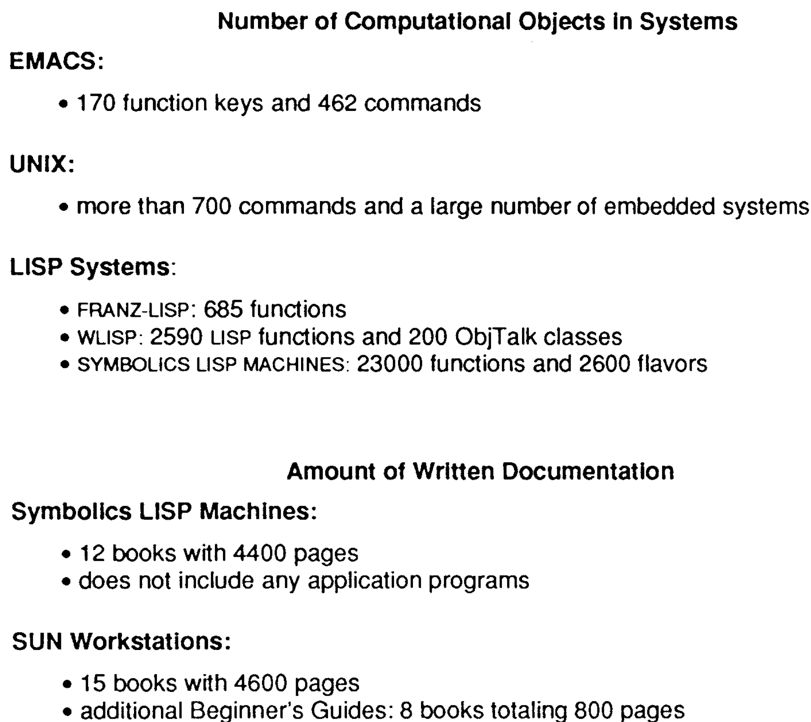
The approach taken by this research is in accord with one of the major goals of Artificial Intelligence. This goal, to provide intelligent support systems and tools to augment human intelligence and support cooperative problem solving processes, has received growing interest in the last few years. One reason for this interest is that work related to the more traditionally cited goal of Artificial Intelligence, to understand the nature of intelligent behavior and to build autonomous, intelligent, thinking machines, has begun to achieve some limited success (e.g., in high quality fully automatic translation, in automatic programming). However the difficulty of building intelligent systems has lead to the increasing belief that systems and tools directed at assisting people in complex, intelligent tasks are more viable and will lead to more important contributions, at least in the short run.

2. Problems

In this section, a number of problems and design tradeoffs towards the design of user-centered systems are identified and briefly discussed.

2.1 High Functionality Results in Low Usability

Modern computer systems are best thought of in terms of their capacity to serve as knowledge stores, rather than in their capacity to compute. The needs of users whose tasks involve a lot of complex knowledge can only be met by systems that are large and complex. We should not expect that these systems will be small and simple. In order to develop computer systems that are able to support complex knowledge use, a high degree of function must be built into these systems. To illustrate the dimensions of this functionality, an analysis of some typical systems is given in Figure 2-1.



Number of Computational Objects in Systems

EMACS:

- 170 function keys and 462 commands

UNIX:

- more than 700 commands and a large number of embedded systems

LISP Systems:

- FRANZ-LISP: 685 functions
- WLISP: 2590 LISP functions and 200 ObjTalk classes
- SYMBOLICS LISP MACHINES: 23000 functions and 2600 flavors

Amount of Written Documentation

Symbolics LISP Machines:

- 12 books with 4400 pages
- does not include any application programs

SUN Workstations:

- 15 books with 4600 pages
- additional Beginner's Guides: 8 books totaling 800 pages

Figure 2-1: Quantitative Analysis of Some Systems

Our preliminary empirical findings indicate that a number of cognitive problems may prevent users from successfully exploiting the potential of high-functionality systems. Users may lack one or more of the following:

- a mental model for the design space, i.e., users do not know about the *existence* of tools;[1]
- retrieval mechanisms, i.e., users do not know how to *access* tools;
- knowledge of applicability conditions, i.e., users do not know *when* to use these tools;
- understanding of the results produced, i.e., users do not understand what the tools produce for them;
- support for change, i.e., users cannot combine, adapt, and modify a tool to their *specific* needs.

A consequence of these problems is that these systems are underused and the broad functionality is of little value. We are strongly convinced that what is needed is *not quantitatively* more information but *qualitatively* new ways to structure and present information.

2.2 Reality is not User-Friendly – or: There Will Always Be a Need for Complex Systems

The dimensions of the modern computer systems (as given in Figure 2-1) indicate that to get acquainted with this amount of information (leave alone to master it) will not be a matter of weeks or months, but potentially years. Because the real world is complex, there will always remain a need for complex systems to cope with the knowledge worker's amorphous tasks. The existence of these huge systems is not based on their designers' inabilities or their unwillingness to strive for simplicity.

Assuming that future systems will not be constructed from scratch, but through reuse and redesign, these systems seem to be ideal candidates for such a methodology. Systems offering such a rich functionality are a mixed blessing: in a very large knowledge store it is much more likely to find something related to what we need, but it is also much more difficult to find something specific.

2.3 Personalized Information Systems Are not Simply Databases

Personalized information systems are constructed by individuals to serve and support their information processing needs (e.g., as a natural extension of their short and long term memory). It is absolutely crucial that the overhead in the *cognitive dimension* (e.g., being forced to create abstractions, being forced to be explicit, etc.) and the *technical dimension* (e.g., typing the information into the system, keeping changing information consistent, etc.) to generate these information structures is minimal. Unless individuals (using their own personal metric for effectiveness) come to the conclusion that the benefits of these systems outweigh their costs, these systems will **not** be used.

Figure 2-2 enumerates the needs met by personalized information systems that are not met by database systems.

[1] *Tool* is used here in a generic sense, meaning either a building block (e.g., a procedure or a class) which becomes part of the artifact constructed or a support system which helps to develop an artifact but does not become part of it. The "data / program equivalence" in LISP systems eliminates this distinction anyway.

	Databases	Personalized Information Systems
Number of Elements	many	few
Complexity of a Single Object	little	great
Inferences	no	yes
Structures of the Problem Domain	definite	must be defined
Selection/Filter/ Perspectives/ Intelligent Summarizing	advantageous	absolutely necessary
Heterogeneity of Information Structures	little	great
Use	storing many facts	support in planning, decision making, problem solving
Representation Techniques	relational databases	inheritance, defaults, rules, constraints, inferencing, active values

Figure 2-2: Databases versus Personalized Information Systems

2.4 Design Constraints Should Be Based on Human Nature and the Real World

In his discussion of large scale designs in the context of social planning, Simon [Simon 81] regards the American constitution as an impressive example of success in human planning. In part, he contributes its success to the acceptance of humans being as they are and not as we would like them to be ([Simon 81], p. 163):

> They (the founding fathers) did not postulate a new man to be produced by the new institutions but accepted as one of their design constraints the psychological characteristics of men and women as they knew them, their selfishness as well as their common sense.

In the discussions during the workshop, many participants generated warnings and examples that to be successful in the design of user-centered systems, the designers must take human nature and the real world seriously as design constraints. Some of the important issues discussed will be briefly described.

Production paradox. The production paradox [Carroll, Rosson 86] articulates why the assumption that people are always eager to learn new things has no empirical justification. People use computers because they want to get something accomplished. The positive side of this behavior is that people focus on their work, the negative side is that they are not motivated to spend time learning about the system

just to find out what they can do.

Do we really need a computer? It was noted that certain things (e.g., calendars which can be carried around all the time, office floors which serve well to provide a spatial organization for our information, etc.) work well in their current form -- and computer supported systems may even have major shortcomings.

Do we want to be explicit all the time? To put computer systems into the loop, we have to be explicit about many things which we just kept somewhat in our head or dealt with in informal ways. Many people do not want to inform a computer system about their intentions and goals, and they do not want to put information in the computer (i.e., type it in) which they have already stored somewhere else. In addition, many participants believed that some of these information stores (e.g., folders in mail systems) are used like diaries: *they are written, but never read.*

Do we support relevant activities? How good are computer-supported office systems or engineering support systems, if these groups of people spend very little time in the office or on actual engineering activities (e.g., many engineers spend 60% on non-engineering activities, and office workers spend more time in meetings than in their offices as revealed by a study done at XEROX PARC).

Certain types of systems only work, when all people in a group buy into them. Examples of such systems are: electronic mail systems, automatic scheduling systems, and community knowledge bases (e.g., databases for references, addresses, etc.). How can we achieve this general participation? Who should make decisions?

3. Ideas for Solutions

In this section, we will briefly describe some ideas for solutions. More detail about specific theoretical approaches and solutions can be found in the contributions in the second part of this report.

3.1 Intelligent Support Systems

We claim that in future computer systems, a high percentage of the computational power of these systems should be used to make them usable, manageable, and useful. Our system-building efforts were based on the belief that *the intelligence of a complex computer system must contribute to its ease of use.* Truly intelligent and knowledgeable human communicators, such as good teachers, use a substantial part of their knowledge to explain their expertise to others. In the same way, the intelligence of a computer should be applied to providing effective communication. We have constructed the following prototypical systems:

1. DOXY: a computer-supported documentation system [Fischer, Schneider 84];
2. PASSIVIST and ACTIVIST: prototypes for passive and active help systems [Fischer, Lemke, Schwab 85];
3. LISP-CRITIC: an example for a critic system [Fischer 87a];
4. KAESTLE: a visualization tool being part of our software oscilloscope [Boecker, Fischer, Nieper 86].

We need an architecture to support a coherent design strategy that treats intelligent support systems not

as add-ons to existing systems but as integral parts of a user-centered design approach. In Figure 3-1 we illustrate a system architecture that we envision in response to this design criterion.

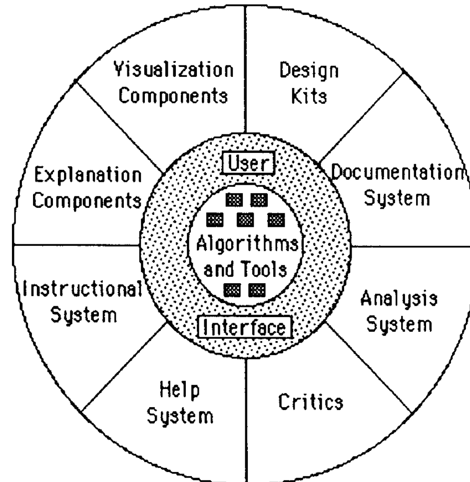


Figure 3-1: An Architecture for Intelligent Support Systems

Intelligent support systems should contribute to the achievement of the user goals. These systems can assist users by

- helping users ask the right (or the interesting) questions.
- eliminating the need to ask questions; this is especially relevant if we want to help users to find out what they do not know about. To ask a question, one must know enough to know what is not known! Active systems are needed which volunteer help in appropriate situations rather than responding to explicit requests. In order to volunteer information, the system needs some understanding of what the user is doing.
- enhancing incremental learning and supporting learning on demand; this may motivate learners more and help to overcome the production paradox.
- eliminating the need for learning by providing better systems.
- supporting design for reuse and redesign.

Our previous statement that "reality is not user-friendly" should not be misunderstood. There is no "conversation law on complexity" [Simon 81], which requires that the complexity and usability of a system is a given constant. Complexity can be reduced by

- exploiting what people already know (e.g., human problem-domain communication [Fischer, Lemke 87a] exploits this aspect);
- using familiar representations (based on previous knowledge and analogous to known situations);
- exploiting the strengths of human information processing (e.g., the power of our visual system [Boecker, Fischer, Nieper 86]);
- segmenting information into microworlds [Fischer 87b];

- eliminating learning through "better" systems.

3.2 Cooperative Problem Solving Systems

Our current intelligent support systems are "one-shot" affairs. They may give advice, but there is no possibility of regarding that advice as a starting point for a cooperative problem-solving process. If we study advice giving and advice seeking between humans, these dialogues are carried out in a natural communication paradigm. *Natural Communication* is more than the ability to communicate in natural language. It is the ability to engage in a dialogue. When humans (e.g., a novice and an expert) communicate, much more goes on than just the request for factual information. Novices may not be able to articulate their questions without the help of the expert. The advice given by the expert may not be understood and/or the advisee may request an explanation of it. Each communication partner may hypothesize that the other partner misunderstood him/her or they may provide information which they were not explicitly asked for.

Instead of investigating the design of "fool-proof" systems, we feel that designing *systems for error* and *for the management of trouble* [Brown 86] are more realistic goals. The problem in human-computer interaction is not simply that communicative troubles arise that do not arise in human communication, but that when the inevitable troubles do arise, the same resources are not available for their detection and repair [Suchman 85].

Errors should not cause a breakdown of the interaction; that is, errors should not be perceived as such in the first place, but rather as misunderstandings. As such, they should be regarded as an integral part of the process of accomplishing a task. All efforts in a cooperative problem-solving process should be regarded as iterations towards a goal. An error should lead to a situation which can be described as "Let's talk about it" [Lewis, Norman 86]. The goal of a cooperative endeavor is not to find fault and to assess blame, but rather to get the task done.

In our work, we will model cooperative problem-solving processes using the basic units U_1 to U_4 represented in Figure 3-2. The four primitives can be combined in arbitrary ways.

Each basic unit can be illustrated by an example:

1. The LISP-CRITIC [Fischer 87a] is an example for U_1. The human generates a product; the computer criticizes it and generates an alternative which can serve as the input to U_2 or U_3.
2. KAESTLE [Boecker, Fischer, Nieper 86] is an example for U_2. The computer automatically generates a layout for a complex LISP data structure. The human criticizes it and provides additional requirements to the planning algorithm, and the computer generates the next layout.
3. KAESTLE can also serve as an example for U_3. The new product in this case is generated by the human, who can edit the layout generated by the system.
4. RABBIT and ARGON (see section 19) are an instantiation of U_4. The user generates a query; the system instantiates the query with an example, and based on the example provided by the system, the user generates the next query.

The criticism provided in such interactions can serve multiple purposes: it can become itself an object of interrogation, and it can serve as a starting point for a learning process.

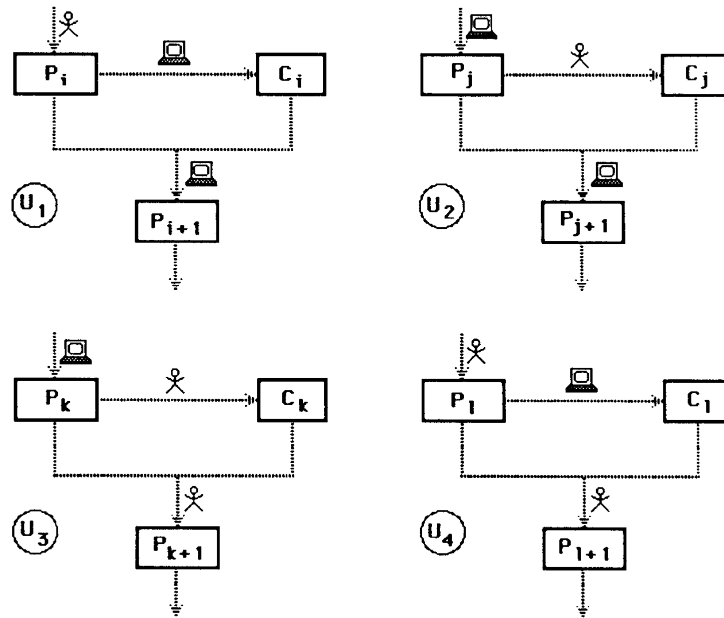


Figure 3-2: Basic Components to Support Cooperative Problem Solving Processes

Cooperative problem solving systems are important for:

1. the cooperation between a user and a system, e.g., RABBIT-like systems (see section 19) assist users to incrementally formulate a query; a design kit (see section 18) guides a user through a large design space, etc.;
2. the cooperation between users, e.g., the use of a community knowledge base as in the STUMPER system (see section 6), mail systems, software production systems. These systems are best characterized as semi-formal systems, where the computer system is able to interpret parts of the information structures (the EUCLID system in section 14 is an example of such a system).

3.3 Situation Models versus System Models

Human assistance, if available on a personal level, is still the most useful source of advice and help. Learners are able to ask a question in an infinite variety of ways; they get assistance in formulating the question; and they can articulate their problem in terms of the *situation model* rather than being required to do so in terms of the *system model.* The human serves as translator from the situation model into the system model. Computer-supported systems only "understand" information and questions expressed in the system model.

The *situation model* [Fischer, Kintsch 86] is a mental representation of the situation as the user sees it, including the problems motivating the design task, general ideas about how a solution is to be obtained, and a characterization of the desired goal state.

The first stage of problem solving consists in transforming this verbal, imprecise situation model into a formal *system model.* The system model consists of a set of operations that, if employed, will result in the desired solution. These operations must all be within the repertory of the system, i.e., for each operation there exist one or more commands, depending upon context, to execute it. Thus, the system model is a sequence of *operations*, which can be instantiated in a particular context through some *command.*

At the level of the situation model, goals refer to actions and states in the real world (the things to be achieved with a computer system); they may be precise or imprecise, but the important point is that they do not belong to the computer world. In contrast, the system model is part of the formal world of computers. Depending on the problem, goals can be expressed in terms of real world actions and events, or in terms of formal, computational methods. Design tasks require translating the problems expressed in the situation model to the system model and constructing the right sequence of operations that yield the solution. To do this, *search strategies* have to be used. Psychological research has shown that there are great differences between the efficient and successful strategies used by experts and the inefficient and ineffective search strategies of novices.

Sometimes a design task is approached with a fully elaborated, precise situation model; the designer knows exactly what he or she wants. The problem is only to figure out how to do it, i.e., to find a sequence of methods that define a workable system model. More typical is the case where the situation model is inadequate, imprecise, incomplete, and inconsistent. In this case, the situation model must be modified. Herein lies one reason why help systems are inferior to human advice. Help systems focus on problems at the level of the system model, that is, the actual operations and commands needed. But users may have a confused situation model, which prevents them from making good use of the help facilities available. A human expert, however, will try to figure out what the user *really* wants or needs, before jumping into the technical details. The situation model resulting from this interaction will have a better correspondence to the system model for accomplishing the task at hand.

3.4 Reusability

The most immediate value of stores of knowledge embedded in personalized information systems is that one can make use of previously acquired information and previous problem solving processes (carried out both by ourselves and others). Having very rich information stores available (see Figure 2-1), it should be possible *to take advantage of what already exists rather than reinventing the wheel.* Reuse of existing components and redesign of existing systems will be the dominant way to construct complex systems in the future. Just as one relies on already established theorems in a new mathematical proof, new systems should be built using existing parts as much as possible. In order to do so, the function of these parts must be understood. An important question concerns the level of understanding that is necessary for successful redesign: exactly how much does the user have to understand?

Issues arising in reusability are discussed in sections 9, 17, and 18.

3.5 From "Human Computer Communication" to "Human Problem-Domain Communication"

Most computer users are not interested in computers per se, but they want to use the computer to solve problems and to accomplish their tasks. To shape the computer into a truly usable and useful medium,

we have to make it invisible and let users work *directly* on their problems and their tasks.

Human problem-domain communication [Fischer, Lemke 87a] provides a new level of quality in human-computer communication because it permits us to build the important abstract operations and objects of a given application area directly into the environment. This implies that the user can operate with personally meaningful abstractions. In most cases we do not want to eliminate the semantics of a problem domain by reducing the information to formulas in first-order logic or to general graphs. Whenever the user of a system can directly manipulate the concepts of an application, programs become more understandable, and the distinction between programmers and non-programmers vanishes. We have to develop models of task-oriented communication and support the development of systematic domains (e.g., we have done this with WLISP for the domain of user interfaces).

4. Open Issues

The borderline between where the workshop participants felt that we had some ideas for solutions and what they considered as open issues is not precisely defined. In this section we will enumerate a number of issues which were identified as being important, but where no consensus was reached as to the best way to approach these problems. One may also conclude that many of these are design tradeoffs. One has to try to come up with the best solution possible given the individual situation, the goals, the abilities, and the environment.

4.1 End-User Modifiability -- Is it a Good or a Bad Thing?[2]

How much control should users have and how much control do users want to have? There is no doubt that taking away the control from users in cases where they did not want to have it is regarded as progress in technology (e.g., the automatic transmission in automobiles). Questions which one has to ask are:

- Does end-user modifiability (or customizability) make systems more or less complex? Potentially both possibilities exist: laws are so complex, because they have to anticipate all possible situations and cannot be adjusted dynamically. Forms on papers are complex (with an elaborate branching structure) because they have to cover many different cases. We conjecture that an electronic form system in which forms would be generated dynamically based on users profiles could lead (if done right) to forms of greatly reduced complexity.
- Should it be achieved by *adaptive* (i.e., the system itself changes its behavior based on a model of the user and the task) or adaptable systems? Do we have to choose?
- How can end-user modifiability be achieved? Is programming necessary for it? Which set of constrained design processes can be supported [Fischer, Lemke 87b] which would give the wanted control to the user?
- The question to be asked about control should in many cases not be "how much customization is possible?" but rather "how little customization is necessary?"

This design decision can be characterized further by the following observations:

[2]Robin Jeffries has made major contributions to this section.

- Users often do not want to be burdened with certain decisions. The example of TROFF requiring the baseline to be set separately from the font size is a case in point. The typical user will never encounter a situation where the baseline he or she wants is not the standard one that goes with that font size. It just increases the memory burden on the user to have to remember to change one when the other is changed. SCRIBE manages to have it both ways -- when you set the font size, the baseline is automatically adjusted, but you can override that value if you like -- but few interfaces do this. Giving people access to every knob and dial not only gives them the opportunity to tweak such things, but also bestows on them the responsibility to do so.

