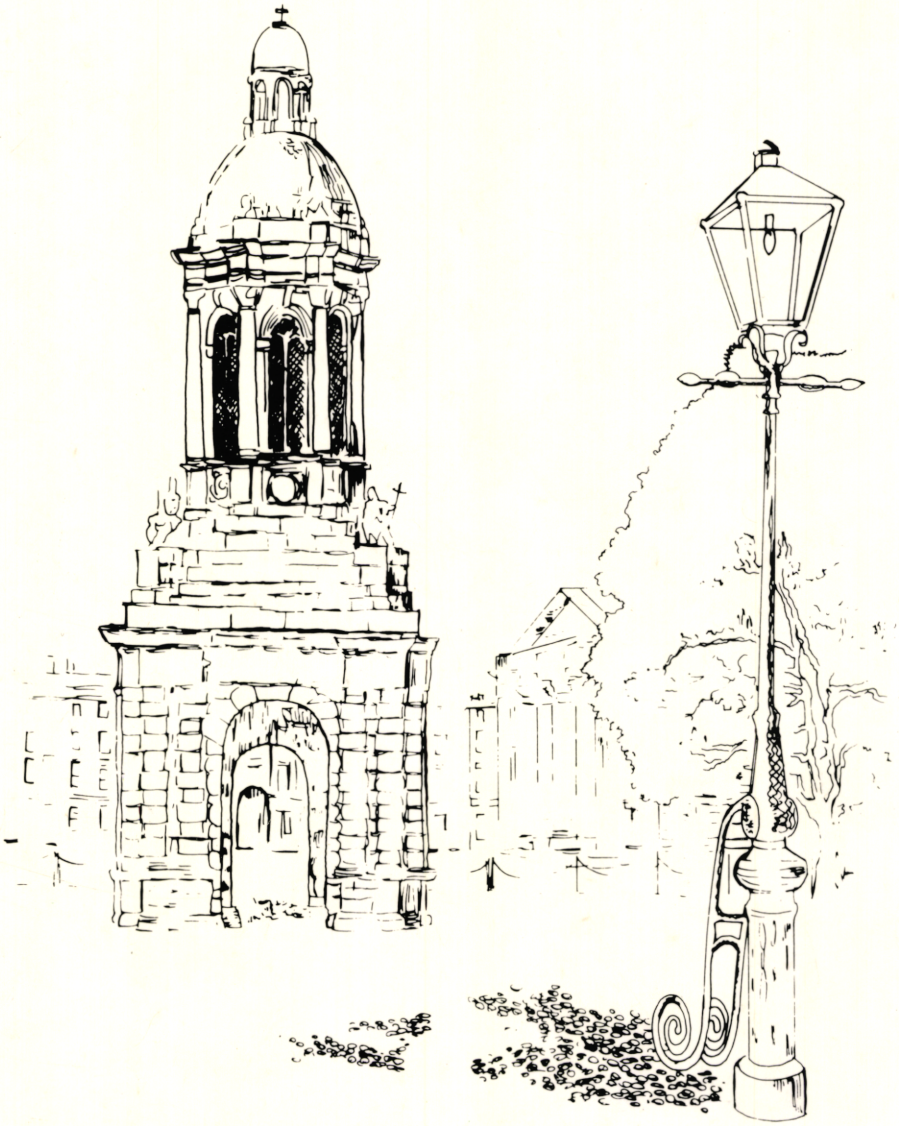
EUUG

European UNIX ® systems User Group

CONFERENCE PROCEEDINGS

Trinity College, Dublin, IRELAND



SEPTEMBER 1987

EUUG

European UNIX[1] systems User Group

AUTUMN '87

CONFERENCE PROCEEDINGS

Trinity College, Dublin, IRELAND

September 21–25 1987

[1] UNIX is a registered trademark of AT&T in the United States of America and other countries

UNIX Conferences in Europe 1977–1987	
UKUUG/NLUUG meetings	
1977 May	Glasgow University
1977 September	University of Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University
1978 November	Dutch Meeting at Vrije University, Amsterdam
1979 March	University of Kent, Canterbury Brian Kernighan and Ken Thompson
1979 October	University of Newcastle
1980 March 24th	Vrije University, Amsterdam Steve Johnson
1980 March 31st	Heriot Watt University, Edinburgh
1980 September	University College, London
EUUG Meetings	
1981 April	CWI, Amsterdam, The Netherlands Dennis Ritchie
1981 September	Nottingham University, UK
1982 April	CNAM, Paris, France
1982 September	University of Leeds, UK
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Ireland
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, UK
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, UK
1987 May	Helsinki/Stockholm, Finland/Sweden

EUUG Secretariat
Owles Hall
Buntingford
Herts SG9 9PL
United Kingdom
+353 763 73039
euug@inset.UUCP

ACKNOWLEDGEMENTS

Many people have contributed to the production of this volume. It is not possible to thank them all individually, but the following deserve special note. Hugh Grant, of the School of Mathematics, Trinity College, who helped with the production of this volume. [2] Timothy Murphy, also of the School of Mathematics, who was the local organiser. Michel Gien, of Chorus SYSTEMES who was on the program committee chaired by myself, Simon Kenyon.

Many thanks also to my employers, ICL — Information Technology Centre, Dublin; who gave me the time to organise this conference.

These proceedings have been produced with the help of the School of Mathematics, Trinity College, Dublin. and in particular I would like to thank Prof. John Miller, Prof. David Simms and Dr. Richard Timoney for their invaluable assistance.

And of course, thanks to all the authors who submitted their paper in time. [3]

Finally it is appropriate to thank the many authors who submitted very good papers, but were not selected for this programme. It is always very difficult to perform the selection, and it is becoming more so as the quality of papers increases. In many cases, the reason for rejection was not that the paper was in any way below standard; rather that the topics addressed were not appropriate for the material planned for this conference. Such authors should, if appropriate, submit their papers again for the next EUUG conference in London.