- Users often do not have enough knowledge of the domain to make certain decisions well. Typesetting is an example. Modern technology (providing formatting systems and requiring camera ready copies) has suddenly put all of us into the role of typesetters, a trade which has accumulated knowledge over the centuries. As many documents illustrate, this knowledge is not automatically possessed by every user. Therefore the advent of document formatters has greatly increased the number of poorly designed documents we see. Most people simply do not have enough knowledge of document design to create a visually pleasing document that clearly conveys their intended message. When we take a task out of the hands of the specialist and give it to the untrained generalist, there will be a decrease in quality. In some areas users would rather have the poor quality result than pay for the specialist. (Apparently this is true for document production.) How useful is it to provide users with a tool, if they don't have the knowledge to use it wisely? Is there any hope of providing that level of expertise within the tool?

- In a system of any size, every decision interacts with some other part of the system. It may not be as obvious as font size interacting with baseline distance, but changes to one component will invariably have far reaching effects. The effects may be aesthetic, as when the user changes one icon of a set so that it clashes with all the others; or functional, as when the user changes the width of the scroll bars in an interface and this makes it difficult (due to the gain on the mouse) to stay within the scroll bar bounds. One needs a good understanding of how any change will affect all the components of the system. The question remains: is it possible to provide a design kit that can anticipate these interactions and explain them to the user?

4.2 Reducing the Need to Learn and Increasing Usability

In analyzing of the design tradeoffs for mastering complex systems, it is crucial that we learn when to *enhance* learning and understanding and when to *reduce* learning by providing more adequate representations. In other words: instead of building help and assistance for inadequate systems, build better systems which require less assistance and support. The amount of expertise necessary to do a task is determined not only by the nature of the task but also by the design of the system being used to accomplish the task. We are convinced that qualitatively different ways of communication between humans and computers can greatly reduce the need for learning in many situations. Innovative systems should be explored as alternatives to support tools developed for inferior systems.

We illustrate the tradeoff between enhancing learning and reducing learning with the example shown in Figure 4-1. In almost all current interfaces to UNIX, directory trees cannot be shown externally in a two-dimensional representation. To get an overview, users draw graphs by hand and must update these graphs also by hand if the directory structure is modified. Using one of our software oscilloscope tools, we can represent the directory as a tree structure on the screen. In the traditional representation, switching back and forth between different directories requires long, error-prone command sequences using the cd command. A system like our active help system ACTIVIST [Fischer, Lemke, Schwab 85] could discover

this kind of behavior and suggest the `pushd` and `popd` commands instead. These commands create a stack with the used directories and make the switching process less cumbersome. This suggestion can be seen as an enhancement of learning that becomes obsolete through the display possibilities offered by the software oscilloscope.

4.3 Structure and Organization

There was a fair amount of discussion about the question "Is structure desirable: yes or no?" People do not like to be forced to generate structures. In early problem solving stages the enforcement of structure may get in the way when people start to do something. Different approaches towards this problem are taken by these systems:

- NOTEPAD (see section 20) postpones the necessity for creating a structure and requires only a minimal amount of structuring;
- SPATIAL THOUGHT DUMPER (see section 13) uses spatial relationship as the only clue;
- NoteCards [Halasz, Moran, Trigg 87; Trigg, Suchman, Halasz 86] requires substantial structuring;
- EUCLID (see section 14) is based on the assumption that argumentation benefits from being structured.

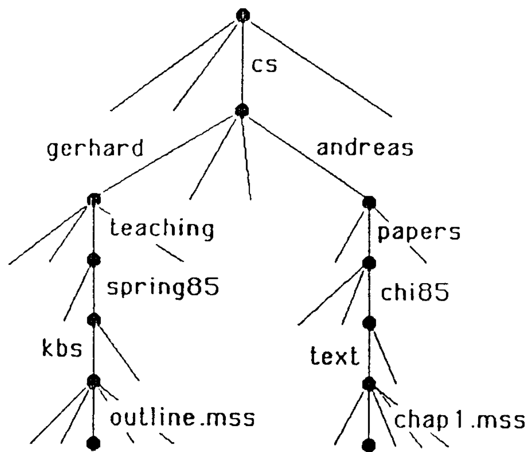
Obviously, there is another design tradeoff: we would like to take advantage of many structuring principles to retrieve and use information -- but we are much less willing to take the overhead into account which is necessary to generate this structure. We don't want to structure information explicitly, but we do want to retrieve information using structural properties. Other questions which were discussed:

- If we have no structure, then there is no need for restructuring (see section 17).
- If we impose a structure, which form should it take: hierarchies, inheritance networks, associations, ...?
- Is a structure statically given or generated on demand? Walter Kintsch (see section 10) asserts there are no such things as frames, scripts, etc., in people's mind; they get generated when needed.

4.4 The Need to Focus on Specific Systems

It was generally agreed that it would be a good idea to test the problems, ideas, and open issues by focusing on some specific examples. Potential candidates which were discussed are:

1. the LISP machines -- especially the *Document Examiner* (see section 5):
    - a system which has the level of complexity (see Figure 2-1) which makes the issues discussed not a luxury but a necessity,
    - Extensions and improvements are needed (and different people work on them):
        a. active examples and STUMPER (see section 6),
        b. extensibility by the end-user, RABBIT/ARGON-like mechanisms to access and update information (see section 19),
        c. hypertext representations (see sections 7 and 8),
        d. structuring the information space according to the model of increasingly complex microworlds [Fischer 87b],
        e. representing information structures which cover situation models of users and not only the system model (see section 3.3);

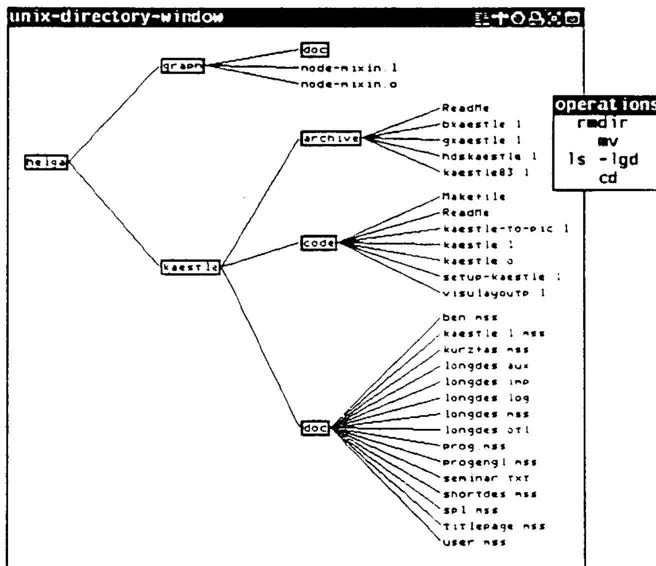


% cd ~gerhard/teaching/spring85/kbs

% cd ~andreas/papers/chi85/text

The UNIX commands to switch between directories

The file hierarchy as hand-drawn picture



The automatically generated, graphical layout structure.
File operations can be directly carried out on this structure.

Figure 4-1: Reducing Learning and Increasing Usability Through New Presentations

2. a *bibliography database:*
    - extend an existing bibliography database (used for SCRIBE) into a knowledge base with a much richer access structure,
    - experiment with knowledge bases with no a priori structure, but allow users to be able to generate structures corresponding to their needs dynamically (see section 10),
    - RABBIT/ARGON-like mechanisms to access and update information (see section 19),
    - link different databases and study naming problems (see secton 9),
    - study the reorganization problem in this context (see section 17).

Part 2 -- Individual Presentations

5. Kate Ehrlich and Janet Walker: High Functionality, Information Retrieval, and the Document Examiner

There were two issues I (Kate Ehrlich) talked about at the workshop:

- Does high functionality imply low usability?
- The Symbolics Document Examiner as an Information Retrieval System.

I will present a few comments on the issue of functionality followed by a summary of the Document Examiner by Janet Walker, one of its original architects.

5.1 Does High Functionality Imply Low Usability?

It is commonly believed that systems with a large number of documented commands or functions (high functionality) are hard to use. For instance, users do not always know where or how to find the information they need and consequently end up failing to exploit the power that the system offers (see section 2.1 of this document). As a user interface person working in a company that produces high functionality systems, I want to make these systems more accessible to users without arbitrarily limiting the benefits of the functionality. The comments below represent some initial thoughts on what makes high functionality systems hard to use and some suggested strategies for making them more accessible, especially to new users.

The Problem

In any system, a user has to search through a set of options to find the command or function which will accomplish the desired task. In a high functionality system, the problem of finding the right command is made harder by the fact that there are frequently many ways to accomplish the same task. Consider the following alternatives for the simple task of deleting a sentence in Zmacs, the EMACS-like editor on the Symbolics machine:

1. Position the cursor at the beginning of the sentence and press Meta-K (delete sentence command).
2. Position the cursor at the beginning of the sentence and press Control-<n> Meta-D where <n> is the number of words in the sentence.
3. Position the cursor at the beginning of the sentence and press Meta-D (delete word command) repeatedly until all the words in the sentence have been deleted.
4. Position the cursor at the beginning of the sentence. Mark the text to the end of the sentence. Press Control-W to delete the marked text. There are several ways to mark the text, here are a few of them:
    - Move the mouse from the beginning to the end of the sentence keeping the left button held down.
    - Mark the beginning of the sentence, use Control-S (the search command) to move cursor to the end of the sentence.
    - Mark the beginning of the sentence, then use Meta-E to move cursor to the end of

the sentence.

- Use the keyboard commands of `Control-A` (go to the beginning of the current line), `Control-E` (go to the end of the current line), `Control-N` (go to the next line), `Control-P` (go to the previous line), `Meta-F` (go to the space before the next word) and `Control-F` (move to the next character), to move to the end of the sentence.

The thing to note is that even though each of these alternatives will achieve the same result, they are not equally easy for most users. Moreover, even in this simple example, the options represent four different strategies or interaction styles: structural delete (#1), modified structural delete (#2), incremental delete (#3), mark-and-delete (#4).

The first option is probably the most basic, because it is a natural extension of the set of delete commands (`Control-D` deletes characters forward, `Meta-D` deletes words forward, `Meta-K` deletes sentences forward). The second option, though probably overkill in this context, is an example of what I would consider a more advanced command because of the `Control-<n>` component as a modifier of a basic command. The third option is a simple extension of the first, and the kind of option that I believe new users will resort to for quite a long time before switching to a more efficient, but harder command. The basic "mark-and-delete" approach given as the fourth option, is a very common interaction style for people who like to use a mouse. It is general enough to have many variations and to be applicable across a wide range of tasks.

New users do not understand which are the basic and which are the advanced commands, nor do they understand that the options represent different, and generalizable, interaction styles. To them, the system seems imponderable because they not only have to search through a large list of options, but they do not yet know how to choose amongst them.

Approaches to Solving the Problem

Thus, having defined the problem of high functionality as being a multiplicity of commands rather than simply a large number of commands, I see the solution as being to present users with "paths" through the system which focus options on basic commands and a single, generalizable interaction style. This approach is somewhat similar to "training wheels" [Carroll, Carrithers 84] in that it seeks to guide the new user's exposure to the system.

An analogy. Let me try and explain the approach a little better by drawing an analogy between learning a high functionality computer system and learning to drive in a new city, in this case Boston.

Boston is like a high functionality system because there are many places to go, but there are also lots of routes to get where you are going. Anyone living in Boston must first learn a few main roads preferably major highways and then, when these routes are well-rehearsed branch out to learn some of the side roads. The main roads are like the easy but less efficient commands. They will get you where you are going with the least problem because there are signs, very few places to turn off and not many distractions. But they won't always be the quickest route. The side roads are like the more complex commands. They may be the quickest way to go, but you have to understand something more about the city before using them because there are few signs, lots of wrong places to turn and it is easy to get lost.

Given this view of high functionality systems as cities with poor road maps and no clear markings be-

tween the major and minor routes (basic and advanced commands), there are several ways in which we can make the system more accessible, especially to new users.

- Design the system right, at the beginning, to distinguish between simple commands and more complex commands (between main roads and side streets).
- Think carefully about the necessity of including a particular piece of functionality
  - is it redundant, and therefore potentially confusing with a similar command?
  - is it really required - does it make something possible which was impossible; will it really be used by even a handful of people?
- Tutor new users by first teaching them a small set of basic commands (the "main roads"), and then build up from that knowledge by revealing information about more advanced, alternative commands ("side streets"). The presentation of these more advanced commands should be accompanied by a description of the model of the system to help explain differences in interaction styles.
- Provide tutoring and information retrieval strategies that help users develop a mental model of the layout - main roads, side streets, connections, common routes. As they develop more experience they can use their mental model to explore and develop routes of their own.

At Symbolics we don't have the luxury of re-designing the system. However, there are projects in-house which are beginning to take one or more of the other approaches to make the system more accessible to new users.

The above solutions are specifically addressed to the hurdles that new users face in learning how to use a high functionality system. But high functionality systems also pose problems for intermediate level users who understand enough to perform most tasks, but who may not always work in the most efficient manner. How do people develop better routes? They learn by chance turns down an unfamiliar road that unexpectedly links up to a road they already know, by learning from taxi drivers or by learning from a friend who is more knowledgeable about a particular part of the city ("experienced users"). But mostly they learn by being curious enough to explore new roads, and by having a good enough cognitive map to help them recover from errors. It should be possible to give new users of even high functionality systems the grounding and confidence they need to develop their own routes.

5.2 The Document Examiner as an Information Retrieval System

Principles and Goals behind the Project

The Document Examiner was part of an overall project to create, maintain, and deliver system documentation. Symbolics' requirements were for a documentation process that enabled our writers to produce documentation that was good enough and to do it fast enough. Design decisions were always made on the basis of these requirements and of overall utility or practicality. This was very much a "real world" project.

Document Structure

My fundamental belief was that writing documentation is a cognitive activity that in many ways parallels writing software. At most places writers and programmers use very different kinds of tools for their work. That was true at Symbolics in early days too. At Symbolics, however, the pace of development was much faster than that of writing. Symbolics programmers, using Symbolics software development environment, were much more productive than industry norms. Symbolics writers, though, were using tools like those used in "state-of-the-art" industry writing groups -- a text editor and a batch formatter. As a result, the amount of new software was growing much more quickly than the amount of new documentation.

The solution we chose for the documentation productivity problem was to provide the writers with a productive working environment paralleling that used by programmers. The environment was based on the same concepts as those used in modern software development and in environments that support it:

- Modularity,
- Fast edit/compile/test cycle,
- Incremental compilation,
- Cross-reference tables,
- Build and distribution tools,
- Incremental patching,
- etc.

To manage documentation development effectively, we modularized the documentation. The modules were termed "topics," to pick the most neutral generic label possible. Like program modules, document modules can be any size at all and can "invoke" any number of other document modules. Thus the modularity let us break the document set down into small manageable pieces and the "invoking" concept let us build conventional documents easily from the pieces.

When we built our information base, we made the document topics be "clusters" of information. That is, we designed a structure for holding information rather than for holding literal documentation. Each topic in the documentation consisted of different categories of information. Some of the categories were intended to be part of the final documents and some were for "internal" use only. This approach is very different from the standard notion of a literal text document where everything that you type into a file appears later in a document in exactly the same order and format in which you typed it.

By having information broken down into categories (like "type," "arguments," or "keywords") within modules, we provided the basis for different selection and processing schemes. We could select different categories of information for different purposes and change the ordering or formatting of the categories according to the particular situation. The way we conceptualized the information base gave us a substrate that allowed a flexible design for a delivery interface. In fact, the delivery interfaces built so far do not exploit the document structure fully.

Principles and Goals for Document Examiner

Document Examiner is the online delivery interface for the full Symbolics document set. It was designed to support end users in finding and reading documentation. Document Examiner was designed to support selecting from large documents and reading any kind of material (conceptual or reference).

Topics in the information base are identified by name, both internally and to the end user. Because of

this design decision, all topics in the documentation are equally accessible; the "namespace" is flat. The emphasis in the user interface design therefore is on selecting and managing lists of topic names.

The user interface design for Document Examiner considered the following kinds of issues:

- *Familiarity.* People know a lot about finding information in paper sources. We tried to design Document Examiner's interface in order to take advantage of this knowledge.
- *Complexity.* A document set like Symbolics' is complex due to sheer size as well as subject matter. Some aspects of the interface help to overcome the complexity by reducing the set of things that the user is considering at any one time.
- *Performance.* Once a book is in your hand, using it is very fast. Any online system that competes with a book has to be extremely fast or has to provide really significant added functionality to make up for its lack of performance.

Document Examiner was built as an interface to a document set, not as a network browser. That is, its design choices were made on the basis of the needs of the application rather than on the structure of the related computer science abstraction. This is perhaps the most important factor in its design. It has been accepted because users interact with it in terms of finding and reading topics rather than in terms of tree traversal.

Many design choices were made on the basis of parallels with information finding strategies in paper manuals. In several cases, features were added because users pointed out things they could do with paper documents that they could not do "online."

It is worth pointing out that we were *not* trying to produce an "automated book." People doing that use quite a different approach; they start with a paper book and directly automate paper strategies. They have books with tables of contents, page numbers, and indexes. Users scroll through indexes to find the page number of what they want and then issue commands in terms of page numbers. Those kinds of systems have retained the surface features of paper rather than figuring out the essential utility of paper and retaining that.

Additionally, we were not designing a "tutorial." Document Examiner is an interface for user-initiated searches for material. Some of the material is explanatory, conceptual, or procedural. It stops short of any standard tutorial, however, in that we don't provide any directions about what order people "should" read topics in (a hallmark of tutorials) nor do we provide any facility for interaction with the material in the books or evaluation of the users' interaction. (Emilie Young explored the interactivity issue with her summer project on active examples; see section 6.1.)

We tried to determine aspects of paper that were important and, where we could, provided interface aspects to help give people the same benefits:

- *"Feeling for where you are."* In a paper document, you get information about context from the size of the book and where you have it open. The overview feature was designed to give people this kind of feeling for local context. It tells you a little about the document structure surrounding a particular topic, its keywords, a short summary, and so on. It can be used for deciding whether a particular topic is in fact what you want to read or as a jumping off point for finding further material related to what you just read.
- *"Feeling for where you have been."* The bookmarks feature was designed to keep a history of what a user had read. It is the basis for the "private documents" feature, which provides many of the same features as real physical bookmarks.

- *Direct access.* People who know a manual well can find a topic they want by just opening the book to the right place. In the online equivalent, Document Examiner users ask for a topic directly when they already know what they want. They are not required to go through any structural manipulations to get there. Topics are identified by name, not by position in a hierarchy.
- *Looking nearby.* In conventional manuals, you often expect to find related material "nearby." A simple but effective information finding strategy is simply to leaf through nearby pages and see what turns up. Document Examiner provides this kind of capability through the overview feature.
- *Skipping around.* In a paper manual, you can keep a finger stuck in one place while you flip to another spot to check out a reference. In Document Examiner, cross-references made in the text are really mouse-sensitive links to other topics. You can follow such a link and still return to the exact spot where you were reading. The bookmarks facility helps a user keep track of any number of such side excursions.
- *Locating likely material.* In a properly indexed book, starting with the index is a good strategy for finding information relevant to a question. Document Examiner provides several "look in the index" commands for finding topics of possible interest. The user can specify either words or substrings within words (which finds more topics); specifying several words finds topics relevant to all of the words. (This area of the interface was left incomplete; it does not support incremental refinement of lists or other kinds of logical combinations of keywords (like "A or B but not C"). This aspect of the design could be much improved by using concepts from RABBIT.

6. Emilie Young: Using the Full Power of Computers to Learn the Full Power of Computers

Though the Document Examiner is a powerful tool for information retrieval, the textual information is still constrained by the limitations of the printed page. Symbolics had addressed some of the issues involved in finding information, I wanted to address some of the issues involved in understanding and using that information. One general goal in this context should be: to change the Document Examiner (which is currently online written text) to an encyclopedia, providing active examples, hypertext representations, RABBIT/ARGON-like interaction mechanisms, and support for understanding of the information provided.

6.1 Interaction with Documentation Code Examples

- *execute to see results.* The example should be immediately executable, so that one can immediately observe the results of running the code.
- *experimentation.* Users may experiment with the example, change parameters, make additions and deletions, etc. I think it's important that users have the chance to see what situations will create errors and how changes will affect the LISP world.
- *use in own programs.* Users may use the examples, or some form of them, in their own software.
- *annotate and save.* Users may personalize the examples, make notes to themselves about the examples and the results of running them or changing them, and save the annotated examples for further reference.

6.2 STUMPER

STUMPER is a system designed to help create a community knowledge base about the use of Symbolics LISP Machines. It addresses many issues involved in user-centered design:

- *Helping users generate goals.* In a system like Symbolics, with its vast functionality, users need help in finding out about those aspects of the system which are outside the bounds of their mental models. STUMPER introduces users to aspects of the system they may have never encountered before. STUMPER generates goals for the user, following the answer-question paradigm.
- *Learning through experimentation and discovery.* STUMPER aids in helping users extend their mental models by suggesting interesting experiments to run and problems to solve. Ideally, STUMPER will protect the user from any disastrous results.
- *Community knowledge bases.* STUMPER promotes a sense of community by increasing communication and informal information exchange. It gives credit to those members of the community who are particularly helpful.
- *Motivation.* Learners will be motivated to learn more about the Symbolics in the same way that they're motivated to play video games. The learning environment is fun, engaging and provides immediate feedback. The contributors will be motivated by ego. Authors will be credited and will take pride in coming up with a particularly "good/hard/interesting/arcane" contribution.
- *Learning as a Displacement Activity.* STUMPER will make use of time not used by more concentrated long-term activities. It will fill the same ecological niche as reading mail, rwho, etc., while providing informal contact with the rest of the community.

- *Teaching problem solving strategies.* Contributors will guide learners through problem solving strategies. Learners will be able to experiment with specific examples as opposed to abstract ideas.
- *Exploiting the full power of computers.* STUMPER exploits the multimedia capabilities of computers.

6.3 Document Examiner Analysis

Janet Walker at Symbolics has written some data gathering routines for their on-line documentation facility, the Document Examiner. The data gathering is essentially transparent to the user, capturing information on "real work" use of the facility. If the data gathering is activated, the Document Examiner will keep a log of user's interactions including:

- user,
- search string or documentation title,
- command,
- `t` or `nil`, depending on if the command was completed successfully,
- date and time of interaction.

I have extended the program to allow users to enter their goals to the log file.

The data gathering has been activated at the University of Colorado for the past three months. Approximately 800 interactions have been logged. A brief, initial analysis of these interactions has produced some interesting results:

1. *Frequency of Commands.* There is an approximately even split between the use of browsing and searching commands. There is another approximately even split between the number of commands to locate information (browsing and searching) and the number of pieces of documentation examined. Looking at the types of documentation examined, 58% explain conceptual ideas, while 42% are explaining actual code objects. Of the code records, 82% are documenting functions, as opposed to options, init options, types, variables, methods, presentation types, or flavors which combined account for the other 18%.
2. *Pairs of Commands.* To qualify as a command pair, the commands must have been issued sequentially, by the same user, with less than two minutes elapsed between commands. After inspecting the raw data, two minutes seemed a reasonable time constraint for almost all commands, reading conceptual documentation may be an exception.

   This analysis produces a sort of state diagram of users' interactions. For example, from the BROWSING state, 40% of the time users issue another browsing command, 31% of the time they look at conceptual documentation, 10% of the time they look at code object documentation, 5% of the time they issue a search command and 14% of time they stop the session.
3. *Unsuccessful Searches.* This is a list of all searches which produce null results, about 4.5% of all commands.

Future analysis will include differences between individual users, differences between groups of users (Symbolics employees versus CU students), and in-depth inspection of separate user sessions.

7. Steven Weyer: Dynamic Books and Computer-Supported Encyclopedias

I introduced my talk by discussing a recent PBS broadcast of "The Day the Universe Changed." As Gutenberg supplanted the reliance on old people's memories and on monks copying scrolls, books evolved to present information; with larger books and collections of books, tables of contents and indexing had to be developed. Somewhat analogously, we must evolve new forms of presentation; with CD-ROM especially, new ways to search must be developed.

Hypertext at its simplest means "non-linear text" (though now it also includes graphics, video and sound -- hypermedia), such as we find in footnotes, hierarchies, and crossreferences. Vannevar Bush envisioned the first electronic hypertext system (Memex); Ted Nelson popularized it (Xanadu), especially multiple versions of documents and the term "hypertext"; Doug Engelbart built one of the first (NLS; Augment); recent microcomputer "idea processors" such as ThinkTank (and More -- its successor) could be considered hypertext systems -- clearly some psychological studies could be done on whether it is "better" to read or write in this structured way; Intermedia at Brown and NoteCards at Xerox are powerful current research prototypes.

I showed slides of two electronic books from my own work that exhibited hypertext qualities: one, a dynamic history book implemented at Xerox in Smalltalk-76, that focused on organizational structure (e.g., hierarchy) and search features (index expansion, subject neighborhoods, bookmarking) -- see [Weyer 82] for further details; second, an electronic encyclopedia implemented at Atari in Zetalisp, that focused on dynamic objects: article crossreferences, conversion of measurements, expansion of abbreviations, text to videodisc references, and animated simulations (e.g., harmonic motion of spring) -- see [Weyer, Borning 85] for further details and other hypertext references. I showed videotapes of these systems at break times.

I discussed Guide, a Macintosh-based hypertext&graphics browser and editor, that featured glossary buttons (for defining), cross reference buttons, and expansion buttons (normally used for expanding/collapsing levels of hierarchical detail); I demonstrated this product at break times.

I summarized some information needs that AI could help address:

- flexible tools and adaptive partners, intelligent filters and guides;
- guidance in refining and browsing: there is ever more information and with hypertext, there will be too many links;
- seamless access to multiple systems: login, command language, search;
- user-orientation in information systems: users are often only an "apple-pie" issue;
- explicit "meta-information": source, bias, user annotations, purpose;
- representing and leveraging user goals: not only how to find information but also what the user wants to do with it;
- learning in addition to retrieving, asking questions in addition to obtaining answers;
- changing user needs: what is different? new? important? unusual?
- represent articles as knowledge networks (e.g., CYC);
- concept retrieval rather than just words;
- document examples and natural language in addition to boolean queries.

Many of the issues I raised are summarized in a draft paper "As We May Learn" that I distributed to

workshop participants [Weyer 87]. In brief, I argue that we must look "beyond Memex" and beyond just browsing, not only to authoring aids (in constructing links semi-automatically) but also to aids for guiding (among many alternate paths). Browsing breaks down when there are too many undifferentiated links. Static text and graphics must be supplanted by knowledge. "Hypertext browsing" should be augmented by "hyperknowledge assistants." More than a mere information system, it must become more problem/solution oriented to address the goals of different learners and users.

Regarding "production paradox" (see section 2.4): It isn't so much that people do not want to learn, but that they do not want learning to be inefficient, unnecessarily distracting, or painful. Agreed, if I just want to get my memo printed, I don't want a treatise on Western Civilization. Most computer systems are not designed with exploration and human error in mind. If human actions can cause catastrophic results (that cannot be undone) or confusing situations for which the system can offer no explanations or guidance, then frustration is certain. Perhaps we should turn the question around and put the blame on the system: why don't systems learn what we are trying to do (rather than us having to learn how to use it).

Regarding "reality is not user-friendly" (see section 2.2): An interesting test of the Macintosh user interface will be with larger, more complex systems, e.g., the Finder interface may not scale well for a large UNIX file system (already there is a desk accessory that allows you to search for a file name in the hierarchy). Complex user interfaces that need to be learned may not be a bad thing, if the system is to be used frequently (Engelbart argues that new kinds of tools (such as NLS) may require training for users to become proficient); simpler metaphors may provide easy initial learning but get in the way of more complex activities. If we had good programming by example systems, could we dispense with most shell scripts?

8. Brian Phillips: Machine-Supported Information Management for Collaborative Work

8.1 Introduction

What does an engineer need to know to to be an effective and efficient member of a design team? Clearly he or she has to have the technical knowledge to create the artifact. Assistance is available in the form of knowledge-based CAD tools (e.g., [Mittal, Dym, Morjarla 86]). Configuration management tools (e.g., [Delisle, Schwartz 86]) help in keeping track of the versions of an evolving design. An ability to fully exploit these tools is also advisable. Much of the work reported at this workshop is concerned with this kind of user assistance. The view is that assistance should be a collaborative effort between human and machine: "the intelligence of a complex computer system must contribute to its ease of use."

However studies have shown that an engineer can spend 60% of his or her time on non-design activities. Preparing reports, scheduling and attending meetings, planning, etc. are a major drain on time. These activities require tools to process the material (word processors and the like), and, again, assistance in their use is appropriate. In addition, the information manipulated with these tools can benefit from computer-based management. The average office gives an insight into the diversity of information in use. There are filing cabinets for documents. Card boxes contain technical references. Rolodex's have phone numbers and addresses. Blackboards sketch emerging ideas. Rosters are tacked to walls. Calendars show past and future appointments. Reminders are taped to the edges of shelves. Not surprisingly it is often difficult to locate information quickly within this amorphous environment. Moreover, as a member of a team, an engineer collaborates with colleagues. Project disfunction can arise because of poor information sharing. Critical data may reside in the mind of a person who is not accessible. A change may be made but all parties affected by it are not informed. To facilitate information management in a collaborative environment for a broad spectrum of information we are developing an Intelligent Project Support Environment (IPSE).

The initial IPSE is essentially a passive system. It provides the means to store and retrieve many kinds of information. The hypothesis is that with efficient access to information, more time is available to a user to apply his or her intelligence to the creative aspect of a task. A practical, highly active assistant is beyond current technology. It would require the automatic comprehension of documents carrying textual and graphical information, for example. Nor do we have comprehensive models of team behavior to implement. Hence we have placed an initial emphasis on interface technology to facilitate direct user control. Self-reporting of tasks will guide the development of an active system. We also are undertaking naturalistic studies of communication and decision making to assist model building [Reder 85].

At the user level three components of the system are being developed: a project encyclopedia, an information communication system (a mailer), and a time management system (a calendar). A fuller description of the early system can be found in [Phillips, Staley, Gold 87].

8.2 Intelligent Project Support Environment

An information management system supports information organization, capture, and access.

Information Organization

The *Project Encyclopedia* is a globally-accessible, logically organized repository of project information. It is a file rendition of a semantic network [Phillips 78] and is built on top of a hyperdata system [Delisle, Schwartz 86]. The hyperdata has a graph structure with arbitrary attributes and values on nodes and links. Hence implementation of semantic net arc labeling is directly possible.

The encyclopedia provides the means for representing corporate organizational structure, programs, plans, design ideas, team members and their skills and assignments, technical references, meetings and their minutes and agenda, colloquia announcements, etc. Figure 8-1 shows a portion of the encyclopedia in which the structure of a program and a project is represented. These are abstract schemata of which actual programs and projects are instantiations. Each node can also store arbitrary information, presently text or figures.

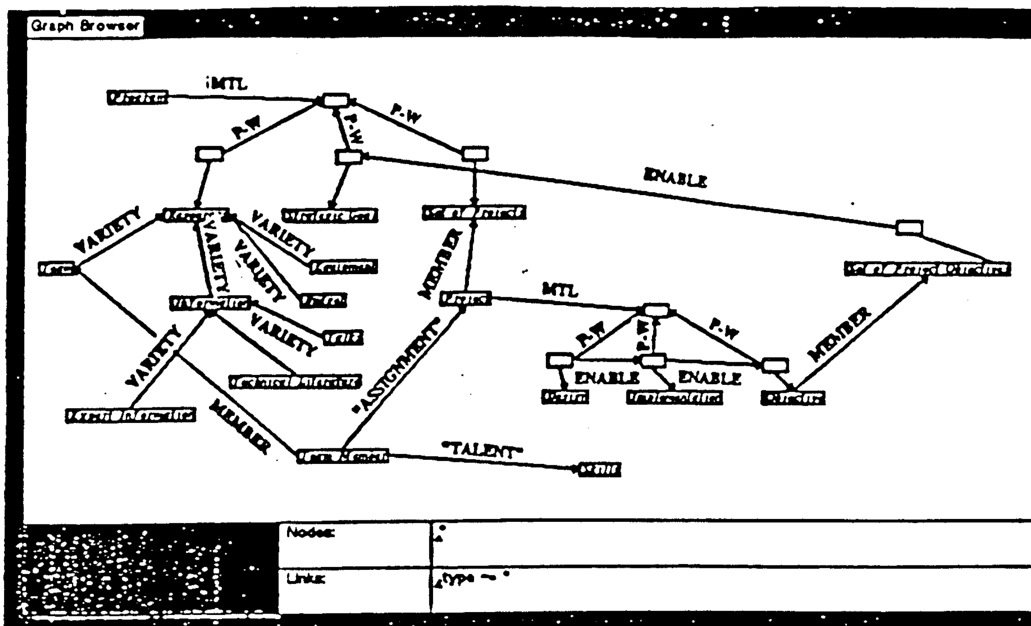


Figure 8-1: Project Encyclopedia

The primitives of the representation have to avoid a bias toward any one topical area. The cognitive organization of Kintsch, reported at the workshop, is relevant to our design. Although the encyclopedia contains much textual material, its volume and diversity preclude adopting a purely textual database approach like Symbolics' Document Examiner. The architecture of and information sharing in STUMPER (see section 6) are closer to our philosophy. The representation of the logical structure of arguments, following EUCLID (see section 14), could be an interesting tool in evolving a design.

Information Access

Access to the encyclopedia is not direct but through views. Many views are possible. A project browser, a calendar, and a mail system are current examples. The inspiration for many of the interfaces comes directly from Smalltalk [Goldberg, Robson 83], the language of implementation. Our present access mechanisms are direct selection within a specific environment, e.g., of an entry in a calendar. The incremental query style of RABBIT, ARGON, and HELGON (see section 19 could also be useful additions.

Project Browser

A general topical view is given by a project browser which hides the formal structure of the encyclopedia. For example, the encyclopedia stores the proposition that "a manager assigns a team member to a project" but the browser does not show the "assign" predicate instead it juxtaposes the team member and the project, Figure 8-2. The information stored at a node in the encyclopedia can be retrieved and displayed. Figure 8-3, for example, shows the image of a whiteboard. The encyclopedia provides a single framework for many kinds of information that manual systems spread over many devices: organization wall charts, 3x5 cards, address wheels, file cabinets, whiteboards, etc.

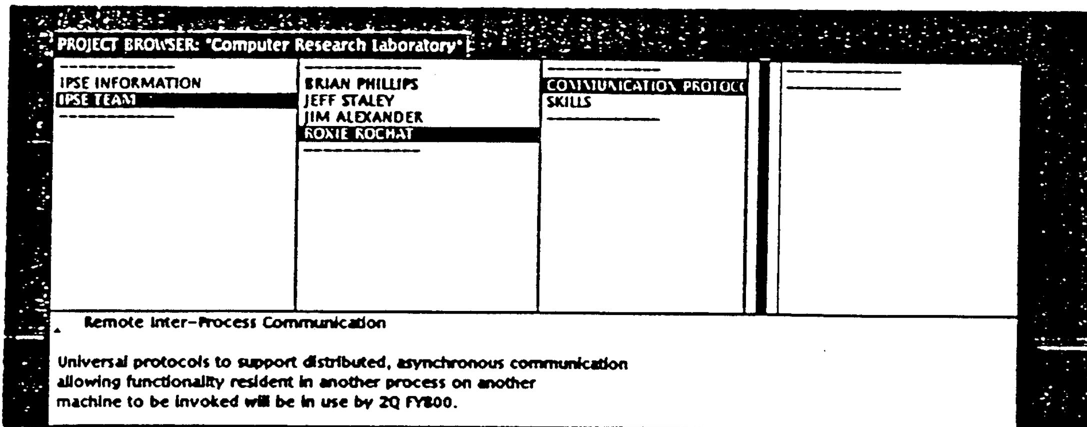


Figure 8-2: Project Browser

Time Management System

Time management is superficially personal calendar management. The user interface models a paper calendar (Figure 8-4). One can zoom from month to daily views. Zooming filters information: on a month view clock time is not shown, and "unimportant" events are omitted. An entry can be inspected to give access to associated text, the agenda of a meeting or the abstract of a talk, for example. We are making the calendar a view onto the temporal dimension of the project encyclopedia.

In many cases, information can have multiple access paths. The project browser and the calendar

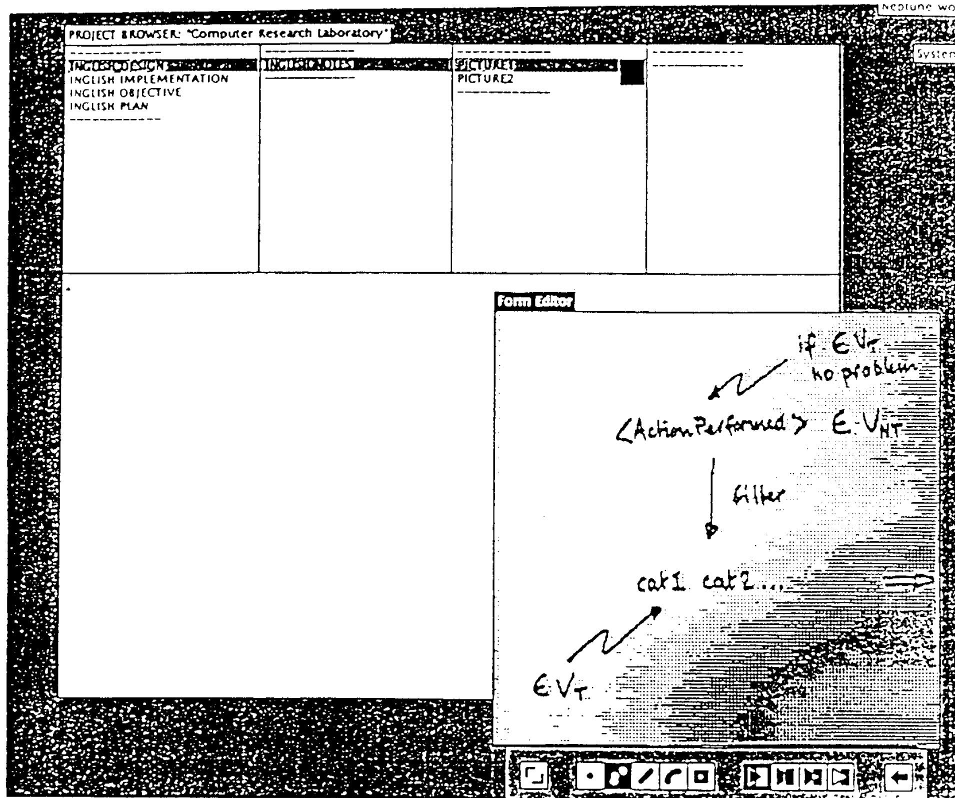


Figure 8-3: Image Stored in Encyclopedia

exemplify this feature. A topical descent into the encyclopedia via the browser is one form of access but the user may also remember the meeting in which the issue was discussed and choose to retrieve it via the minutes of a meeting associated with a calendar entry.

The calendars of other people can be displayed. Thus their availability can be found quickly, avoiding the "telephone-tag" that often accompanies attempts at schedule coordination. Our next step is to investigate automatic scheduling. If the calendar is to be an accepted surrogate in scheduling it has to know more than free time slots. It must know the preferences of its owner. To this end, it is being augmented with a rule system to represent such preferences.

Information Communication System

In addition to its use for transferring information the mailer will also provide means for storing messages as parts of conversations [Winograd, Flores 86] in the encyclopedia (Figure 8-5). And inasmuch as the encyclopedia is the storage medium, any encyclopedic entry can be "mailed." Hence the mailer is another view on the encyclopedia.

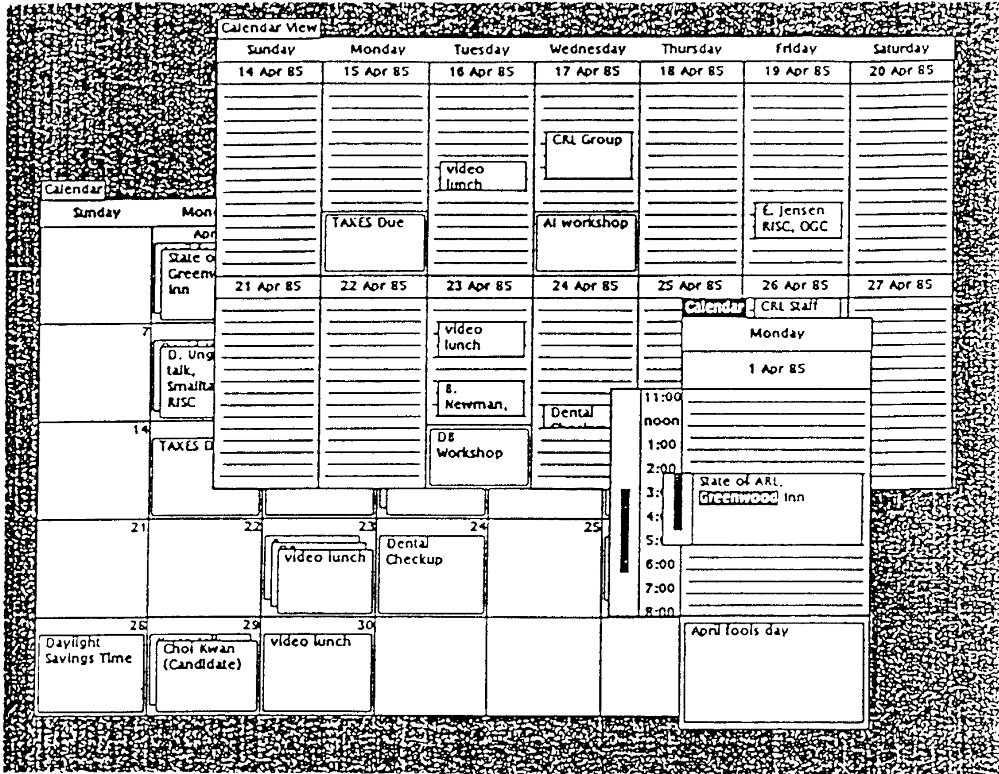


Figure 8-4: Calendar Interface

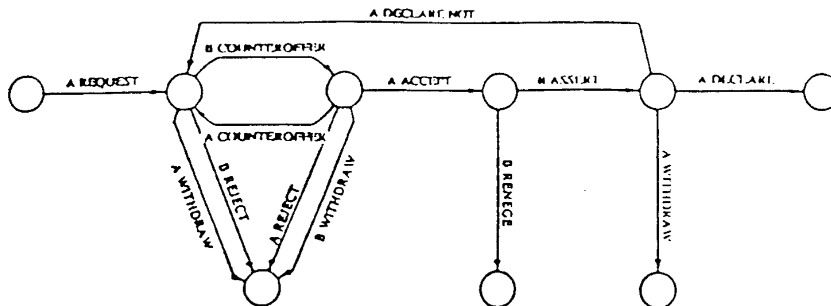


Figure 8-5: Model of Task-Oriented Communication

Information Capture

Ensuring that information is entered into the encyclopedia is crucial to the success of IPSE and many methods of encouraging this are sought. The capture of information involves its physical input as well as its placement in the encyclopedia. Many innovations are taking place in input technology. We have been investigating optical character reading and video image capture (Figure 8-3) and intend to explore others that come to market. Each of the components of IPSE provides a context for entering information to assist in determining its logical placement. However, any interface may not give a new entry the full range of links into the encyclopedia and hence we must also investigate means of classifying raw input, for the present textual material.