[2] evenifhiskeyboardhasabrokenspacebar

[3] Some people will be appalled to learn that some of the papers were converted from troff to LaTeX. The exact mode of production is too horrific to divulge.

TIMETABLE

	Wednesday	Thursday	Friday
9.30	WELCOME		
10.00	Solange Karsenty A Framework for Man Machine Interfaces Design	Allan Milne The Analysis and Manipulation of BNF Definitions	Lori Grob Automatic Exploitation of Concurrency in C, Is it really so hard?
10.30	Coffee/Tea	Coffee/Tea	Coffee/Tea
11.00	Mike O'Dell What They Don't Tell You About Window Systems	Barry Lynch Uncle — A Case Study in Constructing Tools for the PCTE	Stuart Borthwick An Intelligent, Window Based Interface To UNIX
11.30		Christoph Senft A Distributed Design Environmemt for Distributed Realtime Systems	Bart Locanthi Fast bitblt() with asm() and cpp
12.00	Chris Crampton Musk — A Multi User Sketch Program	Dominico Talia NERECO : An Environment for the Development of Distributed Software	
12.30	Lunch	Lunch	Lunch
13.00			
13.30			
14.00	Dan Klein UBOAT — A UNIX Based On-line Aid to Tutorials	Michel Ricard A Knowledge Based CAD System in Architecture on UNIX	Stephen Beer DES — Support for the Graphical Design of Software
14.30	Seamus Kearney Developing ADA Software using VDM in an Object-Oriented Framework	Roger Bivand A User Interface for Geographers — What can UNIX offer?	David Tilbrook Cleaning up UNIX Source, or Bringing Discipline to Anarchy
15.00	Chris Chedgey Papillon — Support Tools for the Development of Graphical Software	Mike O'Dell The HUB : A Lightweight Object Substrate	
15.30	Coffee/Tea	Coffee/Tea	Coffee/Tea
16.00	Gerritt van der Veer Experiments with the User Interface for UNIX Mail	Mark Abdelnour Graphical User Interfaces on Multiuser Systems	EUUG Business Meeting
16.30	Neil Groundwater A SunView User-Interface for Authoring and Accessing a	Mike Hawley More MIDI Software for UNIX	
17.00	Medical Knowledge Base		GOODBYE

Table of Contents

A Framework for Man Machine Interfaces Design, 1
Michel Beaudouin-Lafon, Solange Karsenty

What They Don't Tell You About Window Systems, 11
Mike O'Dell

Musk — a Multi User Sketch Program, 17
Chris Crampton

UBOAT — A Unix Based On-line Aid to Tutorials, 31
Dan Klein

Developing Ada [4] Software using VDM in an Object-Oriented Framework, 41
Chris Chedgey, Seamus Kearney, Hans-Jürgen Kugler

Papillon — Support Tools for the Development of Graphical Software, 59
Chris Chedgey

Experiments with the User Interface for UNIX Mail, 73
Peter Innocent, Gerrit van der Veer, Yvonne Waern

A SunView User-Interface for Authoring and Accessing a Medical Knowledge Base, 93
Neil Groundwater, Neil Bodick, Andre Marquis

The Analysis and Manipulation of BNF Definitions, 105
Allan Milne

Uncle - A Case Study in Constructing Tools for the PCTE, 123
Hans-Jürgen Kugler, Barry Lynch

A Distributed Design Environment for Distributed Realtime Systems, 131
Christoph Senft

NERECO : An Environment for the Development of Distributed Software, 153
Giandomenico Spezzano, Domenico Talia, Marco Vanneschi

A Knowledge Based CAD System in Architecture on UNIX, 169
E. Chouraqui, D. Dugerdil, P. Francois, S. Hanrot, P. Quintrand, M. Ricard, J. Zoller

A User Interface for Geographers — What can UNIX offer? 183
Roger Bivand

The HUB : A Lightweight Object Substrate, 191
Mike O'Dell

Graphical User Interfaces on Multiuser Systems,
Mark Abdelnour

More MIDI Software for UNIX, 201
Mike Hawley

Automatic Exploitation of Concurrency in C, Is it really so hard? 209
Lori Grob

An Intelligent, Window Based Interface to UNIX, 225
Stuart Borthwick, John Nicol, Gordon Blair

Fast bitblt() with asm() and cpp, 243
Bart Locanthi

DES — Support for the Graphical Design of Software, 261
Stephen Beer, Ray Welland, Ian Sommerville

Cleaning Up UNIX Source, or Bringing Discipline to Anarchy, 275
David Tilbrook, Zalman Stern

SPEAKERS' ADDRESSES

Mark Abdelnour
NCR Corporation
Engineering and Manufacturing
3325 Platt Springs Road
West Columbia
South Carolina 29169
USA
+1 803 796 9250
abdel@ncrcae.Columbia.NCR.COM

Stephen Beer
Software Technology Research Group
Department of Computer Science
Livingstone Tower
University of Strathclyde
Glasgow G1 1XW
Scotland
+44 41 552 4400 ext 3390
stephen@cs.strath.ac.uk

Roger Bivand
Høgskolesenteret i Nordland
P.O. Box 6003
N-8016 Markved
Norway
+47 81 17 457

Stuart Borthwick
Department of Computing
University of Lancaster
Lancaster
England
stuart@dcl-cs.UUCP

Chris Chedgey
Generics (Software) Limited
7 Leopardstown Office Park
Foxrock
Dublin 18
Ireland
+353 1 954012
chedgey@genrix.UUCP

Chris M. Crampton
Human Computer Interaction Section
Informatics Division
Rutherford Appleton Laboratory
Chilton
Didcot
Oxon OX11 0QX
England
+44 235 21900 ext 6746
cmc@vd.rl.ac.uk

Lori S. Grob
Courant Institute (NYU)
251 Mercer Street
New York
New York 10012
USA
+1 212 460 7326
grob@cmcl2.nyu.edu