Project Browser

Through the browser, data is added *by example*, that is, the user selects exemplars, and the underlying encyclopedic structure is used as the template for the new knowledge. This requires that there be a certain amount of initialization of the encyclopedia by an expert or that an adequate set of abstract schemata can be provided. For example, the encyclopedia has a schema defining a project as having a design, an implementation, and an objective (Figure 8-1).

Time Management System

Making new entries, moving entries, and scheduling repetitive events use the mouse and menus. All calendar events are explicitly associated with the calendar's timeline which enables their placement along the temporal dimension of the encyclopedia.

Information Communication System

In addition to being a convenient method for bringing items to the attention of others, the communication system is intended to strongly encourage the electronic transmission of information so that it can be captured by the encyclopedia as a byproduct. At present there is a mail system [Gold 86] enhanced by rules to give users filtering and generation capabilities [Malone, Grant, Turbak 86].

A message has many fields which can trigger rules. One problem with such a system is that it apparently requires free text understanding to match field entries with clauses of rule conditionals. This is overcome by providing a *Communications Directory*, Figure 8-6, from which entries can be selected. It does not preclude free text entry of entries not to be found in the directory. There is also the problem of entering the rules themselves. This has been simplified through the use of INGLISH, a restricted natural language system [Phillips, Nicholl 86]. INGLISH can guide the creation of a sentence in the rule language with terms being selected from the communications directory.

The ability to store mail within conversations also provides the conversation as a context for creating new messages.

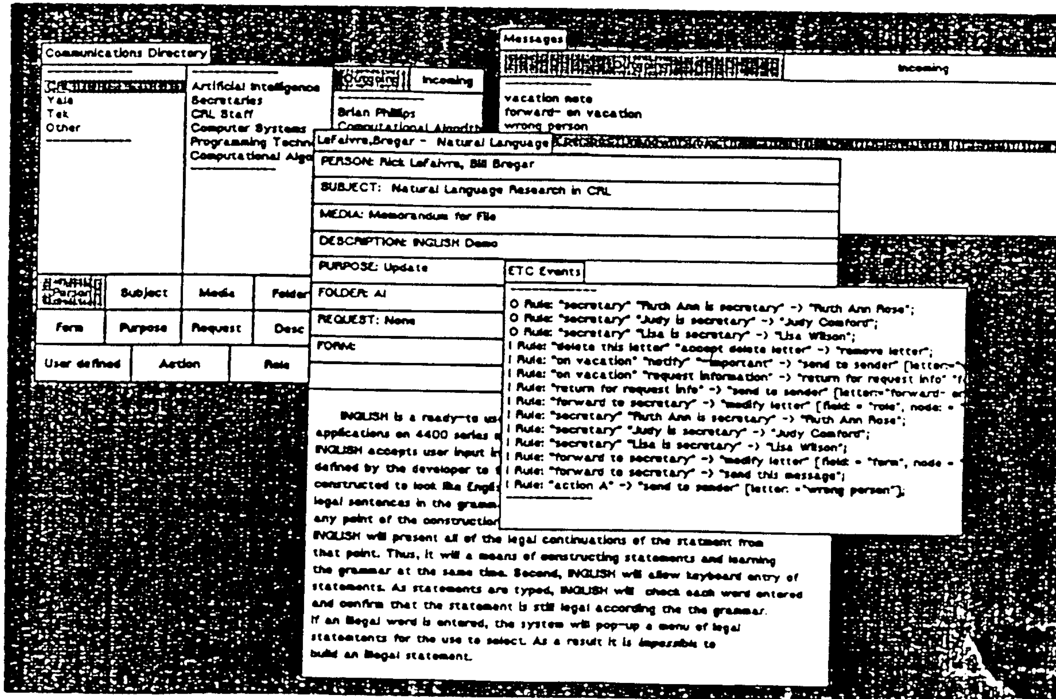


Figure 8-6: Mail Interface

Text Classifier

For the textual information, present language processing techniques offer structural analyses that produce a primitive representation, for example, a conceptual dependency structure [Schank, Abelson 77]. This approach is unfeasible for large amounts of project information. We envisage hybrid approaches which are likely to include keyword searching [Kelly 75], conceptual abstraction [Phillips 78], and marker-passing [Hendler 86].

8.3 Conclusion

Can IPSE reach a threshold of user acceptability? Technically, can any extra effort or discipline to use it be more than off-set by the overall benefits? Socially, can there be a consensus to use it, otherwise it is unlikely to have an adequate information base.

The knowledge in the encyclopedia opens up immense possibilities for decision support systems for the effects of change, task assignment, task sizing, project scheduling, and more. The integration of IPSE's symbolic tools with quantitative tools is yet another avenue to be explored.

Implicit in our intention to extend IPSE to become an active assistant has been an assumption that a

project is a rational, cooperative venture. Discussions at the workshop have called this into question with important consequences. A computational assistant requires that the user explicitly represent in the system the knowledge and procedures that are to be used. However a user will be understandably unwilling to create an entity that requires revealing his or her selfish goals and ambitions that, in reality, do exist.

9. Erich Neuhold: Databases and Information Bases, Architecture and Integration Issues

When constructing an integrated knowledge-based system for some application, e.g., for information gathering and publication purposes many research issues still have to be resolved. After presenting a possible architecture of such a system we describe how, given pre-existing public or private data (information) bases, a dynamic integration of these databases can be achieved. In due course such a system will offer a homogenized single information base view to (a group of) users.

Based on concepts of semantic data modeling in the database area, of knowledge representation in artificial intelligence, and of object-oriented programming in the programming area we develop principles for recognizing and representing interschema relationships in a global scheme which is dynamically evolving during access to a heterogeneous system.

9.1 Introduction

In the last few years information systems have grown in importance in nearly every area of data processing. Whether it is in the office, in the factory, the educational institution, or the home, our work (and pleasure) more and more relies on the availability through one or the other of the many available network facilities.

In order to be able to discuss the state of the art and future research and development directions in this field of distributed information systems we shall first introduce a reference architecture for a system which may be concerned with information gathering and publication support (Figure 9-1).

The illustration is supposed to describe a single (logically integrated) application area including for example the workstations required for a scientific publication process where scientist, author, secretary, referee, editor, publisher, printer, and distributor all have to work together to arrive at a paper or electronic version of some conference proceeding.

In principal such a system can not stand alone but has also to be embedded in some environment that itself is an information system again specially created for the purpose of designing, building, controlling, and servicing distributed systems. From system analysis and design tools, from programming languages and programming environments to network design, control, and servicing facilities a whole spectrum of components is needed herewith will allow us to construct and maintain the complex distributed information-oriented application systems of the future. This design, control, and service system itself will have an architecture like the reference architecture, i.e., a layered, multifaceted, integrated structure and therefore will not be investigated further in this report as it can just be seen as another application system geared specifically to the building, control, and servicing of distributed information systems.

The distributed information system itself could be seen as a four level structure (Figure 9-1) where the lowest layer would be represented by basic services today mostly found in operating system kernels, and the highest layer represents all the programs, packages, and tools specifically provided for an application.

Layers 2 and 3 contain all the many components that have to be (respectively should be) provided to first simplify the development of applications and secondly make them easy to use for the whole spectrum of

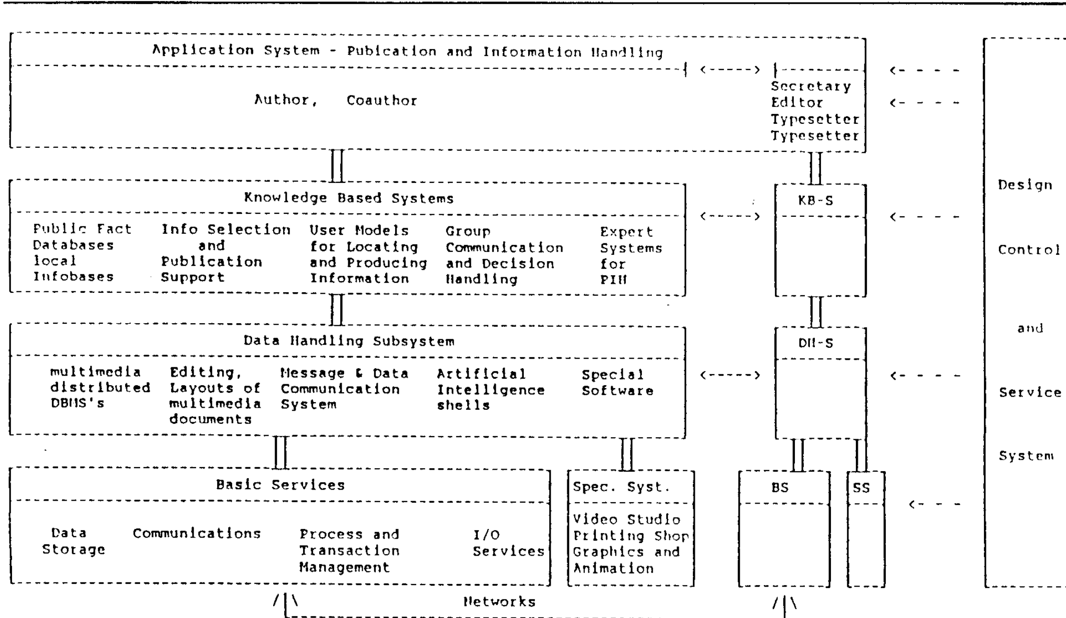


Figure 9-1: Reference Architecture for Advanced Information Systems

expertise usually found among the different classes of system users.

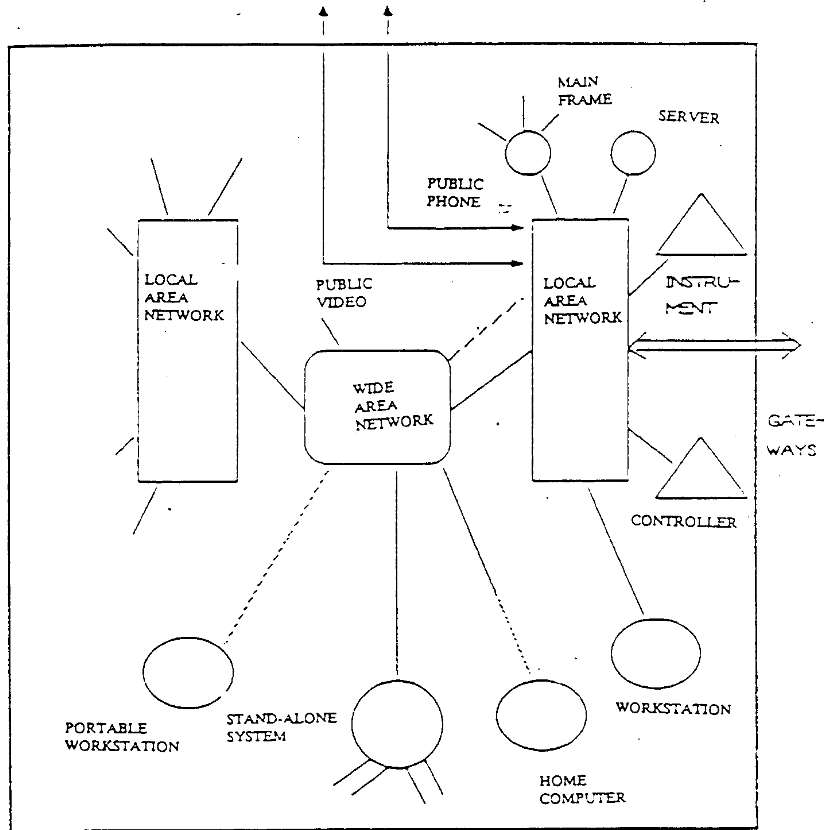
Layer 2 is concerned with components that are "data"-based whereas layer 3 contains the "knowledge"-based system parts. Data here means simple or complex structures that are maintained through structural descriptions and data type classifications (e.g., an integer field, a character string, a forms layout, a video clip, an image, an audio trail) and where operations in general only utilize and control these structures but are unable to work with the information actually "contained" in these data. For example it will in general be possible to write a database operation that allows to add the age and the salary of a person together even though the result of this operation has no obvious meaning in the real world. The knowledge based layer in contrast will (through knowledge representation techniques) directly deal with the meaning of the data stored, and operations on this layer will take this meaning into account when executing. Here an operation adding age and salary would not be acceptable as the resulting value could not be interpreted as knowledge (or information) about the real world that is modeled in this layer.

The model for Advanced Information Systems is based on the abstraction of data and its semantics. In this sense it can be viewed as hierarchical; in the hierarchy upwards, data and operations present increased semantics.

The starting point in the model is the assumption that future information systems will evolve to knowledge based systems, where users will be lead by the system itself towards user specific problem solutions.

Each level in the model encapsulates the necessary integration of the different (heterogeneous) components at that level.

The distributed information system could be seen to be installed on a complex network incorporating multiple local area networks into long haul networks as illustrated in Figure 9-2. Note that the home computers and portables also should be seen as belonging to the system even though one cannot assume that they will be connected to the system proper at all times.



A Single (Distributed) Logical Information System

Figure 9-2: An Advanced Computing Architecture

In our institute we shall install such a network and shall try to build an integrated system for information gathering and publication purposes. Many research issues have to be attached and solved before the system will actually represent the vision offered here, and our institute will have to concentrate on a few of these aspects and rely on others and cooperation with them to solve the problems we shall not approach. In the following we will concentrate on a problem frequently encountered during information gathering, namely the integration of already existing information bases respectively databases.

9.2 Dynamic Integration of Heterogeneous Databases

Database systems have been developed to manage data of several related applications on a single mainframe computer. To achieve this goal the data views of the various applications have to be integrated into a global schema.

Distributed Database Systems like POREL [Neuhold, Walter 82; Stonebraker, Neuhold 77] distribute the data over different sites and are generally designed top down starting with the global schema.

Multidatabase Systems may be created by integrating single-site database schemes from bottom up into a new global schema [Dayal, Wang 84] or by assuming that there is basically no global schema like MRDSM [Litwin, Abdellatif 86] and the federated architecture [Heimbigner, McLeod 85].

Dynamic Integration of Heterogeneous Databases is based on capabilities of handling heterogeneous databases containing different kinds of objects, like texts, pictures, and structured data. Its objectives are a dynamic knowledge acquisition process such that the global schema is gradually evolving during the use of the database, high flexibility in handling schema changes of the underlying databases, and ease of incorporating new databases into the system.

Dynamic Integration of Heterogeneous Databases is based on the following aspects of the Advanced Information System Architecture presented in section 9.1 (see Figure 9-3):

1. The *Operating System Kernel* provides for individual data storage, process management, and communication in a distributed environment, i.e., we do not assume a distributed operating system but individual systems on the different nodes.
2. The *Data Handling System* accepts physical differences between data, i.e., characters, numbers, bits, and more complex structures assembled from them up to and including database systems. Again we assume the existence of individual databases and Data Handling Systems and not a single integrated facility.
3. The *Knowledge Handling Systems* are concerned with data semantics, i.e., capturing the meaning of the data in the different databases respectively Data Handling Systems and describing the interrelationships between these components in the system dictionary. That is, in our approach we assume the integration of the different components of a system to take place at this semantic level and not at a lower level as in the case for example in a distributed database system.
4. The *Application System* combines all the offered facilities to represent the computerized support for the problem solving user.

The System Dictionary

The System Dictionary (Figure 9-3) consists of three components, each representing one facet of the information base structures. The bottom layer of the knowledge handling system is the Logical Structure Dictionary. It describes the objects and operations of each of the different information bases, the database object classes. This description is independent of the special physical representations: graphic, text, and structured databases are treated in a coherent and homogeneous manner as abstract objects. It is the purpose of the underlying Data Handling System to carry out the mapping of the logical structure onto the data structures of the database and vice versa.

The Enhanced Logical Structure Dictionary contains knowledge about intraschema relationships which

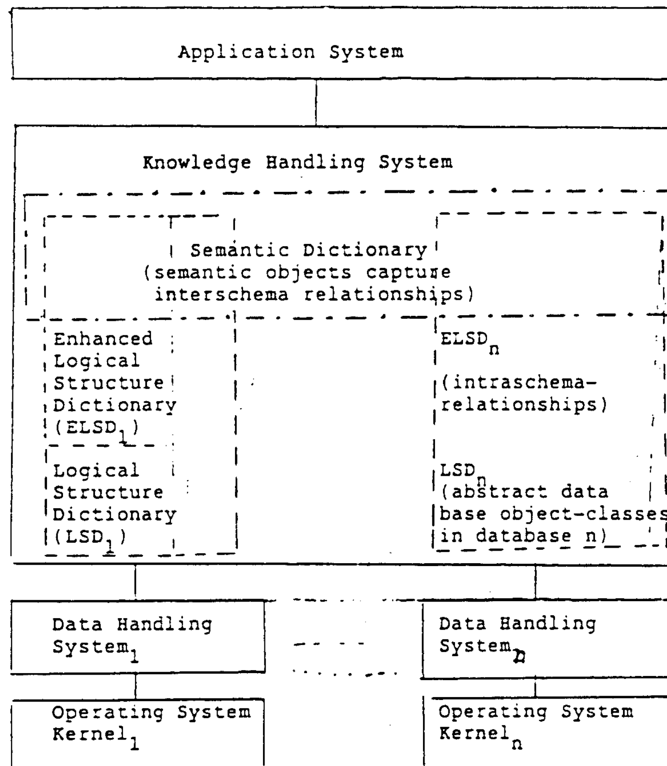


Figure 9-3: Information System Architecture for Database Integration

are not explicitly captured by the Logical Structure Dictionary, but may be needed for the construction respectively representation of interschema knowledge.

The Semantic Dictionary describes interschema knowledge, i.e., contains rules about the relationships between objects in the different (Enhanced) Logical Structure Dictionaries. Semantic objects tie together corresponding database objects which carry information about the same real world entity stored in different databases.

The Method

Our approach is based on concepts of semantic data modeling in the data base area [Neuhold 86], of knowledge representation in artificial intelligence [Fikes, Kehler 85], and of object-oriented programming in the programming area [Stefik, Bobrow 86].

The object-oriented information base descriptions in the system dictionary are based on the semantic data model THM [Neuhold 86] which is modified and enhanced with

1. a *tight coupling* of operations to objects,
2. associated *rule-based* knowledge.

In particular *system statics* are represented by the main static concepts of semantic data models [Schrefl, Tjoa, Wagner 84]: aggregation, generalization, and grouping (cover aggregation).

System dynamics is to be realized by message passing concepts [Stefik, Bobrow 86]. Messages sent to objects will invoke methods, i.e., operations, which are tightly coupled to the objects.

System behavior is determined by behavior rules associated with objects. Active behavior rules allow objects to act as active agents, i.e., invoke other object activities by sending messages. Passive behavior rules determine an object's behavior when invoked by another message.

The Integration Process

We assume the individual database descriptions to be given in the Logical Structure Dictionary and the necessary mappings into and from the underlying databases to be taken care of by the Data Handling System. The integrating process then consists of two phases:

1. In the Preintegration Phase local database administrators add their knowledge to the Enhanced Logical Structure Dictionary.[3]
2. The Integration Phase is an open ended process which starts with the first information retrieval request against the heterogeneous system.

Whenever an information retrieval message can not be handled by an object itself a *dynamic knowledge acquisition process* is initiated: An *inquiry message*, which carries basically the intention of the original information retrieval message, is forwarded and broadcast according to the passive behavior rules of the objects. Message forwarding is done within objects of one database, message broadcasting is done over database boundaries.

The inquiry message returns with a set of *message passing plans* - i.e., metadescriptions of a sequence of messages sent to objects - which describe how to answer the original information retrieval message, and a plan expert assists the user in plan analysis and plan synthesis to come up with a final message passing plan. The final message passing plan is recorded as a semantic attribute of a semantic object and used to answer future information retrieval messages of the same type.

Global broadcasts invoke the integration expert whose task it is to detect new interrelationships between objects in different databases such that the knowledge about semantic objects is steadily increased during the use of the heterogeneous database system.

Remark: A more detailed presentation of these issues is contained in [Schrefl, Neuhold 87]

[3]Note that only intraschema knowledge is required.

10. Walter Kintsch: Knowledge Assessment and Knowledge Organization

As far as the psychological research is concerned that has been proposed as part of the ARI project, the main achievement of the workshop was that it provided us with a much deeper understanding of (a) what the questions and approaches in the field of of user-centered computer design really are, and (b) how psychological work work fits into that. There is clearly a huge gap between basic research and applications in cognitive science, which is neither surprising nor objectionable. What this workshop has done was to help bridging that gap, and provided us with a better understanding of exactly why building such bridges would be a worthwhile enterprise. Below, the presentation by Kintsch & Mannes at the workshop will be briefly summarized, followed by suggestions how this research can be coordinated more closely with some of the system design work. Finally, it will be pointed out how the work Althea Turner is currently engaged in could be used to broaden the scope of our project to deal with several of the issues raised by some of the outside visitors at the workshop.

10.1 Generating Knowledge Structures from an Associative Memory

This work is an outgrowth of the research on knowledge use in discourse comprehension (e.g., [Kintsch 87]). Two claims have been made (though they are not uncontroversial): that knowledge structures such as semantic nets, schemata, frames, and scripts play a crucial role in human knowledge use, and that such structures do not reflect the fixed organization of human memory and/or knowledge, but rather are generated from a more loosely organized, flexible system in response to particular task demands.

The question then becomes what is that "loosely organized, flexible system" like, and how are such structures to be generated from it? A proposed working hypothesis is that the human memory/knowledge system is an associative net with concepts and propositions as nodes, and links among the nodes which may facilitate or inhibit the spread of activation.

In a series of studies [Walker, Kintsch 85; Kintsch, Mannes 87] we have explored how people create such knowledge structures, and how this process could be simulated with an associative net as a base, and with models from the memory literature providing the retrieval mechanism. Walker & Kintsch asked subjects to list category members ("name all the makes of cars you can think of"), and simulated the observed behavior by assuming (a) that memory retrieval from an associative net is adequately described by current models (specifically [Raaijmaker, Shiffrin 81]), given a particular retrieval cue, and (b) that a task specific control process is used to construct and reconstruct this retrieval cue. Kintsch & Mannes similarly studied how subjects generate scripts ("what normally happens when one goes grocery shopping"), and found that a model very much like that devised for category naming could account for their data: the main difference was that subjects appeared to use a rather different control process to construct their retrieval cues, employing local temporal information to guide them (which, of course, is not available in the category naming task). (A somewhat more elaborate description of this research is given in section 11).

As part of the ARI project, we have extended this research to a third domain: tasks involving computer use (e.g., "suppose you receive a request in the mail to send someone a certain part of one of your papers"). These tasks were chosen so that experienced computer users would engage in a minimum of planning, and rely primarily on the retrieval of learned procedures (methods). Once again, we had subjects provide verbal protocols of what normally happens in such a case. These protocols show many

similarities with our script generation protocols. Indeed, the Kintsch-&-Mannes approach can be generalized to simulate verbal protocols in the present situation. The memory structure, and the retrieval mechanism remain the same (the content of memory - the propositions making up the associative net - are of course different), only the control process had to be changed which governs the construction of the sequence of retrieval cues by means of which subjects activate the knowledge necessary to perform the tasks required of them.

What we find is a rich net of propositions encoding the user's knowledge about procedures. The central nodes in this net are at a relatively high level (we call them "methods"), that is, not directly linked to procedures (productions). Retrieval occurs associatively around these nodes, with information about consequences and prerequisites guiding the retrieval process from method to method. At certain points this declarative knowledge is anchored to the world of action: certain propositions are tied to productions, and productions act on the state of the world. Thus, the state of the world (the task specification) activates a declarative network which guides the retrieval of the knowledge necessary for performing the task. The action of performing the task, however, keeps changing the state of the world, so that new knowledge sources become relevant at various points in the action sequence.

Our immediate goal is to work out this simulation of how experienced users retrieve their knowledge of methods to solve standard problems. For this purpose we need to construct a model (in the form of an associative network) of the experienced user's knowledge about these methods and tasks. From there, we plan to move to novice or inexperienced user's, model their knowledge and their ways of solving these problems. We expect to see significant differences both in the knowledge and the solution strategies between experienced and novice users. The exploration of these differences will be our first major goal. Only after we have a clear picture of what these differences are can we proceed to the next tasks: finding ways to support the novices' performance on these tasks.

10.2 Relation of this Work to Specific Design Projects

The projects of Christian Rathke and Hal Eden: *Reorganization of Knowledge Structures* (see section 17), and of Helga Nieper: *Information Retrieval by Reformulation - From ARGON to HELGON* (see section 19) are good examples of research upon which the theoretical work discussed above might have some impact in the future. Rathke & Eden suggest that we live with inadequate structures because the effort to change the existing structures is too great. They explore ideas for how systems as well as databases could be reorganized more easily. Clearly, what is needed is an organization which is flexible, and which readily adjusts to the ever changing demands of specific tasks. This is precisely the question which our simulation models approach. It is not obvious what such a flexible organization should be like; it is not obvious how the desired structure can be generated on demand from such a flexible organization. It is obvious that the human mind achieves this routinely. By studying how people do it, we perhaps can give hints or inspiration to system designers as to how they might organize their systems or databases. At present, because our theories are still in the infant stage, we can do little more than suggest to Rathke & Eden how to reformulate their questions, how to think about their problem in a somewhat more general way (generating global structures from only locally organized components) - but our aspiration is to do much more than that!

Helga Nieper's ARGON/HELGON project is another point in case. RABBIT-like systems have the goal of approximating human retrieval capacities in a fuzzy environment: we often don't know precisely what it is

that we want, but manage to think of it anyway. RABBIT was a first try, based specifically on psychological theory ([Williams, Hollan 81] took the idea of retrieval by reformulation from protocol studies of subjects' retrieval efforts). The direction is fine, but it doesn't go as yet nearly far enough. If we have a better understanding of how people retrieve information in the tasks studied here, we can potentially design much more powerful RABBIT-like systems. Once again the theoretical work we are proposing is concerned with precisely that problem; our hope is that soon it will be able to guide system development such as ARGON/HELGON.

10.3 Generating and Updating Knowledge Structures from Text

One thing common to many of the discussions in the workshop was an emphasis on the need of having rich knowledge structures. To provide effective tools, systems must have available a great deal of information - of course it has to be organized the right way, and must be retrievable, but first of all, it has to be there. The task of encoding truly rich knowledge bases so that they can be used by a computer system is not a trivial one. Not much progress will be made until we devise automatic or semi-automatic encoding methods.

Much of our knowledge is in the form of texts. Thus, the question of building knowledge bases from textual input is an important one. Weyer's dynamic texts, and computer-based encyclopedia, as well as Lenat's CYC are relevant in this respect. A project that was originally started by Turner and Kintsch provides a potential basis for designing such a system. Turner has developed a computer system that permits minimally experienced users to turn an arbitrary text into a semi-formal semantic representation. This "propositionalyzer" is now a finished product, delivered to its AFOSR sponsors [Turner 87]. It could be used as a tool in research on text-based knowledge bases. We know how to represent texts as propositional textbases; considerable work has been done on how such textbases are generalized, how "gist" is extracted, etc. (e.g., [Turner, McCutchen 87]); questions of knowledge organization and retrieval, which are crucial for such a project, are already being investigated by us as part of the present project. On the other hand, questions about knowledge updating, knowledge integration, and the construction of situation models are at present poorly understood. A beginning has been made in [Dijk, Kintsch 83], but these questions would have to be made into a secondary research focus within the present project.

11. Suzanne Mannes: Modeling the Generation of Knowledge Structures: The Basics

Researchers have had much success modeling various types of human knowledge retrieval using accepted psychological models [Raaijmaker, Shiffrin 81]. The types of retrieval include free association and category naming among others [Walker, Kintsch 85]. Unfortunately, further attempts to use these same models to model knowledge retrieval about procedures have fallen short of expectations. In particular, it is the nature of the temporal constraints necessary to carry out procedures which has eluded modeling efforts. We collected verbal protocol data from subjects about various well established procedures, such as going to the grocery store and going to a restaurant for a meal and then devised a way of graphing the obtained protocols which clarified the structure of the protocol content. This provided us with some understanding (or informed guesses) about how our subjects were using their knowledge structure in order to produce their protocols. We found high agreement on the events to be included in each of these procedures as well as high agreement about the order in which these events occurred. This high degree of agreement among subjects suggested that we were indeed dealing with a modelable phenomenon although previous attempts to do so had failed. The fact that temporal structure was so well agreed upon led us to believe that this sort of information was what was lacking in those failed attempts.

We started with an associative network of the events mentioned in the protocols and added to this more events which were produced as free associates to the original events. In this network, not only are the events represented but the temporal relationships between them are represented explicitly. With this knowledge built into the network, we set out to use the same model which had previously failed [Raaijmaker, Shiffrin 81] with one small modification. In the original model, retrieval is probabilisticly determined and although item retrieval is dependent upon the current cue being used, the establishment of future cues is not seriously constrained by the current cue. In our modification, the current cue is used in a matching process to determine the next cue to be formed. Once the model has recalled all it can about getting to the store it looks for an item (propositions in our network) which tells it not only to what it is associated but also what comes next in the sequence of events under consideration. This insures faithfulness to the procedure and results in fairly constrained temporal retrieval of the procedure subcomponents.

12. Peter Polson: Cognitive Complexity Theory and Transfer of Skills

This section reviews work done by Polson and his colleagues on transfer of user skills in human-computer interaction. It is based on material from [Polson 87] and [Polson, Muncher, Kieras 87].

12.1 Introduction

It is possible to give a very precise theoretical characterization to positive and negative transfer effects. Existing models of the acquisition and transfer of cognitive skills enable us to provide precise theoretical descriptions of these transfer processes. These same models can in turn be used to design consistent user interfaces for a wide range of tasks and systems to avoid negative transfer

12.2 Models of Human Computer Interaction

The GOMS model [Card, Moran, Newell 83] and Cognitive Complexity Theory (CCT) [Kieras, Polson 85] both characterize the knowledge necessary to make effective, routine use of software tools like an operating system, a text editor, or a data-base manager. The GOMS formalism describes the *content* and *structure* of the knowledge underlying these skills. CCT represents this knowledge as production rules which permits one to quantify *amount*. CCT incorporates *all* of the assumptions of the GOMS model. The production rule formalism enables one to derive quantitative predictions of training time, transfer of user skills, and performance. The next two sections describe each framework.

The GOMS Model

The GOMS model represents a user's knowledge of how to carry out routine skills in terms of *goals*, *operations*, *methods*, and *selection rules*.

Goals represent a user's intention to perform a task, a subtask, or a single cognitive or a physical operation. Goals are organized into structures of interrelated goals that sequence cognitive operations and user actions.

Operations characterize elementary physical actions (e.g., pressing a function key or typing a string of characters), and cognitive operations not analyzed by the theory (e.g., perceptual operations, retrieving an item from memory, or reading a parameter and storing it in working memory).

A user's knowledge is organized into *methods* which are subroutines. Methods generate sequences of operations that accomplish specific goals or subgoals. The goal structure of a method characterizes its internal organization and control structure.

Selection rules specify the conditions under which it is appropriate to execute a method to effectively accomplish a specific goal in a given context. They are compiled pieces of problem solving knowledge. They function by asserting the goal to execute a given method in the appropriate context.

Content and Structure of a User's Knowledge

A user's knowledge is a mixture of task-specific information, the high-level methods, and system-specific knowledge, the low-level methods. Intermediate level methods can appear in many different contexts. The knowledge captured in the GOMS representation describes *both* general knowledge of how the task is to be decomposed as well as specific information on how to execute functions required to complete the task on a given system.

Cognitive Complexity Theory

Kieras and Polson [Kieras, Polson 85] propose that the knowledge represented in a GOMS model be formalized as a production system. Selection of production systems as a vehicle for formalizing this knowledge was theoretically motivated. Newell and Simon [Newell, Simon 72] argue that the architecture of the human information processing system can be characterized as a production system. Since then, production system models have been developed for various cognitive processes (problem solving: [Simon 75; Karat 83]; text comprehension: [Kieras 82]; cognitive skills: [Anderson 82]).

An Overview of Production System Models

A production system represents the knowledge necessary to perform a task as a collection of *rules*. A rule is a condition-action pair of the form

```
IF (condition) THEN (action)
```

where the condition and action are both complex. The *condition* represents a pattern of information in working memory that specifies when a physical action or cognitive operation represented in the *action* should be executed. The condition includes a description of an explicit pattern of goals and subgoals, the state of the environment, (e.g., prompts and other information on a CRT display), and other needed information in working memory.

Production Rules and the GOMS Model

A production system model is derived by first performing a GOMS analyses and then writing a program implementing the methods and control structures described in the GOMS model. Although GOMS models are better structural and qualitative description of the knowledge necessary to perform tasks, expressing the knowledge and processes in the production system formalism permits the derivation of well motivated, quantitative predictions for training time, transfer, and execution time for various tasks.

Kieras and Bovair [Kieras, Bovair 86], Polson and Kieras [Polson, Kieras 85] and Polson, Muncher, and Engelbeck [Polson, Muncher, Engelbeck 86] among others have successfully tested assumptions underlying these predictions. These authors have shown that the amount of time required to learn a task is a linear function of the number of new rules that must be learned in order to successfully execute the task and that execution time is the sum of the execution times for the rules that fire in order to complete the task. They have shown that transfer of training can be characterized in terms of shared rules.

12.3 Theoretical Models Transfer

The dominant theoretical approach for explaining specific transfer effects is due to [Thorndike, Woodward 01] and [Thorndike 14]. Thorndike assumed that transfer between two tasks is mediated by *common elements*. Common elements acquired in a first task that successfully generalize to a second do not have to be relearned during the acquisition of the second task. If a large number amount of the knowledge required to successfully perform the second task transferred, there can be a dramatic reduction in training time.

Kieras and Bovair [Kieras, Bovair 86] and Polson and Kieras [Polson, Kieras 85] proposed that a common elements theory of transfer could account for positive transfer effects during the acquisition of operating procedures. The common elements are the rules. Tasks can share methods and sequences of user actions and cognitive operations. These shared components are represented by common rules. It is assumed that these shared rules are always incorporated into the representation of a new task at little or no cost in training time. Thus, for a new task in the middle of a training sequence, the number of new unique rules may be a small fraction of the total set of rules necessary to execute this task.

Positive Transfer

This section briefly describes results from the human-computer interaction literature demonstrating the magnitudes of the transfer effects and showing how CCT [Kieras, Polson 85] can explain these results.

Polson, Muncher, and Engelbeck [Polson, Muncher, Engelbeck 86] found very large transfer effects, on the order of four to one reductions in training time, for learning to perform a simple utility task on a menu-based, stand-alone, word processor. Their theoretical analysis showed that a significant portion of the knowledge, when quantified in terms of number of rules, required to perform these tasks were in consistent low-level methods methods for making menu transitions, entering parameters, and the like.

Singley and Anderson [Singley, Anderson 85] found large transfer effects between different text editors, e.g., transfer from a line to a screen editor. Polson, Bovair, and Kieras [Polson, Bovair, Kieras 87] found effects of similar magnitude for transfer between two different screen editors. Their theoretical analysis showed that editors share common top level methods that decompose the task of editing a manuscript into a series of subtasks involving individual changes in the manuscript. Furthermore, even very different editors share low-level methods, e.g., cursor positioning. Text editing is a task where transfer is mediated by knowledge of the general structure of the task as well as shared methods.

The Xerox STAR is a workstation that was explicitly designed to maximize the transfer of methods both within a given application as well as across different applications [Smith et al. 83]. All commands have a common format. The user first selects an object to be manipulated using specialized selection methods for different kinds of text or graphic objects. The operation is selected by pressing one of four command keys on the keyboard. For example, hitting the delete key causes the selected object to be deleted.

Ziegler, Vossen, and Hoppe [Ziegler, Vossen, Hoppe 86] carried out transfer experiments with the STAR workstation. They studied transfer between text and graphics editors. They showed that common methods acquired in one context were successfully transferred to the other leading to very large transfer effects. Further, they were able to provide a quantitative analysis of the magnitude of these transfer effects using a production system model like those in [Polson, Bovair, Kieras 87].

Negative Transfer

Current application programs on most systems provide ample opportunities for negative transfer. There are a limited number of functions that reoccur in a surprising number of different contexts. For example, enter and edit a line of text in a manuscript, in a spread sheet, or as an operating system command. Often, different sequences of user actions must be employed to accomplish common goals in each different context.

There are two interpretations of negative transfer effects. An *interference* interpretation of negative transfer assumes that representations of how to use first and second editors interfere with each other and predicts that negative transfer should increase with increasing similarity of two editors. A *repair* interpretation assumes that knowledge of how to use the second editor is acquired by modifying the representation of how to use the first editor. Learning a new highly similar skill requires limited modifications to the representation of how to use the second editor. Thus, negative transfer is a decreasing function of similarity.

Negative transfer occurs when previously acquired rules fire in a new situation and generates an incorrect action or sequence of actions. For example, the user intends to scroll forward one page and instead the system deletes a line. The generalization process, causing old rules to fire in the new situation is identical to the processes assumed for positive transfer. However, in the negative transfer case, the user must make repairs modifying old rules and acquiring new rules that generate the correct sequence of actions in the new situation.

Three different components of the repair process could contribute to negative transfer. The first involves discovering that an old method is in fact incorrect in the new context. For example, users may perceive that they are correctly transferring skills from an electric typewriter to a word processor ending each line with a hard carriage return. Only later, they discover that they've made a serious error when they attempt to edit the text. Another possibility is that old and new methods are very similar to each other. Users take a long time to realize that repairs are necessary because the old method frequently works in the new context. Finally, the simplest situation is when a method immediately leads to an obvious error in the new situation.

The second component of the process is actually making the necessary repairs. Users may have to acquire a large number of new rules because old and new methods are very different. Another possibility is that users may have a great deal of difficulty discovering the correct new method. For example, they have to search through a large poorly organized reference manual to discover a description of the new method.

The third source of negative transfer is inefficient new methods. It may be the case that there is a large amount of transfer with the new method requiring a very small modification of the original representation in order to function effectively in the new context. However, the resulting method may be very inefficient leading to poor performance.

Classical interference theory [Postman 71] provides another class of explanations for negative transfer occurring during the acquisition of new methods. Old and new rules are highly similar to each other. Such similarities can cause serious difficulties in acquiring rules that represent the new method.

A Test of Repair versus Interference Interpretations

It was shown in the preceding section that there are a number of explanations for negative transfer. The repair position assumes that detecting inconsistencies and constructing repairs can lead to a wide range of transfer effects depending on the difficulties of both processes. Polson, Muncher, and Kieras [Polson, Muncher, Kieras 87] report an experiment designed to constrain the repair processes permitting a test of repair versus interference interpretations.

Polson, Muncher, and Kieras told subjects that they were being transferred to a new editor; subjects were given explicit instruction on the new methods enabling them make effective repairs. Thus, the repair position predicts the least difficulty in transfer for subjects learning two similar editors and the greatest difficulty for subjects learning two dissimilar editors. The interference position makes the converse predictions.

The Polson, Muncher, and Kieras experiment had five groups. The groups differed in terms of the similarity of their original learning and transfer tasks. All groups during original learning mastered a computer-based task, text editing for four groups, and then practiced that task for two days. On the fourth day all five groups learned a common text editor in which 15 editing functions were activated by control shifted single characters. The pairing of editing function and character was non-mnemonic. On the fifth day of the experiment they practiced on this second editor.

On day four, the five groups were rank ordered in the following sequence. The group that performed the best worked with the identical editor on all five days of the experiment; they had nothing new to learn. The next best group had learned a first editor with an identical structure but in which the same 15 letters were paired with different editing functions. The next group's first editor had a similar structure. However, the commands were mnemonic, e.g., DW activated the function DELETE WORD. The fourth group learned a very different first editor. The commands on this editor were all block mode editing functions. The fifth group learned wide collection of utility tasks for two word processors using menu based interfaces.

The five groups are rank ordered exactly according to the number of new rules they must generate in order to master the second editor as predicted by repair theory. Interference theory would predict that the group learning as second editor with letters associated with different editing function should have more difficulty than was observed. These results are strong support for the repair interpretation of negative transfer.

An Example of the Impact of Low Level Inconsistencies

Karat et al. [Karat et al. 86] examined transfer between three highly similar word processing systems that were intended by their designers to facilitate the transfer of user skills from one system to another. The first system was developed as a menu-based, stand alone word processor. A major goal in the design of the follow-on systems was to facilitate transfer from the dedicated, stand-alone, word processor to word processors hosted on a general purpose personal computer and a departmental computing system.

Karat et al. evaluated the magnitude of transfer effects from the dedicated version of the system to the other two system environments. The transfer effects were disappointingly small. Karat et al. found users' difficulties transferring their skill were due almost entirely to subtle differences in low level-methods. For example, many problems were caused by the fact that the dedicated version of the system has

specialized, labeled function keys. On the general purpose personal computer and the departmental computer system versions, the user had to learn and retain the locations of the corresponding functions on an unlabeled, generic keyboard. Inconsistencies in key assignments for activating known functions disrupted performance when users attempted to transfer their skills from one version of the system to another.

From the point of view of the transfer theory, these differences between the three systems caused negative transfer because the repairs were very difficult to make. Subjects had no trouble discovering that repairs are necessary. The reference documentation was available, but subjects rarely made use of it. They tried to discover the new correct action and almost always failed requiring experimenter intervention.

13. Clayton Lewis: NoPumpG, EXPL, and Spatial Thought Dumper

13.1 NoPumpG: A System for Rapid Implementation and Prototyping of Advanced User Interfaces

Goal: To create a system permitting advanced user interfaces to be created without the need for conventional programming.

Rationale: Development of advanced user interfaces, and research in the area of human-computer interaction, is hampered by the cost of programming. While a variety of user interface management systems are available as tools, these provide adequate support only for more traditional dialogue-style or form-style interfaces, or for special predefined classes of interface, as for the Apple Macintosh. Researchers or developers needing to explore a wide range of innovative interfaces must make a major investment in programming skills to allow them to exploit powerful, general-purpose tools such as are available in LISP or Smalltalk environments. Not only is the required investment in programming skills high, preventing behavioral scientists, graphic artists, and other persons with pertinent design skills from shaping interfaces directly, but even for skilled programmers the effort needed to create a new interface is very great, easily amounting to several person-years of effort for typical projects. There is an opportunity to accelerate significantly the pace of development in the human-computer interaction field if tools can be devised that dramatically reduce these costs.