Neil Groundwater
Sun Microsystems, Inc.
8219 Leesburg Pike # 700
Vienna
Virginia 22170
USA
+1 703 883 0444
npg%sundc@sun.com

Mike Hawley
Next Inc.
3475 Deer Creek Road
Palo Alto
CA 94304
+1 415 424 0200
mike@media-lab.mit.edu

Solange Karsenty
Laboratoire de Recherche en Informatique
Universite de Paris-Sud - Bat. 490
91405 Orsay Cedex
France
so@lri.lri.fr

Daniel V. Klein
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh
PA 15213
USA
+1 412 268 7791
dvk@sei.cmu.edu

Seamus Kearney
Generics (Software) Limited
7 Leopardstown Office Park
Foxrock
Dublin 18
Ireland
+353 1 954012
kearney@genrix.UUCP

Barry Lynch
Generics (Software) Limited
7 Leopardstown Office Park
Foxrock
Dublin 18
Ireland
+353 1 954012
lynch@genrix.UUCP

Allan C. Milne
Dundee College of technology
Dept. of Maths & Computer Studies
Bell Street,
Dundee
DD1 1HG
Scotland
+44 382 27225 ext 299

Mike O'Dell
MAXIM Technologies, Inc.
8618 Westwood Center Drive
Vienna
VA 22180
USA
+1 703 893 3660
mo@maximo.UUCP

M. Ricard
GRTC/CNRS,
31, Chemin Joseph Aiguier
13402 MARSEILLE
Cedex 9
France
+33 91 22 40 00

Christoph Senft
Institut fuer Technische Informatik
Technische Universitaet Wien
Gusshausstrasse 30
A-1040 Vienna
Austria
+43 222 588 01
senft@vmars.UUCP

Domenica Talia
CRAI
Localita S. Stefano
87036 Rende (C.S.)
Italy
+39 984 833255
dot%crai@i2unix

Gerrit van der Veer
Subfaculteit der Psychologie
en Pedagogische Wetenschappen
Vakgroep Funktieleer en Methodenleer
Vrije Universiteit
Postbus 7161
1007 MC Amsterdam
The Netherlands
+31 20 548 3868

A FRAMEWORK FOR MAN MACHINE INTERFACES DESIGN

Michel Beaudouin-Lafon and Solange Karsenty
Laboratoire de Recherche en Informatique - UA 410 du CNRS
Université de Paris-Sud - Bât. 490
91405 Orsay Cedex - France
Tel : (1) 69 41 66 29
e-mail : mbl@sun1.lri.fr so@lri.lri.fr

Abstract

With the increasing importance of man-machine interfaces appears the lack of tools for the construction of user interfaces. Some principles are coming out, such as the separation of the interface from the application, and allow us to start designing environments for their construction. To facilitate their construction means to design fast, to modify easily and eventually to reuse. On the user side, the interface should be dynamically adaptable to his taste and experience.

Towards such interfaces, we propose an environment built upon two tools : Graffiti and MetaGraph. The first one allows for interactive construction of the control structure of the interface, while the second one establishes a relation between the program data and the interface. This correspondance of high level abstractions will extend the functionalities of the interface by reducing graphic manipulations in the application.

Keywords : construction of man machine interfaces, interaction, object oriented system, flexibility.

1 - Introduction

Man machine interfaces (MMI) are now considered as a real part of sofware engineering. The lack of tools, and first of all the lack of abstractions, to design MMI prove that this area requires more experimentations. The complexity of the MMI problems is due to the fact that human factors and programming factors are closely dependent.

We propose a model for the design of man machine interfaces. Our aims are :

- separation of the interface from the application
- easy reconfiguration of the interface
- handling of standard functions such as *selection, cut, copy, paste*

Separating the interface from the application has many advantages : for the implementation, and for the reusability of the interface. Reconfiguration leads to more flexible interfaces in order to adapt dynamically the interface to the user, and not the reverse as it is most of the time. Separation and reconfiguration are also useful to build homogeneous interfaces to different applications, or to upgrade an interface to different kinds of users.

The separation between the interface and its application can be easily achieved, as long as the interface does not directly manipulate objects belonging to the application. Some functions such as selection, are often part of an application. They involve both a specific interaction and a specific action on the objects of the application. Integrating such functions in the interface requires the definition of a protocol with the application. We show that such a protocol can be very flexible as it does not constraint the actions to be performed by the interface or by the application, but rather describes how these actions take place.

The overall structure of the system is composed of the application, which is accessed by the interface through a set of entry points. The interface itself is made of a control manager and a data manager. The control manager is generated with Graffiti : it defines a set of objects such as windows, menus, icons and other interactive objects, to control the application. It also processes the input devices and sends them to the application or to the data manager, by means of a mapping between input events and entry points. The data manager is generated with MetaGraph : it defines a data representation model, such as lists, trees, graphs, and the mapping between this representation and the application data.

Both Graffiti and MetaGraph are interactive. This means that the control objects and the representation model are described through graphic editors. A part of these editors can be included in the generated interface to provide *dynamic* modification of the interface by the end user. These tools are built upon UFO, an object oriented system especially adapted to graphic interfaces and manipulation of graphical objects.