Technical approach: NoPumpG is a user interface construction system that aims to reduce construction costs by substituting spreadsheet-style programming for conventional programming, while retaining the flexibility to create a wide range of interaction styles, including direct manipulation of graphically-displayed objects. By eliminating the need for conventional programming constructs like flow of control, input/output, and variable declarations, and relying instead on spreadsheet concepts of proven accessibility to non-programmers, NoPumpG will make it possible for designers to shape and implement interfaces at less cost and with less investment in programming skills than can be attained today.

The key to NoPumpG is the coupling between spreadsheet programming, capable of handling arithmetic operations, and graphical primitives needed to create direct manipulation interfaces. In contrast to existing spreadsheets with graphics facilities, which can be used to display a limited range of business presentation graphics, usually with no interactive capability, NoPumpG provides low-level graphics primitives which can be combined to form arbitrary graphical objects and which support full interaction.

The linkage between the spreadsheet and the graphical primitives is provided by specialized spreadsheet cells called *control cells* which are associated with each graphical primitive. For example, a line segment has four control cells, one for each coordinate of its end points. If a control cell contains a formula, then the corresponding coordinate of the line is determined by that formula, which can depend upon arbitrary computations in the spreadsheet. If a control cell contains no formula, the coordinate is not determined, and the line segment can be moved by a graphical interaction using a mouse. As the line segment is moved, and the coordinate changes, the value in the control cell is automatically updated. Since formulae in other cells in the spreadsheet can refer to the control cell, this allows arbitrary spreadsheet calculations to be performed on values controlled by a graphical interaction. By placing appropriate formulae in some but not others of the control cells of graphical primitives, a variety of constrained motions can be obtained. For example, it is possible to create a movable needle dial, which can be pivoted

around a fixed end, or an assemblage of lines forming a rectangle or other shape which moves as a unit. The resulting flexibility appears to be adequate to construct a wide range of interactive graphical interfaces, including for example a simplified flight simulator with graphical control, starting from low-level (and hence not specialized) primitives, with only spreadsheet programming.

A note on the name: The name "NoPump" refers to a style of computation, best exemplified by spreadsheets, in which data objects are shared between user and system, so that they need not be "pumped" back and forth between user and system by explicit i/o coding as in conventional programming [Draper 86]. The suffix "G" stands for "graphics," allowing for extensions of the NoPump style to other domains, such as knowledge representation.

Status and plans: A preliminary version of NoPumpG is currently in development, with completion scheduled for May 1987. The preliminary version will support only the basic paradigm sketched above, and will lack many features that will be desirable in an effective tool. Follow-on work will undertake to expand the preliminary version to include some of these features, as follows.

Aggregate move and copy. This feature will simplify the development of complex interfaces, and support the reuse of structures across interfaces, by permitting groups of cells and graphical primitives to be identified, moved, and copied.

Access to mouse position. In preliminary NoPumpG mouse movements can influence computations only via movements of graphical primitives. Increased flexibility can be obtained in two ways. First, control cells associated directly with the current mouse coordinates can be established. Formulae referring to these cells could then carry out arbitrary computations based on mouse position. While this would make possible the construction of mouse-sensitive objects, higher-level support will also be provided by associating a mouse position control cell with each graphical primitive. For example, a cell associated with a line segment could contain the value 1 or -1 depending on which side of the line the mouse is currently on (using a convention to differentiate the two sides.) A simple logical computation on such cells for the lines making up a box tells whether the mouse is currently inside the box, with out the need to do any arithmetic.

Character data and operations. Preliminary NoPumpG will support only arithmetic (and some logical) operations within the spreadsheet. Extensions will be provided to permit the manipulation of character strings.

Use of color. Preliminary NoPumpG will provide only limited control of color of graphics primitives. Color of primitives, and use of color to fill areas, will be explored.

Control of cell presentation. Spreadsheet cells are a natural input mechanism to an application, but they lack flexibility in appearance. In preliminary NoPumpG, the name, formula (if any) and value of a cell would all be displayed in a standard way. By associating (optional) control cells with cells themselves it will be possible to control the size and position of a cell, whether or not all components are displayed (or only the value, for example), and the like. By using the control cell mechanism to provide this control, these variations can be controlled by computations in the spreadsheet. For example, cells could be hidden or displayed according to whether they become relevant in the course of a computation. The appearance of a previously-invisible cell could be used to prompt a user for input.

Linkage to application code. Many interfaces could be adequately prototyped, or even implemented for actual use, using only spreadsheet computations. Others will require services provided by other programs (for example, to access a database, or control external devices.) The natural way to provide this in NoPumpG is via functions in formulae whose evaluation triggers the execution of programs outside NoPumpG. In the Hewlett-Packard Bobcat environment, in which preliminary NoPumpG is being built, this extension would make possible escape into arbitrary LISP programs.

Session logging and playback. In studying interface prototypes it is useful to be able to record for later analysis or re-creation a record of user actions. It appears that the best point of access for such a record in NoPumpG is the input handler, which could (optionally) generate a timestamped record of mouse movements and button and key presses. The handler could be enabled to read back such a record, rather than accepting actual input, thus allowing sessions to be replayed. In addition, higher-level, less detailed traces should be supported. A possible approach is to permit any cell to create a trace of all modifications to its value. There is no natural playback possible from such a partial record, but it would be useful for analyzing user sessions at a high level. For example, a flight simulator could record all uses of a particular control without tracing all the details of a session.

13.2 EXPL: A Model of Causal Analysis and Generalization for Human-Computer Interaction

Lewis [Lewis 86a] argued that learners invest effort in understanding how a system works so as to be able to generalize examples of its use that they see. The EXPL system [Lewis 86b; Lewis 86c] is a machine model of this process. EXPL accepts a (manually-prepared) description of a user-system dialogue, including user actions and system responses. It uses a small set of heuristics to conjecture why each user action was performed (that is, how each action contributes to the apparent goal), and why each system response (or aspect of a response) occurred. For example, if an object on the screen disappears, EXPL attempts to determine why something disappeared, instead of (say) turning red, and why the particular thing was what disappeared, instead of something else. EXPL then uses its conjectures about the causal relationships in the dialogue as the basis for generalizing to new dialogues. For example, if its analysis of two different dialogues suggest how a particular object was specified in one, and how a particular operation was specified in the other, EXPL can construct a new dialogue that attempts to apply the operation to the object.

The intent of the EXPL model is to provide insight into what is easy to understand from seeing a system demonstrated and what is hard. When EXPL's simple heuristics are adequate to determine how a dialogue works, it may be that human learners would have no trouble in performing their own analysis. Studies of artificial learning situations have confirmed that human learners do use heuristics similar to EXPL's [Lewis 86c]. However, human learners are able to draw on extensive background knowledge that EXPL lacks in interpreting novel dialogues, and this knowledge may suffice to provide understanding in cases where EXPL would fail.

13.3 The Spatial Thought Dumper and the Explicitness Bug

For me, a major obstacle to the use of existing information systems is the need to be explicit about aspects of the data which I wish to process. In general, many proposed systems, from personal databases to calendaring systems to mail filters to advanced text formatters, require the user to think

about and represent explicitly aspects of data that normally remain implicit, such as: "These seven digits are a telephone number," "I am willing to let a meeting with Doe preempt a meeting with Smith," "I am not interested in messages about the Faculty Senate unless they are from Lloyd," or "This is an enumeration." Striking the right balance between making (or permitting) users to specify more information like this, so that the system can manipulate it and provide functions that depend on it, and leaving users free to deal with such issues on a casual, ad hoc basis is a major challenge in system design. I conjecture that many systems set the balance wrong, and will be little used because they require users to say too much up front about what's going on. We can call this the "Explicitness Bug."

The Spatial Thought Dumper is a notional system that attempts to strike the balance at a different point from most systems, requiring no commitment whatever from the user to represent anything explicitly. It aims to provide a personal data repository comparable in function, but superior in usability, to the welter of notes and jottings that cover many people's desks. The system manages an indefinitely large plane, part of which is seen on the screen at any moment. Any part of the screen can be typed on. The user writes information at any desired location, using spatial arrangement to represent semantic connections, just is as done on paper. Users don't have to say "This is Doe's phone number," in any formal or informal way: they just type the digits somewhere near Doe's name. The system can provide functions that scraps of paper don't, such as global search. It can also make room to put new information where it's wanted: when the user indicates an insertion point the system can move existing items apart in such a way that items that are close together stay close together, while ones that are far apart may be separated more.

The Spatial Thought Dumper might present an attractive compromise to users like me. While it offers more limited function than more structured representation systems, it could be used with no investment in structural definition before use, and no need to make relationships explicit at the time information is stored.

14. Paul Smolensky and Brigham Bell: EUCLID, a System for Argumentation Support

Reasoning is a demanding intellectual task that is greatly facilitated by appropriate notational systems. We are developing a new notational system for supporting *reasoned discourse*: the construction, communication, and assessment of reasoned arguments about non-formal domains. This notational system is an active one: a computer environment called EUCLID. In EUCLID, arguments are displayed in tabular and graphical forms; a library of argument types provides users with tools for constructing arguments, and EUCLID's knowledge of argument structure offers users tools for argument comprehension and assessment. EUCLID rests on a language called ARL for formally representing argument structure; argument content is expressed in natural language. The development of EUCLID includes research on:

- the structure of actual examples of reasoned discourse,
- the representation of argument structure using a formal language,
- the management and modeling of large data structures representing arguments,
- the nature and effectiveness of human reasoning when the new notational system is used.

15. John Carroll: Consultant Dialogues

John Carroll reported on their investigations carried out in the User Interface Institute at IBM to augment the technological feasibility of systems (especially advice-giving systems [Carroll, McKendree 85]) by empirical studies of whether and how people will find a new technology useful and tractable. He claimed that it is pointless to build such systems unless we take into account behavioral requirements on their usefulness and usability.

Their empirical studies showed that the questions which users ask are primarily result-oriented and not method-oriented or fact-oriented, and the questions are followed up by a request for verification.

Carroll pointed out that advisors can and have to play many different roles:

- they act as information source;
- they infer unstated goals and assumptions;
- they describe what is happening and draw conclusions;
- they assist in structuring the problem;
- they can deal with incomplete queries by making assumptions and providing alternatives;
- they have strategies for pre-empting follow-ups.

He also observed that there exists a credibility problem for the advisor. Advice-seeking persons build a model of the advisor based on whether the advice received was useful or not.

The empirical work has been carried out by studying advisory dialogues conducted by electronic mail, by phone, and by face-to-face communication.

Details about this work can be found in [Carroll, McKendree 85] and [Aaronson, Carroll 87].

16. Robin Jeffries: Task Scripts -- An Approach to Increased Access to Functionality

Many of the talks at this workshop described how artificial intelligence techniques and/or graphically rich interfaces could provide people with improved access to high-functionality systems. The project I reported on took a different approach to this problem. A careful analysis of the domain -- voice mail -- led us to provide access to the basic functionality by organizing an interface around a collection of canonical tasks; users could then take advantage of a variety of customization features to access additional capabilities of the system and to adapt the interface to their particular needs.

A voice mail system permits users to record, send, receive and otherwise manipulate voice messages using a telephone touch-pad for input and phone speaker for output. The typical user is not computer sophisticated and uses the system regularly but not frequently. In analyzing the uses for such a system, we determined that there were a large number of actions to be provided (as many as 500 distinct options); however, there were a relatively small set of tasks that all users would want to accomplish (e.g., listen to new messages, send a message). Users would vary in the exact steps they used to carry out a task, but there were strong commonalities across the major steps. This analysis suggested similarities with the concept of *scripts* [Schank, Abelson 77]. The interface we designed was inspired by the attributes of scripts; in particular, it provided:

- a canonical version of a task, understood by all users,
- variants based on individual preferences,
- variants based on differing circumstances,
- new scripts derived from existing ones.

The basic concept of the task script interface was that a user signed on to the system, selected a task, and the task was executed to completion, with the system making all the necessary decisions from a set of predetermined defaults. At each point where a decision was required, the user heard a short question suggesting the options available at this point (e.g., "Header style?"); then the system selected an option and echoed it to the user ("Brief headers."). If the user did nothing, the system continued. Thus, the novice user, who is likely to be satisfied with doing only standard things, is able to accomplish useful work without any prior knowledge of what the components of a task are or how to access or combine them. The method used may not be the most efficient one for the particular task, but it will get the job done. Since novice users are typically more overwhelmed by the number and meaning of the myriad options available than they are concerned with efficient use of a machine, the interface is optimized to make the new user's interaction go smoothly.

As users become more experienced, they wish to go beyond the canonical tasks. Users had other options available at each choice point. They could interrupt the system and have it play out the available options. So long as the question at each choice point properly suggests the nature of the options available, very little search is necessary for the user to find her desired choice. A single button press overrode the system default with the user's current selection.

A crucial aspect of the success of the interface was that any action, whether specifically selected by the user or chosen by the system, could be undone. This was true no matter how many intervening actions had occurred. So the user could explore the system without fear of disastrous consequences. If he made a selection that produced an unintended result, it was a trivial matter to reverse the effects of that

action.

For a novice user, the success of the interface is critically dependent on having the right set of tasks and default options. Users must be able to get useful work accomplished with the system as initially configured. However, as users become more experienced, they will want to go beyond this basic functionality set. The usefulness of the interface to the more experienced user depends on how well it can be customized to her changing needs. We provided several levels of customization. One was simply the ability to chose options other than the defaults when circumstances required. The next was the ability to make those changes permanent -- to incorporate them into the user's script. We also provided the ability to set global attributes, such as the language the system spoke to you in. This was used for features that were independent of a particular script.

Our most extensive customization capability was the ability to clone an existing script and modify its defaults. This enabled users to customize their scripts to differing situations. Thus, a user could have one script for listening to new messages and another one, with different default actions, for listening to urgent messages only.

The fact that the system permits (in fact, requires) that customization be done by actually carrying out the new version of the task is an important contribution of the interface. Customization "by doing" has two advantages: (1) the user gets immediate feedback about the effects of the change being made, including any unintended side effects, and (2) the user makes the change in the appropriate context, with the additional cues it provides.

We considered additional ways to customize the interface, but most of them proved to be infeasible given the constraints of the telephone as keyboard. One such idea was the ability to build new scripts that were combinations of existing scripts. An arbitrary ability to concatenate choice points without unintended interactions is an unsolved problem in user programming, but a subset of that -- the ability to branch to the beginning of a second script from within a script, with control returning to the original script when the second was exited -- was within our means. This would have provided a substantial increase in functionality; however, we were never able to determine a scheme that would provide users with the necessary feedback as to where they were.

The ideas from this project are fairly general and could be applied to many situations where more functionality is available than is immediately obvious to users. In particular, the general ideas of: (1) organizing the functionality around real tasks that users want to do, (2) providing gradual access to additional functionality, (3) making exploration painless by making all actions trivially undoable, and (4) providing layers of customizability, apply to most, if not all, interfaces to complex systems. The specific idea of having the interface take control of the interaction and perform an entire task without intervention probably applies to a smaller set of applications. The usefulness of such a technique depends on how predictable the user's actions are. It will be most useful if the user's choice can be predicted 50%--80% of the time. Any less predictable, and the overhead of having to correct the system so often is too high; any more predictable, and the user needs to be provided with an even simpler interface. There are a fairly substantial number of applications whose predictability falls in that range. Many office applications, such as mail, calendar, etc., should have such a property.

The goal of this presentation was to show that a careful analysis of the tasks a system will be used for and a good understanding of the principles of human-computer interaction can go far towards making a

complex system more usable. The task analysis is crucial to providing the right core functionality to build the interface upon and to expose to the user initially. The additional step of providing users ways to customize the interface to their idiosyncracies and task demands allows the system to smoothly adapt to the user's growing expertise.

17. Christian Rathke and Hal Eden: Reorganization of Knowledge Structures

Finding information on a computer system often is a difficult problem. Depending on the task at hand, it often involves looking up documentation, applying sophisticated search procedures (which have to be known and found themselves), or recalling the location and access path.

Organizational structures such as file hierarchies, inheritance networks or naming conventions help. This is especially true in an environment that we create ourselves. These personalized environments can be compared to physical world in which we live. As a matter of fact, it can serve as a success model of how to organize and find information. Examples are:

- *Office environment.* It consists of locations such as a desk, book shelves, drawers, and folders. The ordering principles are determined by us. They are constraint by the physical properties of the items (shape, volume, size, weight).
- *Libraries.* Much effort has been investigated in organizing libraries. Standards have evolved that support the acquisition and retrieval of new items.
- *File hierarchies.* The hierarchical structure of libraries is reflected in computer based hierarchies of files.
- *LISP functions.* Different ordering schemes are applied to LISP functions. They can be grouped according to the type of arguments they take, to the purpose for which they are designed, to the package in which they are defined.

Many more examples can be found that demonstrate how structure is used to facilitate the management and retrieval task. In general, organizational structures are designed to support information retrieval and usage:

- Our personal office environment has a rich structure. This structure helps us in retrieving books, papers, envelopes, etc.
- UNIX file hierarchies, although sometimes awkward, structure our electronic information.
- Databases can be seen as structuring devices that make large amounts of data manageable.
- Catalogues in libraries or software systems such as the Smalltalk Browser help us in finding information.
- The UNIX password file provides structures to be used by the `login` program or `finger`.
- ObjTalk inheritance hierarchies provide structure for sharing code.

The specific organizational structure chosen is determined by usage, by the amount of information, by personal preferences and experiences, by the knowledge about the domain, and by constraints imposed from outside (physical, hardware/software restrictions).

One of the main problems with organizational structures is that each of the determinators can change. For instance, the UNIX password file is not only used by the `login` program but also by programs that tell about names and addresses of known users (the `finger` program). The original structure of the password file is no longer adequate.

For areas that evolve dynamically, there is an almost constant need to reorganize. On the other hand, we often stick to inadequate structures, because the effort to change existing structures is too difficult. There

is only little support to solve this problem. Possible approaches to solving it are:

1. *Support for multiple structures.* Since in a typical working environment information is used in different ways, organizational structures must not be restricted to a single principle. Whereas telephone number may be best structured alphabetically by the persons' names, the dates of meetings should be structured chronologically.
2. *Allow multiple structural views.* The same information may need different structures for different tasks. It should be possible to view, for example, a literature database according to Authors' names as well as according to titles or contents.
3. *Allow for shared views (global, group vs personal).* Structural properties may depend on the individual or the group that makes use of the information. Filter mechanisms and perspectives should adjust the structure to its users.
4. *Allow for combined views (global view determined by combination of personal views).*
5. *Editing tools*
    - for manipulation of structure,
    - for verification of structural consistency.
6. *automate (re)structuring.*

These tools must avoid to replace old problems by new ones. They have to be as non-obtrusive as possible and the benefit/effort ratio has to be high.

If we want to support (re)organization and provide tools that deal with organizing things, we have to represent knowledge about organizational structures and tools explicitly. Knowledge about structures is often hidden or is not general enough to be widely applicable:

- In knowledge systems, that provide automatic subsumption mechanisms, the knowledge about the structure itself is not made explicit. Part of the knowledge is captured in procedures that work on generalization hierarchies. This part of the knowledge is difficult to access and even more difficult to change.
- In operating systems such as UNIX programs make heavy use of structures (which are represented in the heads of the designers/users), e.g., `cd` path, e-path, files which comprise the source for the program (`make` provides structuring methods). This makes it almost impossible to change the structure of some file, because no information is kept who makes use of it and in what way.
- In the WLISP environment there are design kits (specialized programs) that have specific knowledge of how to combine ObjTalk classes. Again, this knowledge is represented as procedures and therefore hard to manipulate.
- In UNIX the namespace system is made up of many different data files (/etc/hosts, /etc/networks, /etc/passwd, /etc/group, /usr/lib/aliases, /usr/lib/sendmail.cf ...) each with a certain structure, and many tools (rlogin, login, route, mail, who, rwho, finger, etc.) which operate on this structure. Reorganization might require both parts of this knowledge.

In the current state of the research we are able to indicate some of the problems of structuring and the need for reorganization support. These are the theses on which our research will be based:

- A single organizational structure serves multiple purposes.
- We do not know about future use of the information we want to structure.
- Organizational structures are hard to invent.
- Organizational structures don't fit.
- Changes to structures are likely.

18. Andreas Lemke: Support for Design and Redesign

Support for design and redesign is a prerequisite for users to truly master their computing systems. Section 18.1 will describe some problems in this context. Section 18.2 illustrates some global ways of dealing with them. Section 18.3 will introduce the notion of design kits and describe a general architecture for design kits. Section 18.4 concludes with some unsolved problems.

18.1 The Problems

When using complex information processing systems users face the following problems:

1. *The usability problem.* Users do not master their complex (computing) systems (e.g., editors, document formatters, LISP systems, Smalltalk, mail systems). They use only a small fraction of their functionality.
2. *The redesign problem.* Systems often do not meet particular requirements and, therefore, won't be used.
3. *The learning problem.* Many tasks are relatively unique and require the use of new, unknown facilities.
4. *The end user control problem.* Users want the right amount of control over their systems.

The following sections will analyze the last three of these problems in more detail.

The Redesign Problem

There are many reasons for the necessity of end-user redesign. Most software products are nowadays not built specificly for one particular customer with one clear cut use. Mail systems, for example, are used by people with possibly very different needs and usage patterns that cannot be predicted. Operations that one user executes very frequently may not be needed by another user in a month. Definable macros and function keys are a step in this direction.

A change in the size of the application may also be a reason for modifications. As long as all the data fit on the screen there may be no problem, but when the application grows there must be means for dealing with large amounts of data.

Personal styles like preferences for using keyboard or mouse may cause a desire to change a system.

Often the reasons for redesign are very mundane. Systems must be integrated with other software that was independently developed. The user should be able to repair small bugs without having to find the system developer and (possibly) having to wait months for a solution.

The Learning Problem

Often there exists a tradeoff between finding a simple solution with much search and the necessity to understand the facilities of the system, and, on the other hand, a complex solution with little learning but that requires one to explicitly build up a complex structure.