2 - An object oriented architecture

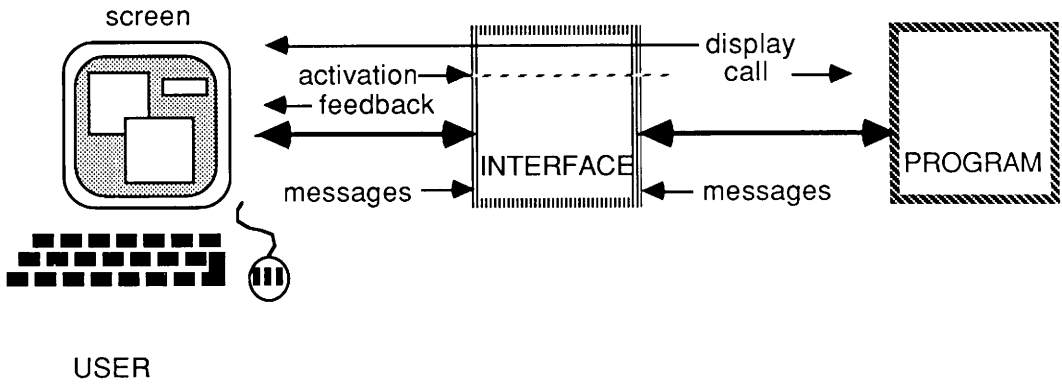
Several works have been done to provide high level tools to manipulate and build interfaces [Hanusa83, Hayes85]. Our goal is to build an environment of tools, commonly named a UIMS (User Interface Management System) [Kasik82], providing a variety of style of dialogue and a standard protocol between the interface and several kind of applications.

Each interface has two sides : one to the user and the other to the system to be interfaced. Both need to be adaptable. The user wants to adapt the interface dynamically to his taste and knowledge, and the connections between the interface and the program need to be easily and dynamically changed. The object oriented approach [Goldberg83] with the message passing mechanism is well-suited to provide such a scheme (figure 1) : it provides high level abstractions to define the protocols between the different components. The hierarchy of classes corresponds to a taxonomy of graphical objects [Barth86]. The consequence of this approach is that any interface of this kind could be re-usable for other programs.

2.1 - Some typical points

Interaction

Programming MMI is often closely related to graphical programming. There is today no programming language really adapted to graphics. What we want is to provide the designer the ability to perform actions such as drawing a rectangle as fast as you say it : we mean interactively. The MMI designer works usually with a finite set of graphic objects to be created, displayed, deleted and so on... Allowing these operations to be interactive means



- *Figure 1* -

that we propose a new way of programming : visual programming. The power of such a language is of course limited, but completely dedicated to MMI design.

In the MMI area, prototyping is essential for the quality of the interfaces. Their development is so long and so hard that no one would think of testing the interaction with the user, once it is done. Giving a specific way of designing MMI that allows easy prototyping will contribute to the development of more powerful, homogeneous and consistent interfaces. Finally, the benefit will go to the most concerned person : namely the user.

what kind of MMI ?

The interfaces we are dealing with are composed of graphic interactive objects such as icons, menus, windows, scroll-bars, buttons. Some of these objects are commonly included in a "toolbox" yet this is not a standard fact. The toolbox belongs to a specific machine while our system offers an extensible set of graphic objects. These objects can be activated in many different ways.

The dialog with the user has many levels. The activations of the different objects named above, through keyboard sequences or mostly mouse clicks, bring more or less explicit answers to the user. Sometimes it is just a graphic signal, sometimes it is a full text message. This feedback depends on the user's knowledge. That is why we want MMI to be as much as possible adaptive to provide a large range of styles of interaction.

Intended applications

Lots of application needs a MMI. We don't pretend to give a universal tool but we think that our system will be useful and efficient in most cases. The part (Graffiti) that deals with objects belonging to the interface is really application independant. More power to the interface means more complexity and involves coordination with the application. This coordination leads us to make some suppositions : the application is accessed through a set of entry points, and the top-level is included in the interface. The application can call the interface from its entry points, but it should not enforce any constraining dialog with the user. Rather the user should be able to do anything at any moment, and thus the application

should be prepared to do so.

The application manipulates objects to be displayed and connected. For example, a Petri-net editor manipulates transitions, places and edges. An edge is connected from one side to a place and from the other side to a transition. More generally, the application deals with a graph having specific constraints. In this case, we are able to represent this structure (MetaGraph) and to perform directly from the interface some operations like selection.

2.2 - Object oriented approach : UFO

To really provide an adaptable environment of this kind, we need abstractions. The tools we present are based on **UFO** [Beaudouin-Lafon85a] [Beaudouin-Lafon85b], an object oriented system specially adapted to interfaces and manipulation of graphical objects. One important point is that UFO is portable on many UNIX machine with bitmaps graphics and a window manager. Actual implementations include ICL Perq/PNX, SUN workstations with XWindows [Scheifler86], HP9000 series 300. Thus the tools described in this paper and the interfaces they generate are portable as well – provided the application itself is portable. UFO is written in C, as a functions library.

The purpose of **UFO** is to separate external and internal representation, and to associate with an object one internal representation, and one or more external representations. The internal representation holds the description of the objects while the external one is generated and manipulated through messages sent to the internal representation. Thus an application using UFO needs just to maintain the mapping between these two representations.

UFO objects are complex entities as they are made of a graph, called the object graph. Vertices and edges of this object graph are typed and attributed. Objects can be hierarchised with the notion of sub-object : a vertex of an object graph can be a reference to another object. This makes possible to manage very complex objects in a modular way. Messages can be sent to objects, and to vertices and edges of the object graph. Moreover, iterators allow to enumerate the structure in various ways, easily and efficiently.

Beside the manipulation of the internal representation, UFO contains a dedicated graphic library. This library is a uniform, evolutive, and low-cost layer between the applications and today's graphic worsktations which share the bitmap graphics hardware concept, and the window manager software concept, with distinct and incompatible implementations. Thus, it allows to write highly portable graphic software.

3 - Components

3.1 - Metagraph

Lots of applications can use graphs to represent the data they operate on : Petri-nets, SADT diagrams, flow-chart programs, PERT diagrams, ... When they do not, the user often uses a sheet of paper instead. The aim of Metagraph is to provide a tool to build graph editors and make them cooperate with such applications.

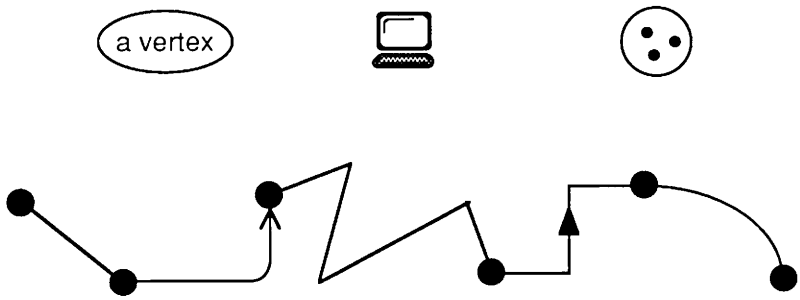
Metagraph is basically a very general graph editor that can be parameterized to fit some constraints. There are two main kinds of constraints : syntactic constraints and layout constraints. Syntactic constraints include constraints on the structure of the graph : for

instance, in a Petri-net, edges can connect a place and a transition, but not two places or two transitions; some graphs should not have cycles, etc ... Layout constraints include all the constraints to build a graphical representation of the graph : for instance, edges must be made exclusively of horizontal and vertical segments, or the vertices should not overlap, etc ...

More than a parameterizable graph editor, Metagraph is a meta-editor : it generates an editor to be used as a front-end for applications. This means that Metagraph manages the link between the generated editor and the application. This of course implies some constraints on the applications.

The general graph editor of Metagraph is based on the following features :

- vertices : a vertice can be a simple graphic shape, or a UFO object. It can even be a Metagraph object.
- edges : edges can be simple, oriented or not, or multi-edges : this means they can have several end vertices. There are several graphical representations available (see figure 2).
- annotations : annotations can be attached to any vertex, edge or group of them. An annotation can be for instance a comment, or a graphic shape as a rectangle around a set of vertices.
- constraints : as annotations, constraints can be attached to groups of vertices and/or edges. They are most of the time layout constraints : keep two vertices aligned or with a minimal distance between them, ... Syntactic constraints are handled in a different way : they are defined as rules and these rules are checked and enforced when editing a graph.



- Figure 2 -

Metagraph comes with a predefined set of vertices, edges, annotations and constraints, but it can be extended to include more. The above features describe the components of the graphs managed by Metagraph. To manipulate these graphs, a set of operations are included. Most of them use the concept of selection : at any one moment, a subset of the graph being edited is called the selection. The selection can be changed by the user, and most operations act on the selection. This makes the interaction easier and homogeneous. Here is a quick overview of the operations provided :

- Select : several operations exist to change the selection : select one, add to / remove from selection, select all, ...
- Create : create a vertex, an edge, an annotation. The actual interaction protocol for the creation depends on the object to create : creating a multi-segment edge will require several user actions to define the intermediate points while creating a vertex is usually easier. When creating an annotation, the current selection defines the subset of the graph to which is linked the annotation.
- Cut / Copy / Paste : these operations act on the current selection. Cut deletes the selection and saves it, Paste inserts the last deleted selection, Copy saves the selection without deleting it.
- Move : move the selection. Layout constraints are checked.
- Edit : edges, vertices and annotations can be edited : for instance the size of a vertex can be changed or the text of a comment can be edited.

Using Metagraph means first to define within these available sets which items to use. This is the "easy" part, as it is a static definition of the items that will be used in the graphs. The second part concerns the dynamic behaviour of the graph on the screen. This behaviour will be described with scripts. Each script will be activated through an interface entry point, that is a function of the editor generated by Metagraph.

Scripts can involve user input actions (clicks, keystrokes ...), Metagraph operations, and application entry points. The latter will be used to implement a control on the user action by the application. Metagraph operations include the basic operations on the graph and utilities. Utilities include for instance functions to browse through the graph. This is useful when a script needs to check and eventually modify the selection before operating on it.

As an exemple, consider the delete script. It is intended to delete the current selection. Most of the time the selection must be adjusted to contain exactly what needs to be deleted. For instance all edges connected to vertices in the selection must also be selected. This can be described in the script. If the rules are more complex, a function can be written directly and called from the script.

As another example, consider the creation of an edge. Creating an edge consists in defining two vertices and creating an edge between them. One can suppose that one of the vertices is the selection, while the other must be selected by the user. The script must check that the selection contains exactly one vertex, then wait a click, check that a vertex has been selected, check that the edge can be created between these two vertices without breaking any constraint, if so create the edge and call an application entry point to inform that an edge has been created. Such scripts are very standard and they are included in Metagraph so that one needs just to edit them instead of creating them from scratch.

At any moment, it is possible to test the scripts that are being defined. Of course in this case there is no application and it is simulated, but this provides a very fast and effective way of prototyping the interface.

The result of a session with Metagraph is a source file to be compiled containing an initialization function and one function per script defined. Another file contains declarations to be included in the application so that it can access the interface. Of course a session can be saved so that it can be reentered to modify the definitions and scripts.

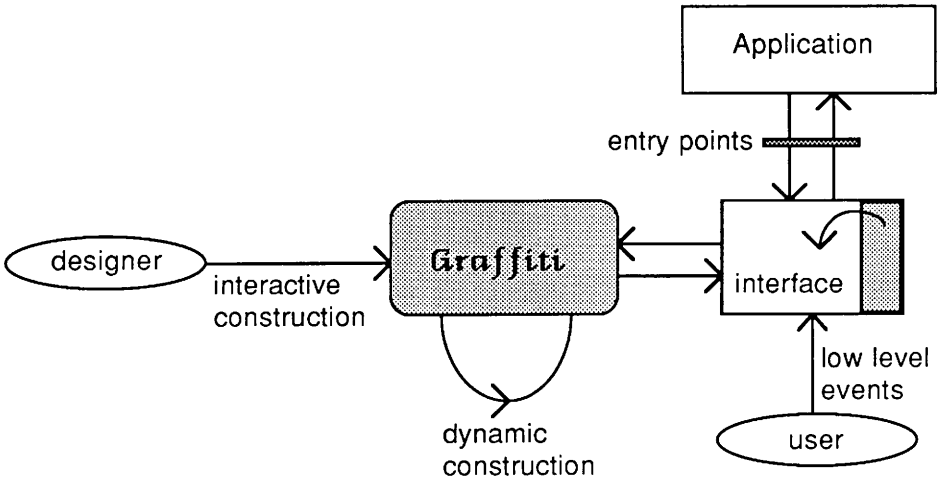
3.2 - Graffiti

Graffiti [Karsenty86] is an interactive tool for the design of flexible graphic interfaces. Its functionalities range from the construction of the overall structure of the interface, the dialogue with the user, to the link with the application.