Early programming environments, for programming languages like Pascal or C, offered only the latter

choice. They offered some powerful primitives that could be combined to build programs of arbitrary complexity.

Newer, rich programming environments as available on LISP machines or the Smalltalk system also offer the first choice. They have abstractions on many levels ranging from domain independent ones to very specific ones. It is our goal to solve this tradeoff by reducing that what must be learned and by enhancing learning processes with knowledge-based support systems.

The End User Control Problem

Many systems suffer from the problem that they do not provide their users with enough control. Control was done away with in order to simplify usage. This is a shortsighted solution. We need to find more sophisticated approaches.

In general, however, users do not want to exercise more control than they must. Systems should do all reasonable actions automatically. In the TROFF text formatter font size and interline spacing are separately controlled, i.e., changing the font size does not mean that interline spacing will be automatically adapted. If the user forgets to also adjust spacing, lines in large type fonts will overlap (see also the discussion on end-user modifiability in section 4.1).

The system's decisions should not be final. The user should be able to overwrite those decisions for particular instances or for certain classes of situations. The SCRIBE text formatter has abstractions like a major heading whose exact specification may be changed at many levels: a particular major heading, for example, may be modified to use a different font. The same modification may be done globally for a whole document or even for all documents of a certain type or all documents at all. Controlling the behavior of systems also has its problems. Often it is easier to explicitly control something than to tell the system how to do it, especially if this behavior cannot a priori be easily described (the delegation problem). It is very difficult, for example, to explicitly describe those electronic mail messages that are important and that one wants to be alerted of.

18.2 Ways to Go

Systems should be constructed so that they are easy to redesign. Systems should not be monolithic blocks (Figure 18-1). These systems can only be used on one level. Users have no means for modifying their behavior except possibly on the source code level. Good systems are structured into multiple levels of different conceptual complexity. Some of these levels are closer to the application domain, some are closer to the implementation level (Figure 18-2).

In order to provide these intermediate levels it is necessary to explicitly represent abstractions that are easy to understand for users. Fonts in the PostScript page description language [PostScript 85] are represented logically so that their pixel images can be derived for any font size. This causes computational overhead, but presents a clear conceptual view. The computational power of today's computer systems should be used for this purpose rather than to help users to understand implementation-biased abstractions.

In addition to supplying the right abstractions, systems should also have knowledge about typical tasks in

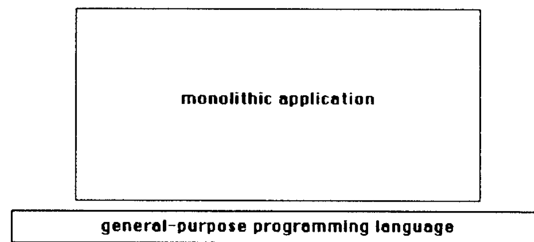


Figure 18-1: Monolithic Systems do not Support Redesign

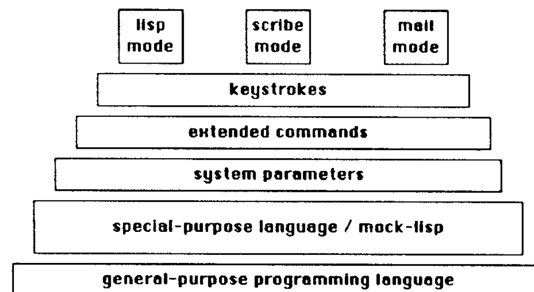


Figure 18-2: A Structured System (like the EMACS Editor) Supports Redesign

the application domain. In an office environment for example, a system should provide support for scheduling tasks by knowing about the concepts of a participant of a meeting, a meeting room, people's preferences and their commitments that influence the finding of a meeting time. Although this task may not be possible to do fully automatically because at some point more important meetings have to supercede less important ones, still, a system can find an approximate solution that makes decisions much easier for humans. Systems can also keep track of subtasks that still have to be done and present examples that can serve as a basis for a problem solution.

18.3 Construction Sets and Design Kits

Construction sets (e.g., the Pinball Construction Set [Budge 83]) provide application domain level concepts and hide implementation aspects. The use of direct manipulation as a user interface technique further reduces the conceptual distance between system and problem. Construction sets provide syntactic help: it is (ideally) not necessary to know how to specify something. All available components and options are or can be made visible. Construction sets provide a safe experimentation environment that shifts the emphasis from error avoidance to constructively dealing with problems.

All these features, however, do not guarantee that useful results will be produced. Donald Norman says about the Pinball Construction Set:

> Much as I enjoy manipulating the parts of the pinball sets, much as my 4-year-old son could learn to work it with almost no training or bother, neither of us are any good at constructing pinball sets. I can't quite get the parts in the right places: When I stretch a part to change its shape, I usually end up with an unworkable part. Balls get stuck in weird corners. The action is either too fast or too slow. Yes, it is easy to change each problem as it is discovered, but the number seems endless.

Design Kits are intended to deal with this problem and give additional support (Figure 18-3). The goal tree represents knowledge about typical design tasks. It allows the user to indicate current goals, intentions, and focus to the system. The system has a library of samples that can be inspected, copied from, or modified to serve as a problem solution. Another component is a critic who, in turn, inspects the user's product (at any stage of the solution process) and suggests critical issues and modifications for the user to take into account. The hierarchical structure of the goal tree allows it to deal with very many components and options because at any point of time the focus is limited to a small part.

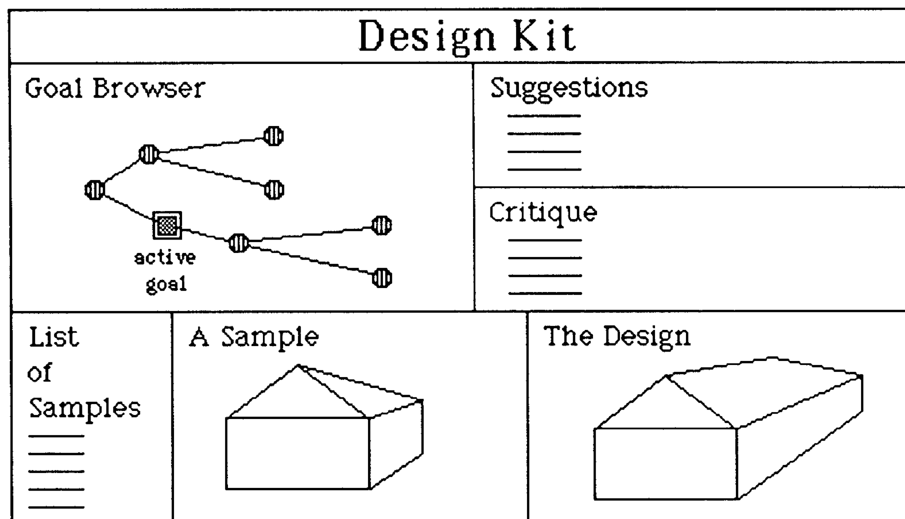


Figure 18-3: An Architecture for Design Kits

18.4 Problems of Design Kits

Two prototypical design kits [Fischer, Lemke 87b] have shown that it is not easy to provide this additional support without loosing the easy learnability and usability of direct manipulation construction sets. In order to describe goals, examples, or suggestions a terminology must be used. Users of a design kit may not know this terminology. If systems can be modified on different levels, consistency is a problem. Once a system has been modified on a lower level, a higher, more abstract level may no longer be able to describe it. Direct manipulation has problems when more abstract entities must be constructed. It is well applicable if there exists a physical metaphor for the domain. Conditions and actions are not easily described.

19. Helga Nieper: Information Retrieval by Reformulation: From ARGON to HELGON

19.1 Introduction

ARGON [Patel-Schneider, Brachman, Levesque 84] is an information retrieval system that is based on an earlier system called RABBIT [Tou et al. 82; Tou, Williams 82; Williams 84]. The basic principle of both systems is *retrieval by reformulation*. The query is created incrementally, the set of instances matching the current query is shown in one window, one example instance is shown in another window (Figure 19-1). All information shown can be used to modify the query (e.g., by requiring or prohibiting a certain value, or by selecting an alternative value from a menu). Thus users who are not familiar with the knowledge base (e.g., with relation or attribute names) or who do not know exactly what they want can be guided towards the appropriate information.

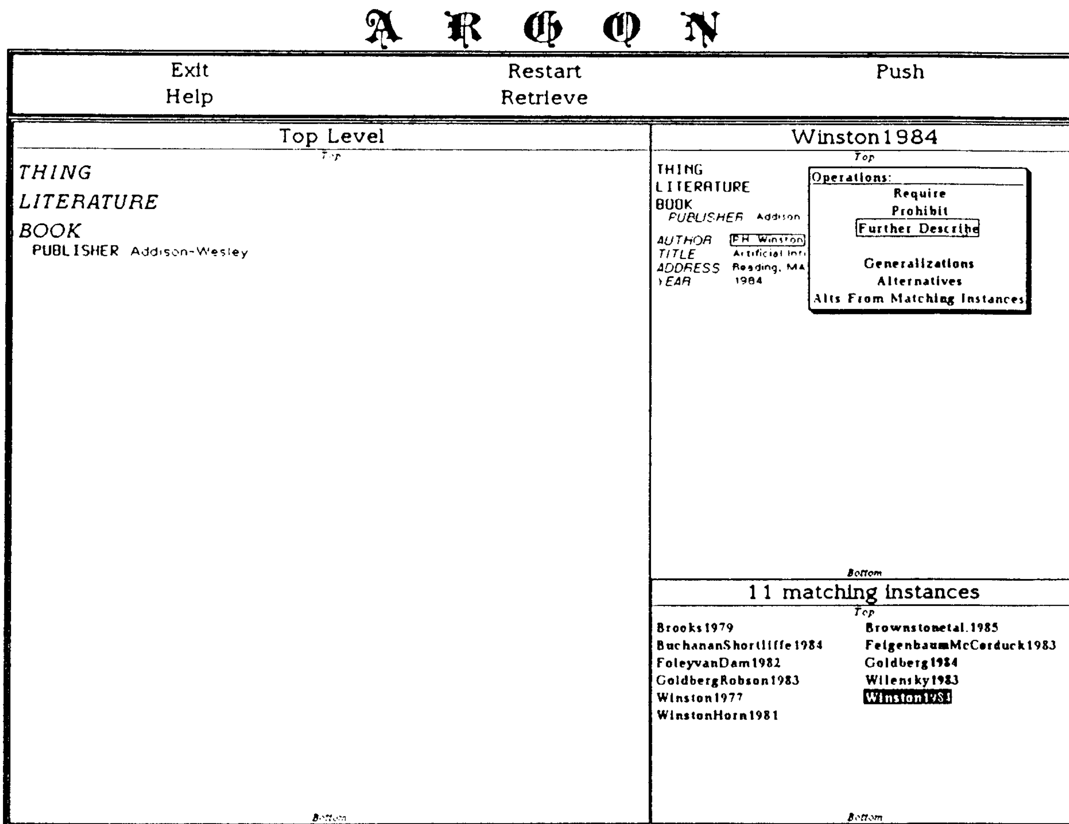


Figure 19-1: ARGON

ARGON uses a frame-based knowledge representation system called KANDOR [Patel-Schneider 84]. Basic units of a KANDOR database are *frames* and *individuals*. Frames are organized in a hierarchy. Individuals

are instances of one or more frames. Information is associated with individuals by means of *slots* (Figure 19-2).

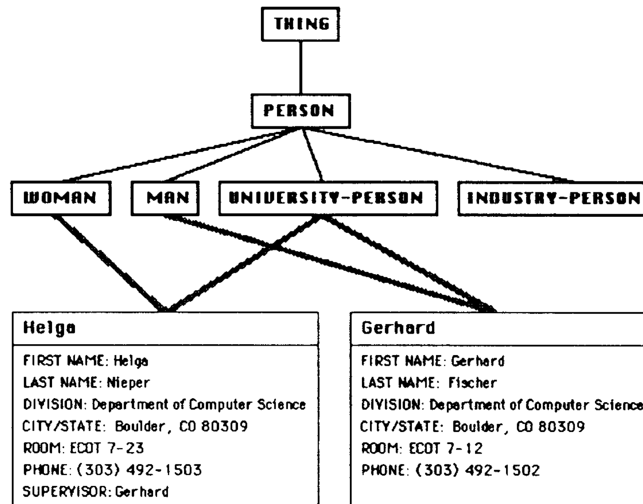


Figure 19-2: KANDOR

19.2 Two Case Studies: Bibliography Data and Mail Messages in ARGON

To test ARGON, already existing data were transformed into a KANDOR database: a bibliography for SCRIBE with about 800 entries, and about 20 UNIX mail folders containing about 500 mail messages. These data were already structured in slots, and therefore the transformation was pretty much straightforward (a sample entry of each of the databases is shown in Figure 19-3).

In case of the bibliography database, a frame LITERATURE is defined with specializations ARTICLE, BOOK, TECHREPORT, etc. (i.e., the same classification SCRIBE uses). The database is set up, so that alternatives to author, publisher, institution, journal, etc. can be selected from a menu. The title of a bibliography entry is split up in words. In this way it is possible to ask questions like the following: retrieve references that have x in their title, references from author x, references in journal x, or references from year x.

For the mail database, only one frame MAIL is defined. The mail messages are not classified, as in the 'message type lattice' in the 'information lens' system [Malone, Grant, Turbak 86; Malone et al. 86]. Alternatives for the sender or receiver of a message and for the folder are selectable from a menu. In the subject field, a 'Re: ' or 're: ' is separated from the rest, so that an original message together with all its replies that are created using the Reply (reply) command can be retrieved. Questions like the following are possible: retrieve messages in folder x, messages from person x, messages sent to person x, messages sent on day x, or messages about subject [Re:]x.

LevesqueBrachmanPatel-Schneider1984

AUTHOR:	H.J. Levesque R.J. Brachman P.F. Patel-Schneider
TITLE:	ARGON: Knowledge Representation Meets Information Retrieval
TYPE:	Fairchild Technical Report
INSTITUTION:	Schlumberger PALO ALTO RESEARCH
NUMBER:	654
MONTH:	September
YEAR:	1984

helga 1/8/87 12:43:03

FROM:	helga
TO:	haleden
CC:	andreas
DATE:	1/8/87
TIME:	12:43:03
SUBJECT:	Re: zmacs in Release 7
FOLDER:	/cs/helga/folders/symbolics
TEXT:	This is really strange...

Figure 19-3: KANDOR Individuals of a Bibliography and a Mail Database

19.3 Extensions to ARGON

One shortcoming of ARGON is that it does not allow input from the keyboard. People usually know *something* about what they are after, and the only way to require a value is to select it from the displayed example or from an 'alternatives' menu. But some slots have so many different values that it is not feasible to select an alternative value from a menu.

The problem is: How do we avoid the users ending up with nothing only because they use different expressions than the system does (e.g., 'Germany' instead of 'West Germany' or 'East Germany')? This implies that we have to allow more than identity (substrings, a 'close match,' or synonyms), not just with input from the keyboard, but in general when database entries are matched against the query. It should also be possible to use predicates in the query, such as '<' or '>' on numerical values.

It is not obvious to the users that the frames are organized in a hierarchy, and it is easy to get lost if this hierarchy is not visible. A graphical display of the frame hierarchy can be used to add frames to the query. Frames that are required can be highlighted in the display, frames that are prohibited crossed out. We need an intelligent layout planning algorithm to arrange the nodes in the frame hierarchy and operations to rearrange the layout by hand if the user is not satisfied with the layout.

In general, it is not possible to show the whole hierarchy at once. The amount of information presented to the user can be restricted through filters. The system can start out with the hierarchy displayed to a certain depth only and allow the users to zoom in on parts of interest to them. If information is classified along several orthogonal schemes, one of those schemes can be filtered out, and in this way the information can be viewed from different perspectives. Because individuals may be instance of more than one frame and therefore may have slots that are not applicable to all of those frames, slots presented to the

user can be filtered depending on the frame that appears in the query.

ARGON is only a system to view information. The user cannot add, change, or delete information. Operations to add frames or individuals (using existing frames/individuals as templates), to edit existing frames or individuals, to restructure the frame hierarchy (through direct manipulation in the graphical display), and to delete frames or individuals are necessary.

19.4 HELGON

HELGON is still in its first steps, and its only advantage compared to ARGON so far is the graphical display of the frame hierarchy (Figure 19-4). Frames in the graphical display are mouse sensitive. Clicking on a frame with the left mouse button means adding it to the query. Frames that are required in the query are highlighted in the graphical display.

Figure 19-4: HELGON

20. Allen Cypher: Thought-Dumping

A few years ago I developed a personal database program, called Notepad. The "personal" part means that the person retrieving the notes is the same as the person creating them, so we assume that the organization is idiosyncratic, and we need not worry about users unfamiliar with the organization.

The main purpose of the program is to support thought-dumping: You get a sudden rush of ideas, and want to be able to jot them down before they are forgotten. Notepad is a support tool for managing the flow of ideas in one's stream of consciousness. Without a program for support, one is likely to pursue one particular train of thought and forget about numerous unpursued branches that could have been fruitful. In Notepad, interrupted notes are stored on a stack. It is not uncommon when using the program to press the Resume key and be presented with a partial idea that you had forgotten about.

Since the program is tailored for thought-dumping, its focus is on data entry rather than data retrieval. When brainstorming, the very act of verbalizing or writing down ideas makes those ideas more concrete and aids one's thought processes. When you write down your ideas, the benefit you get from making the ideas concrete is often of more importance than the benefit of being able to read those ideas later. Notepad's design reflects this focus by imposing only minimal structure on notes, and by allowing the user to postpone *all* structuring -- the user is never required to file a note or give it a title before moving on to some other note. Minimal structure makes it easy to create notes, at the expense of ease in later retrieval of those notes.

20.1 Postponed Structuring

Notes are organized in two ways: by throwing them into bins, and by summarizing them with a title (Figure 20-1). When you look in a bin, you see the titles of all of the notes stored there, so ease of retrieval motivates you to provide titles and to file related notes in a bin. Since the system will retrieve notes by searching for words in the title, you can retrieve notes without looking in bins. And since you can view notes that don't have titles, titling is not required. Furthermore, you can interrupt a note at any time and postpone titling and filing it until you return to that note at some later time. In summary, no structuring is required, and all structuring can be postponed indefinitely.

20.2 Minimal Structuring

Since titling and filing are the only organizational features of the system, users focus on the ideas at hand rather than on the use of the tool. This contrasts dramatically with other tools such as Xerox's NoteCards. In that system, there are a whole variety of link types, so that when a note is placed in a bin, the user specifies *how* the note is related to the bin (e.g., "source article" "example" "annotation") [Trigg 83]. The link types allow the user to specify more explicit information about relationships between notes, so the system can do more automatic processing of the information, perhaps by displaying annotations as marginalia and by listing the source or author of a note in a special field. When Kurt Van Lehn used NoteCards to represent his thesis work, he used the link types to organize hypotheses into a two-dimensional array. Missing spaces in the array showed him missing pieces in his argument [Lehn 85]. Notepad, in contrast, trades off this sort of computer assistance for the benefits of minimal cognitive load in using the tool.

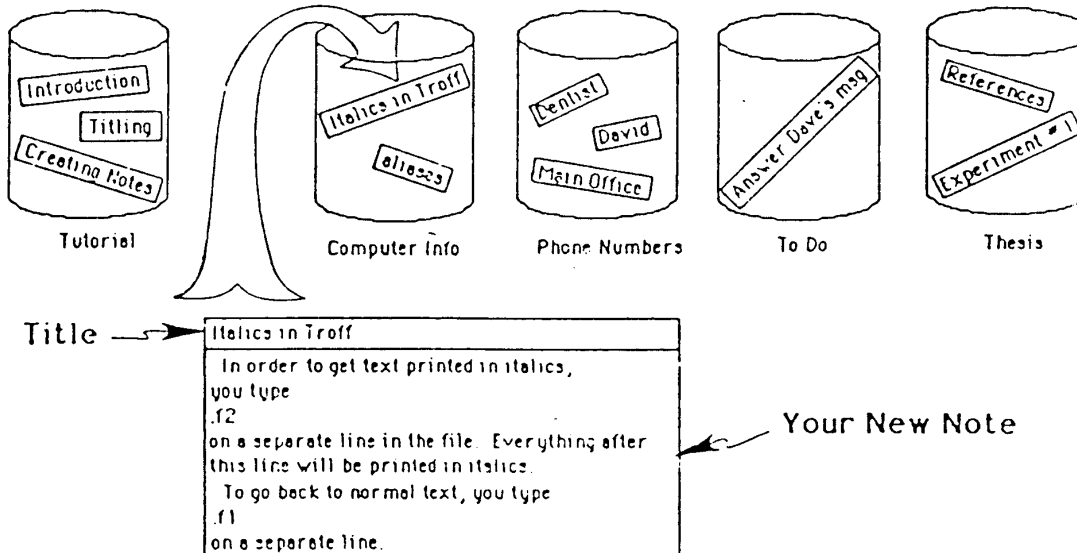


Figure 20-1: Filing a Note in a Bin

20.3 Tools versus Intelligence

Another way of characterizing Notepad's design is to say that it is a tool to support intelligence, as opposed to an intelligent tool. In other words, it is crafted to support a user's intelligence, as opposed to being an intelligent program. An example of the contrary approach can be found in Vic Lesser's programs which try to assist a computer user by figuring out what the user is trying to do. For instance, if the user edits a file and then compiles it, the computer assumes that the user will next want to run the compiled version of the file [Huff, Lesser 82].