The objects manipulated are high level graphic objects : there is a set of basic objects and they can be combined to build more complex objects. The interface has its own internal structure while the user manipulates and sees the external representation. The interface is not static : the user as well as the application can modify dynamically any part of the interface. This possibility allows to build adaptive interfaces : adaptability corresponds to the amount of Graffiti's self functionalities included in the interface constructed by Graffiti. Figure 3 shows the steps for the construction of an interface. The designer builds interactively an interface and includes some functionalities of Graffiti in it. These functionalities can be divided in five parts :

- interaction style
- editing menus, scrollbars, icons...
- feedback to the end-user
- screen layout
- link to the application

The interaction style and the feedback are the parts that should be accessible for the user. The others are for special purpose since they operate directly on the functionalities of the interface.



- Figure 3 -

Graffiti is build on top of UFO classes. There are basic classes to represent windows, icons, scrollbars (integers), buttons and menus. These objects are combined together. The kernel of Graffiti manipulates these classes in order to create the bricks of an interface. The glue which is necessary to establish the relations between those objects is put in a super class. This class, called *Screen*, manipulates objects of basic classes and adds the semantic relations between the structures.The result is a graph of objects. It also manages the screen display and dispatches the events coming from the mouse or the keyboard. The classes of Graffiti are not exclusively for Graffiti. An alternative is to use them directly as packages to program some other application.

The application is accessed by the interface through entry points (Figure 3). The menus' commands, the windows regions, the mouse clicks and keyboard sequences can be attached to any of these entry points. The top level generated by Graffiti scans all kind of low level events and finds either an entry point to call directly some function, or the object to which was send this event. In the last case, it is mostly a message passing.The top level is defined this way. An event associated to an object can be linked to a specific message. For instance, a click in the main region of a window will send the message Move-Window to that one. A click on the desktop displays a pop-up menu, waiting for the selection of a command. The control of the application is partially defined : the control of the application's data is done with MetaGraph. Graffiti only takes control on the objects private to the interface. This control is not static : the links can be redefined at any time. Some standard interface is quickly designed, and the architecture allows reusability by plugging other applications to the same interface. Therefore, it provides a powerful tool for testing and prototyping several applications.

A menu is an object with a set of vertices to be displayed, selected, scrolled, edited : its commands are given in the vertices of the object graph. Each vertex contains some attributes and this class is generic : a command can belong to any class and have its own external and internal representation. This genericity provides an infinite number of possible menus. The mechanism of sub-objects correspond exactly to the hierarchy of menus : each command (vertex) can be a node leading to some other menu (sub-object). The interface is an object of the class *Screen*. The nodes of the object graph are sub-objects of basic classes. Dependencies are expressed through edges between vertices. An icon can represent a window, a menu can belong to a window, and so...

The application designer builds a file containing a list of entry points of the application. The interface designer specifies interactively and graphically the interface. Then Graffiti generates the top level of the interface. The description of the interface is saved in another file to be read at the initialization and this description can be reloaded in order to modify any part of it.

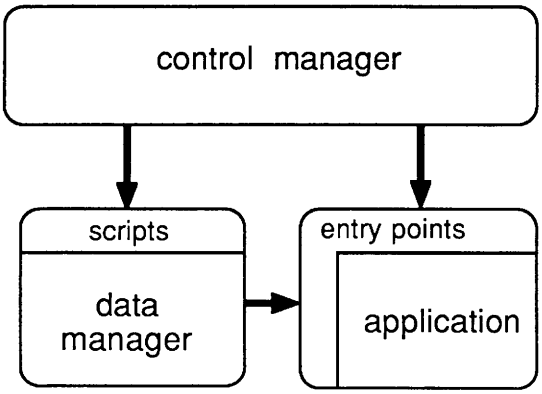
3.3 - Integration

Integration is the phase where the control manager, the data manager and the application are linked together. The connections between these three parts are depicted in figure 4. This figure shows that from the point of vue of the control manager, the data manager is an extension of the application : this extension supports the graphical representation and manipulation of the data of the application.

The data manager accesses to the application to know what to represent. It also communicates to the application the modifications applied to the structure by the user. The control manager interacts whith the application and the data manager as a whole, through

entry points. This scheme emphasizes the use of entry points for the communication between the different modules. We believe that this architecture is powerful enough for most interfaces. Moreover it has the advantages of conceptual simplicity and implementation efficiency.

The concept of user interface generator has already been experimented through events ordering specifications [Olsen84]. These specifications are low-level and static. In our case, the flexibility provided with the entry points gives a dynamic and high level mean for dialogue specifications. The protocol between the application and the interface manipulates high level abstractions independant on the syntax of the interactive dialogue with the user. The functionalities of the data manager form a complement to the control manager : the control manager sees the application through the standard representation of the data manager.



- Figure 4 -

4 - Conclusion

The design and implementation of high-level graphic interactive applications are an important challenge. Although some systems already exist, they are most of the time dedicated to specific applications. Trying to standardize the user interface has proved to be very hard, and leads to interfaces using a toolbox which is most of the time too low-level. Standardization must be considered in terms of abstractions of the dialogue, but not in terms of toolboxes. The architecture of the system we proposed is built this way, thus allowing for extensibility of the differents components.

A system for building graphic interfaces to be connected to existing applications has been presented. A preliminary version of this sytem is currentlly running for small-scale applications. The system is itself interactive and allows immediate execution of the interface being designed. This can be considered as the very first steps toward graphical programming [Reiss86].

References

[Barth86]
Paul S. Barth, "An Object Oriented Approach to Graphical Interfaces". ACM Transactions on Graphics, Vol.5 nº 2, pp.142-172, April 1986

[Beaudouin85a]
M. Beaudouin-Lafon, "UFO : un méta interface graphique pour la manipulation d'objets". Proc. Congrès AFCET Matériels et Logiciels pour la 5ème Génération Mars 1985

[Beaudouin85b]
M. Beaudouin-Lafon, "Vers des interfaces graphiques évolués : UFO, un méta-modèle d'interaction". Thèse de 3ème cycle Octobre 1985

[Goldberg83]
Adele Goldberg and David Robson, "SmallTalk 80 : The Language and Its Implementation", Addison-Wesley, 1983

[Hanusa83]
H. Hanusa, "Tools and techniques for the monitoring of interactive graphics dialogues" *Int. J. Man-Machine Studies* 19, pp. 163-180, 1983

[Hayes85]
Philip J. Hayes, Pedro A. Szekely and Richard A. Lerner, "Design Alternatives for User Interface Management Based on Experience with Cousin", pp. 169-175 in *Proc. of the CHI'85 Conference,* The Association for Computer Machinery Publ., April 1985.

[Karsenty86]
S. Karsenty, "Object Oriented Tools for the Design of High Level Interfaces : the Key for Adaptability", *Proc. IFIP WG8.4 Working Conference : Methods and Tools For Office Systems*, Oct. 1986

[Kasik82]
David J. Kasik, "A User Interface Management System, pp. 99-106 in *Computer Graphics*, July 1982.

[Olsen83]
Dan R. Olsen Jr. and Elizabeth P. Dempsey, "SYNGRAPH : A Graphical User Interface Generator", pp.43-50 in *Computer Graphics*, ACM, July 1983.

[Reiss86]
Steven P. Reiss, "An Object-Oriented Framework for Graphical Programming", SIGPLAN Notices Vol.21 nº 10, October 1986

[Scheifler86]
Robert W. Scheifler & Jim Gettys "The X Window System", ACM Transactions on Graphics #63, Special issue on User Interface Software, 1986

What They Don't Tell You About Window Systems

Michael D. O'Dell

Maxim Technologies, Inc.
Vienna, Virginia USA

1. Preamble

People communicate using languages, whether it be with other people or computers. The nature of these languages determines what can be expressed and how effectively it may be communicated. By providing a commonality, languages bind people together into groups, but they also tend to exclude those not well-versed in the particular tongue. Languages, then, become powerful mechanisms for both rewarding and discouraging behavior.

Languages can be both aural and visual. Visual languages can be linear and verbal, representing and recording meanings or sounds from an aural language. But visual languages can also be powerfully non-verbal and transcend simple linearizations: the visual languages of the artist painting, sculpting or taking photographs, the visual language of the typographer creating fonts, or the visual language of the graphics arts designer creating images that inform and communicate across verbal language barriers (e.g., international signs). These non-verbal languages are some of our most powerful ways to communicate.

The languages people traditionally have used for communicating with computers are linear, verbal languages, strongly rooted in imperative forms. This is probably the product of some deep-seated cultural desire to command rather than to interact – why else would we call them *command languages?* These languages come in dramatically different flavors of complexity: the rigorous complexity of programming languages like C, Algol68, Ada and PL/I, the apparently random complexity of systems like **troff**, the gratuitous complexity of "screen editors" like **Emacs**, the flexible complexity of the UNIX shell, the rococo complexity of OS/MVS JCL, and the elegant complexity of pure, unconditional linear sequences (e.g., byte-stream files).

It is bad enough for the humans involved that these languages are often quite complex. What is worse, however, is that they are all radically *different* from one another. We speakers-with-computers live in a world of linear chaos – as though the Tower of Babel had fallen over onto our keyboards. Some of us fare better than others. Some cope by being polyglots in the extreme, and others cope by learning a small number of languages and then insist upon using of one of their select, fluent tongues for whatever needs to be accomplished. This is both good and bad. If the right set of survival languages is picked, one need not be terribly cut-off from the general goings-on and can get on quite well in many different circumstances. If the wrong set is picked, however, as the saying goes, "If all you have is a hammer, everything looks like a nail!" This results in solving every conceivable problem with either a Version 8 Shell script or a LOTUS 1-2-3 macro, depending on one's religious upbringing.

What does this have to do with window systems? Nothing in particular, except that we are witnessing the re-creation of our one-dimensional, linear chaos on a new

two-dimensional canvas. What does this have to do with software using visual languages for communications? Everything, because creating rich, effective languages, especially visual ones, is *very, very hard.* It is the province of artists and designers, of thinkers-about-language, of those who can consider *all* the things one will *ever* want to express in a language. It is certainly not the province of programmers or computer scientists, as such.

2. An Object-Oriented Lesson

Imagine, if you will, that you have been given an assignment to write a brilliant short story about what you did over summer holiday. A great deal is expected of this story since the market for such stories is already heavily populated with other quite good ones. This writing assignment is really rather tricky.

Now imagine that you have taken complete leave of your senses and decided to start this project by first inventing an entirely new language in which to write it. This is a bad idea for more than a few reasons.

The sheer complexity of a language with the richness to say what one really wants to say is staggering. Having to invent syntax, semantics, and create a lexicon is an immense amount of work. It is hard enough to find just the right word in English, a language already rich with many words and subtle shades of meaning. If you are creating an entirely new language in the process, there is a good chance the word you need must be created and you will want to carefully adjust its meaning relative to all the other similar words. This quickly starts to approximate hard work! Even after all this effort, how many people will want to learn this new language so they can read your wonderful story?