20.4 Memory Organization and Retrieval

Some of the design principles in Notepad are relevant to psychological principles of memory organization. Titles for notes are in effect short summaries of the contents of the notes. The process of summarizing a new idea, of trying to determine a few-word description of that idea which can then be used as a label for that idea, is very helpful in making the idea more memorable and more palpable. So by encouraging summaries, Notepad facilitates the thought-dumping process.

The search retrieval algorithm matches on prefixes for any of the words in the title. This algorithm has proven to be quite effective in retrieving notes created several months in the past. It seems to map well on to the type of partial information that one remembers about a title.

People sometimes remember unrelated topics that they were pursuing at the same time as the note being searched for, in the same way that they often remember where they were or what they were doing when they had a particular idea. Since Notepad's notes are stored chronologically, it is possible to locate a note by examining the neighborhood of some more easily locatable note.

20.5 Hypertext

There is a popular personal database program on PCs called Think Tank (the newest version is called More). This program uses the Hypertext principle of a dynamic outline, where sections of interest can be expanded to the desired level of detail. In this way, the user gets a simultaneous view of detailed information and of the position of that information in the overall structure of the document.

Notepad employs a similar approach, based on Furnas' idea of a Fisheye view [Furnas 86]. The idea is to present a distorted view of an image, showing lots of detail for the most important part, and to surround that with a much cruder view of the surrounding context (Figure 20-2). In Notepad, there is a "Fisheye Outline" which automatically displays the current note, the children of the current note, the siblings of the current note, and the parents, grandparents, etc. of the current note. Depending on the number of children, siblings, and ancestors, and depending on the amount of space the user has allocated for the outline, the system has rules for determining how best to utilize the available space to provide an informative fisheye view.

Complete Outline:

- Cypher's Main Topics
  - Tutorial
    - Introduction
    - Creating a note
    - Topics
  - Computer info
    - troff
      - temporary indents
      - italics, bold, etc.
      - setting column widths in tbl
    - SUN workstations
      - large type
      - rsetup, tfetch, & rreturn
    - rcp
  - ToDo
    - three changes to search program
      - speed up search
      - exclude current note from Fileunder
      - ignore case in search
    - Suchman papers

Fisheye View:

- Cypher's Main Topics
  - --------
    - --------
    - --------
    - --------
  - -->> Computer info
    - troff
      - --------
      - --------
      - --------
    - ***SUN workstations
      - large type
      - rsetup, tfetch, & rreturn
    - rcp
  - --------
    - --------
      - --------
      - --------
      - --------
    - --------

Figure 20-2: A Fisheye View

20.6 Actual Usage

In actual usage, my personal experience was that the system was most useful for jotting down notes on how to use the computer -- for instance, on how to format references with our text processor. The main reason is that both while creating and retrieving these notes, you are already sitting at and using the computer. The organizational scheme of titles and topic bins was very effective for retrieving these notes at a later time.

In terms of thought-dumping, the system was very helpful in recalling partial trains of thought that would otherwise have been forgotten. However, I almost always used the system only for dumping thoughts, and not for reading them later. The main problem is that all of the ideas on a particular subject would be collected together in a bin, but the ideas were quite random and their complex interconnections -- the vague reasons why one idea was related to another -- were not explicit, and so the text seemed too random for later perusal. A truly effective use of Notepad for thought-dumping would require a second tool for organize these rough ideas into some sort of coherent structure -- perhaps a "rough draft" facility would be the appropriate step. To be effective, the rough draft tool would have to be used fairly soon after the thoughts were dumped, before their tenuous interconnections were forgotten.

Several other users of the system used it to collect notes for papers, and found it quite effective. I believe in this case that the users had enough conceptual structure in their heads to compensate for the minimal structuring of the tool.

References

[Aaronson, Carroll 87]
A. Aaronson, J.M. Carroll, *Intelligent Help in a One-Shot Dialog: A Protocol Study*, Human Factors in Computing Systems and Graphis Interface, CHI+GI'87 Conference Proceedings (Toronto, Canada), ACM, New York, April 1987, pp. 163-168.

[Anderson 82]
J.R. Anderson, *Acquisition of Cognitive Skill*, Psychological Review, Vol. 89, 1982, pp. 369-406.

[Boecker, Fischer, Nieper 86]
H.-D. Boecker, G. Fischer, H. Nieper, *The Enhancement of Understanding through Visual Representations*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 44-50.

[Brown 86]
J.S. Brown, *From Cognitive to Social Ergonomics and Beyond*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, ch. 22.

[Budge 83]
B. Budge, *Pinball Construction Set (Computer Program)*, Electronic Arts, San Mateo, CA, 1983.

[Card, Moran, Newell 83]
S.K. Card, T.P. Moran, A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Carroll, Carrithers 84]
J.M. Carroll, C. Carrithers, *Training Wheels in a User Interface*, Communications of the ACM, Vol. 27, No. 8, August 1984, pp. 800-806.

[Carroll, McKendree 85]
J.M. Carroll, J. McKendree, *Interface Design Issues for Advice-Giving Expert Systems*, Communications of the ACM, Vol. 30, No. 1, January 1985, pp. 14-31.

[Carroll, Rosson 86]
J.M. Carroll, M.B. Rosson, *Paradox of the Active User*, Technical Report RC 11638, IBM, Yorktown Heights, NY, 1986.

[Dayal, Wang 84]
U. Dayal, H. Wang, *View Definition and Generalization for Database Integration in a Multidatabase System*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, November 1984, pp. 628-644.

[Delisle, Schwartz 86]
N. Delisle, M. Schwartz, *Neptune: A Hypertext System for CAD Applications*, ACM SIGMOD'86 Proceedings, May 1986, pp. 132-143.

[Dijk, Kintsch 83]
T.A. van Dijk, W. Kintsch, *Strategies of Discourse Comprehension*, Academic Press, New York, 1983.

[Draper 86]
S.W. Draper, *Display Managers as User Interfaces*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, ch. 16.

[Fikes, Kehler 85]
R.E. Fikes, T.P. Kehler, *The Role of Frame-Based Representation in Reasoning*, Communications of the ACM, September 1985.

[Fischer 87a]
G. Fischer, *A Critic for LISP*, Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milano, Italy), August 1987.

[Fischer 87b]
G. Fischer, *Enhancing Incremental Learning Processes with Knowledge-Based Systems*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, Berlin - Heidelberg - New York, 1987.

[Fischer, Kintsch 86]
G. Fischer, W. Kintsch, *Theories, Methods and Tools for the Design of User-Centered Systems*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1986.

[Fischer, Lemke 87a]
G. Fischer, A.C. Lemke, *Design Kits: Steps Toward Human Problem-Domain Communication*, Paper Submitted to the Journal 'Human-Computer Interaction', Department of Computer Science, University of Colorado, Boulder, CO, 1987.

[Fischer, Lemke 87b]
G. Fischer, A.C. Lemke, *Constrained Design Processes: Steps Towards Convivial Computing*, in R. Guindon (ed.), *Cognitive Science and its Application for Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[Fischer, Lemke, Schwab 85]
G. Fischer, A.C. Lemke, T. Schwab, *Knowledge-Based Help Systems*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), ACM, New York, April 1985, pp. 161-167.

[Fischer, Schneider 84]
G. Fischer, M. Schneider, *Knowledge-Based Communication Processes in Software Engineering*, Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, March 1984, pp. 358-368.

[Furnas 86]
G.W. Furnas, *Generalized Fisheye Views*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 16-23.

[Gold 86]
E. Gold, *Envoys in Electronic Mail Systems*, Proceedings of the Conference on Office Information Systems, October 1986.

[Goldberg, Robson 83]
A. Goldberg, D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley Publishing Company, Reading, MA, 1983.

[Halasz, Moran, Trigg 87]
F.G. Halasz, T.P. Moran, R.H. Trigg, *NoteCards in a Nutshell*, Human Factors in Computing Systems and Graphis Interface, CHI+GI'87 Conference Proceedings (Toronto, Canada), ACM, New York, April 1987, pp. 45-52.

[Heimbigner, McLeod 85]
D. Heimbigner, D. McLeod, *A Federated Architecture for Information Management*, ACM Transactions on Office Information Systems, Vol. 3, No. 3, 1985, pp. 253-278.

[Hendler 86]
J.A. Hendler, *Integrating Marker Parsing and Problem-Solving: A Spreading-Activation Approach to Improved Choice in Planning*, Technical Report CS-86-1, Department of Computer Science, Brown University, Providence, RI, 1986.

[Huff, Lesser 82]
K. Huff, V. Lesser, *Knowledge-Based Command Understanding: An Example for the Software Development Environment*, Technical Report TR 82-6, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1982.

[Karat 83]
J. Karat, *A Model of Problem Solving with Incomplete Constraint Knowledge*, Cognitive Psychology, Vol. 14, 1983, pp. 538-559.

[Karat et al. 86]
J. Karat, L. Boyes, S. Weisgerber, C. Schafer, *Transfer Between Word Processing Systems*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 67-71.

[Kelly 75]
E.F. Kelly, P.J. Stone, *Computer Recognition of English Word Senses*, North-Holland, Amsterdam, 1975.

[Kieras 82]
D.E. Kieras, *A Model of Reader Strategy for Abstracting Main Ideas from Simple Technical Prose*, Text, No. 2, 1982, pp. 47-82.

[Kieras, Bovair 86]
D.E. Kieras, S. Bovair, *The Acquisition of Procedures from Text: A Production-System Analysis of Transfer of Training*, Journal of Memory and Language, Vol. 25, 1986, pp. 507-524.

[Kieras, Polson 85]
D.E. Kieras, P.G. Polson, *An Approach to the Formal Analysis of User Complexity*, International Journal of Man-Machine Studies, Vol. 22, 1985, pp. 365-394.

[Kintsch 87]
W. Kintsch, *The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension*, Technical Report 152, Institute of Cognitive Science, University of Colorado, Boulder, CO, 1987.

[Kintsch, Mannes 87]
W. Kintsch, S.M. Mannes, *Generating Scripts from Memory*, in J. Hoffmann, E. van der Meer (eds.), *Festschrift for F. Klix*, North-Holland, Amsterdam, 1987, also published as Technical Report No. 87-3, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[Lehn 85]
K. van Lehn, *Theory Reformulation Caused by an Argumentation Tool*, Technical Report, Xerox Palo Alto Research Center, March 1985.

[Lewis 86a]
C.H. Lewis, *Understanding What's Happening in System Interactions*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, ch. 8.

[Lewis 86b]
C.H. Lewis, *A Model of Mental Model Construction*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 306-313.

[Lewis 86c]
C.H. Lewis, *Why and How to Learn Why: Analysis-Based Generalization of Procedures*, Technical Report CU-CS-347-86, Department of Computer Science, University of Colorado, Boulder, CO, 1986.

[Lewis, Norman 86]
C.H. Lewis, D.A. Norman, *Designing for Error*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, ch. 20.

[Litwin, Abdellatif 86]
W. Litwin, A. Abdellatif, *Multidatabase Interoperability*, IEEE Computer, Vol. 19, No. 12, 1986, pp. 10-18.

[Malone et al. 86]
T.W. Malone, K.R. Grant, K.-Y. Lai, R. Rao, D. Rosenblitt, *Semi-Structured Messages are Surprisingly Useful for Computer-Supported Coordination*, Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'86), MCC, Austin, TX, December 1986, pp. 102-114.

[Malone, Grant, Turbak 86]
T.W. Malone, K.R. Grant, F.A. Turbak, *The Information Lens: An Intelligent System for Information Sharing in Organizations*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 1-8.

[Mittal, Dym, Morjarla 86]
S. Mittal, C.L. Dym, M. Morjarla, *PRIDE: An Expert System for the Design of Paper Handling Systems*, Computer, No. 7, 1986, pp. 102-114.

[Neuhold 86]
E.J. Neuhold, *Objects and Abstract Data Types in Information Systems*, Database Semantics (DS-1), IFIP, 1986.

[Neuhold, Walter 82]
E.J. Neuhold, B. Walter, *POREL: A Distributed Data Base Management System*, in H.J. Schneider (ed.), *Distributed Databases*, North-Holland, Amsterdam, 1982.

[Newell, Simon 72]
A. Newell, H.A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Norman, Draper 86]
D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[Patel-Schneider 84]
P.F. Patel-Schneider, *Small can be Beautiful in Knowledge Representation*, AI Technical Report 37, Schlumberger Palo Alto Research, October 1984.

[Patel-Schneider, Brachman, Levesque 84]
P.F. Patel-Schneider, R.J. Brachman, H.J. Levesque, *ARGON: Knowledge Representation Meets Information Retrieval*, Fairchild Technical Report 654, Schlumberger Palo Alto Research, September 1984.

[Phillips 78]
B. Phillips, *A Model for Knowledge and its Application to Discourse Analysis*, American Journal of Computational Linguistics, 1978.

[Phillips, Nicholl 86]
B. Phillips, S. Nicholl, *INGLISH: A Natural Language Interface*, in K. Hooper, I.A. Newman (eds.), *Foundation for Human-Computer Communication*, North-Holland, Amsterdam, 1986.

[Phillips, Staley, Gold 87]
B. Phillips, J. Staley, E. Gold, *Artificial Intelligence in Project Support*, in B.G. Barryman (ed.), *Expert Systems for Business*, Addison-Wesley Publishing Company, Reading, MA, 1987, forthcoming.

[Polson 87]
P.G. Polson, *Cognitive Factors in the Design and Development of Software in the Space Station*, in D. Kruser, S. Deutsch (eds.), *Humans in Automated and Robotic Space Systems*, Committee on Human Factors, National Research Council, 1987.

[Polson, Bovair, Kieras 87]
P.G. Polson, S. Bovair, D.E. Kieras, *Transfer Between Text Editors*, Human Factors in Computing Systems and Graphis Interface, CHI+GI'87 Conference Proceedings (Toronto, Canada), ACM, New York, April 1987, pp. 27-32.

[Polson, Kieras 85]
P.G. Polson, D.E. Kieras, *A Quantitative Model of the Learning and Performance of Text Editing Knowledge*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), ACM, New York, April 1985, pp. 207-212.

[Polson, Muncher, Engelbeck 86]
P.G. Polson, E. Muncher, G. Engelbeck, *A Test of a Common Elements Theory of Transfer*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 78-83.

[Polson, Muncher, Kieras 87]
P.G. Polson, E. Muncher, D.E. Kieras, *A Repair Model of Negative Transfer*, Technical Report 87-1, Institute of Cognitive Science, University of Colorado, Boulder, CO, 1987, forthcoming.

[Postman 71]
L. Postman, *Transfer, Interference, and Forgetting*, in J.W. King, L.A. Riggs (eds.), *Woodworth and Schlosberg's Experimental Psychology*, Holt, Rienhart, and Winston, New York, 1971.

[PostScript 85]
Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley Publishing Company, Reading, MA, 1985.

[Raaijmaker, Shiffrin 81]
J.G. Raaijmaker, R.M. Shiffrin, *Search of Assocative Memory*, Psychological Review, Vol. 88, 1981, pp. 93-134.

[Reder 85]
S.M. Reder, *Selection and Effects of Channels in Distributed Communication and Decision-Making Tasks*, Northwest Regional Educational Laboratory, Portland, OR, 1985.

[Schank, Abelson 77]
R. Schank, R.P. Abelson, *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.

[Schrefl, Neuhold 87]
M. Schrefl, E.J. Neuhold, *Data Bases and Information Bases, Architecture and Integration Issues*, Working Paper, GMD, 1987.

[Schrefl, Tjoa, Wagner 84]
M. Schrefl, A.M. Tjoa, R.R. Wagner, *Comparison Criteria for Semantic Data Models*, Proceedings of the IEEE COMPDEC Conference, 1984, pp. 105-133.

[Simon 75]
H.A. Simon, *Functional Equivalence of Problem Solving Skills*, Cognitive Psychology, Vol. 7, 1975, pp. 268-286.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.

[Singley, Anderson 85]
K. Singley, J.R. Anderson, *Transfer of Text-Editing Skills*, International Journal of Man-Machine Studies, Vol. 22, 1985, pp. 403-423.

[Smith et al. 83]
D.C. Smith, C. Irby, R. Kimball, B. Verplank, E. Harslem, *Designing the Star User Interface*, in P. Degano, E. Sandewall (eds.), *Integrated Interacitve Computing Systems*, North-Holland, Amsterdam - New York - Oxford, 1983, pp. 297-313.

[Stefik, Bobrow 86]
M.J. Stefik, D.G. Bobrow, *Object-Oriented Programming: Themes and Variations*, AI Magazine, Vol. 6, No. 4, Winter 1986.

[Stonebraker, Neuhold 77]
M.R. Stonebraker, E.J. Neuhold, *A Distributed Version of INGRES*, Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, University of California, Berkeley, CA, May 1977.

[Suchman 85]
L.A. Suchman, *Plans and Situated Actions: The Problem of Human-Machine Communication*, Technical Report ISL-6, Xerox Palo Alto Research Center, Palo Alto, CA, February 1985.

[Thorndike 14]
E.L. Thorndike, *The Psychology of Learning*, Teachers College, New York, 1914.

[Thorndike, Woodward 01]
E.L. Thorndike, R.S. Woodward, *The Influence of Improvement in one Mental Function upon the Efficiency of other Functions*, Psychological Review, Vol. 8, 1901, pp. 247-261.

[Tou et al. 82]
F.N. Tou, M.D. Williams, T.W. Malone, R.E. Fikes, A. Henderson, *RABBIT: An Intelligent Interface*, Technical Report, Xerox Palo Alto Research Center, 1982.

[Tou, Williams 82]
F.N. Tou, M.D. Williams, *RABBIT: An Intelligent Database Assistant*, Technical Report, Xerox Palo Alto Research Center, 1982.

[Trigg 83]
R.H. Trigg, *A Network-Based Approach to Text Handling for the Online Scientific Community*, Technical Report TR-1346, Maryland Artificial Intelligence Group, Department of Computer Science, University of Maryland, College Park, MD, November 1983.

[Trigg, Suchman, Halasz 86]
R.H. Trigg, L.A. Suchman, F.G. Halasz, *Supporting Collaboration in NoteCards*, Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'86), MCC, Austin, TX, December 1986, pp. 153-162.

[Turner 87]
A.A. Turner, *The Propositional Analysis System Version 1.0*, Technical Report 87-2, Institute of Cognitive Science, University of Colorado, Boulder, CO, March 1987.

[Turner, McCutchen 87]
A.A. Turner, D. McCutchen, *The Generation of Macropropositions During Comprehension*, 1987, forthcoming.

[Walker, Kintsch 85]
W.H. Walker, W. Kintsch, *Automatic and Strategic Aspects of Knowledge Retrieval*, Cognitive Science, No. 9, 1985, pp. 261-283.

[Weyer 82]
S.A. Weyer, *The Design of a Dynamic Book for Information Search*, International Journal of Man Machine Studies, Vol. 17, No. 1, July 1982, pp. 87-107.

[Weyer 87]
S.A. Weyer, *As We May Learn*, Multimedia in Education: Interfaces to Knowledge, Education Advisory Council Conference Proceedings, Apple Computer, April 1987.

[Weyer, Borning 85]
S.A. Weyer, A.H. Borning, *A Prototype Electronic Encyclopedia*, ACM Transactions on Office Information Systems, Vol. 3, No. 1, January 1985, pp. 63-88.

[Williams 84]
M.D. Williams, *What Makes RABBIT Run?*, International Journal of Man-Machine Studies, Vol. 21, 1984, pp. 333-352.

[Williams, Hollan 81]
M.D. Williams, J.D. Hollan, *The Process of Retrieval from Very Long-Term Memory*, Academic Press, New York, 1981.

[Winograd, Flores 86]
T. Winograd, F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ, 1986.

[Ziegler, Vossen, Hoppe 86]
J.E. Ziegler, P.H. Vossen, H.U. Hoppe, *Assessment of Learning Behavior in Direct Manipulation Dialogues: Cognitive Complexity Analysis*, Working Paper B.3.3, ESPRIT Project 385 HUFIT, Fraunhofer-Institut fuer Arbeitswirtschaft und Organisation, Stuttgart, West Germany, 1986.

Appendix: List of Participants

Guests:

John Carroll
User Interface Institute
IBM Watson Research Center
Yorktown Heights, NY 10589
phone: (914) 789-7743

Allen Cypher
Intellicorp
1975 El Camino Real West
Mountain View, CA 94040-2216
phone: (415) 965-5500

Kate Ehrlich
Manager, User Interface Group
Symbolics Inc
Eleven Cambridge Center
Cambridge, MA 02142
phone: (617) 577-7560

Robin Jeffries
Hewlett Packard Research Labs
1501 Page Mill Road
Palo Alto, CA 94304
phone: (415) 857-8784

Erich Neuhold
Director, Institute for Electronic Publishing
GMD
Rheinstrasse 75
D-6100 Darmstadt
W-Germany
phone: 011-49-6151-869-289

Brian Phillips
Principal Computer Scientist
Computer Research Laboratory
Tektronix, Inc.
P.O. Box 500, Mail Sta. 50-662
Beaverton, OR 97077
phone: (503) 627-627-1119

Stephen Weyer
Applications Technologies Advanced Development
Apple Computer, Inc
20525 Mariani Avenue, MS: 22C
Cupertino, CA 95014
phone: (408) 973-6131

University of Colorado, Institute of Cognitive Science and Department of Psychology, Campus Box 345, Phone (303) 492-5063:

- Walter Kintsch
- Althea Turner
- Suzanne Mannes
- Peter Polson

University of Colorado, Institute of Cognitive Science and Department of Computer Science, Campus Box 430, Phone (303) 492-1592:

- Brigham Bell
- Hal Eden
- Gerhard Fischer
- Andreas Lemke
- Clayton Lewis
- Helga Nieper
- Christian Rathke
- Paul Smolensky
- Emilie Young