Even if one takes a less radical approach, not creating the language from whole cloth but instead starting with a known language like English, and enriching it with lots of new words and novel new constructions, the problem remains that most people will have a hard time understanding what is being said. "They just don't speak the language." An example that comes to mind is Anthony Burgess's *A Clockwork Orange.* In this particular example, adding new language was artistically quite successful, but many readers found the work daunting, even though most of the writing was, in fact, still well within the confines of traditional English. The implication is that even carefully-controlled injections of innovation can sometimes cause considerable trauma when they appear at odds with an existing macrolinguistic framework.

Now imagine you are setting out to create a program which allows a user to perform some complex, variable task with great ease and aplomb. Believing that visual interfaces often make the user's task easier, you decide to give the program a visual user interface, running under whatever window system is currently all the rage. So, you get out the applicable three feet of documents and start reading about all the ways you can draw scroll bars and create menus. You almost certainly don't think of it as such, but you are about to create a language, a visual, two-dimensional language, rich in powerfully non-verbal concepts like selection and extension. Blissfully unaware you are creating a language, you haven't the slightest notion of the quagmire you're about to sink into. The worst part of this scenario is that you certainly aren't alone; everywhere there are many other programmers-cum-unwitting-designers about to wander off into the weeds, creating their own visual languages, completely oblivious to the language you are busily creating.

Finally, imagine for a moment, if you dare, just how much fun it will be to have five different windows active on your favorite workstation, with each one of the five programs communicating using a radically different visual language. "You mean that in this window, left button means 'keep' and in that window left button means 'kill'?!?!" Oh boy! I can hardly wait: pixelated pictures of the Tower of Babel!

3. The Current State of Confusion

Having "good window systems," whatever that might be, available on UNIX is neither necessary nor sufficient for having good visual software. The richness, coherence, and consistency of the visual languages used by such programs are the critical issues. While everyone can agree there is a clear requirement that the individual visual language used by any particular visual program be *internally* rich, coherent, and consistent, the issue of overwhelming importance is the requirement of *external* richness, coherence, and consistency *across* all the languages used by all visual programs in a given environment. If the global visual linguistic environment does not exhibit global richness, coherence, and consistency, two-dimensional chaos will unavoidably result.

The creation of a rich, powerful, consistent visual language for use by visual programs (in the UNIX environment or elsewhere) is an astonishingly difficult task. The number of persons in the world who can really do this job is probably about the same as the number of persons who can really design type fonts: at most, a two-digit number.

Expressed another way, one thing that separates true artists from persons merely chasing paint around a canvas is the successfulness of their visual language. It is the ability to create a language which has an identifiable clarity, style, and aesthetic. This is what makes a Picasso immediately distinguishable from a Monet, and either of those trivially distinguishable from an O'Dell. This is what makes good Macintosh programs instantly identifiable, whether in the company of bad Macintosh programs or Macintosh imitators.

Why am I concerned about this? At present, there is vanishingly little visual software running on any UNIX box. Currently, a "window environment" on UNIX means a large, complex collection of software that manages to provide a mediocre implementation of multiple ASCII terminals on the same piece of viewsurface. Most of the underlying window systems have the capability to support genuinely visual programs given sufficiently tenacious programmers, but to look at some popular window systems, it seems as if the last ten years of device-independent graphics development simply never happened.

Now consider what would happen if UNIX workstations with some "industry standard window system" were to suddenly catch-on in the general computer marketplace. Let's assume, against all reason, that software builders immediately scramble to build a pool of visual applications for this blossoming new market. What would be the result? Disaster.

There are already a few brave starts on real visual products out there: *FrameMaker* and *Interleaf* come to mind as more than credible forays into the current wilderness, but they only drive home the point: they already speak radically different visual languages, and this is only two programs. If you think people find "editor shock" disorienting, wait until they experience "mouse shock." A user faced with a collection of programs, each truly wonderful when used in isolation, but with the collection having no global, external linguistic consistency, will become horribly frustrated and will come to hate the system.

And that is, of course, the correct response.

4. What to do, what to do...

The author feels somewhat like the little boy who pointed out why the Emperor was feeling drafts all the time, but unlike the child, feels compelled to offer suggestions as to a good tailor. I fear this is rather difficult for several reasons, but will still hazard some suggestions for addressing the current and future problems, none the less. But first, a question.

5. Do We Even Have a Problem?

This is a somewhat odd question to raise after belaboring its description at length, but it bears close examination. If workstations running window systems and UNIX become successful in the larger computing marketplace, then either a miracle occurs and the software companies wake up one morning all designing to the same visual language standard (poof! no problem!), or we must solve the problem outlined above. There is, however, a much more serious way we might not have to solve the problem, and we should consider it before getting on with any proposed solutions.

6. No, We Don't Have Much of Anything

If the UNIX workstation market does not expand to include users who demand that their $30,000 UNIX workstation not be continually embarrassed by a $3000 Macintosh when it comes to having truly useful software for something other than the most raw programming task, then none of the issues raised in the first half of this paper matter one whit, for the simple reason that no software company in its right mind is going to spend money writing useful software for a market that isn't interested in it. This will reasonably be taken as proof that all our pontifications about the wonders of workstations and video interfaces are just so much self-aggrandizement, at least as far as the UNIX marketplace is concerned.

I happen to belong to the user community which has grown tired of the overwhelming irony of programming on one machine (a large, "powerful" UNIX workstation), but then doing everything else I do for a living on another. For example, besides writing C code, I do a lot of project management, viewgraph and briefing preparation, financial and resource modeling, and thought-organizing (using an outline processor) on a computer which costs one tenth as much, both in terms of hardware and software, as my "powerful" workstation machine. (One admission: I do still write final documents with **troff**, not because it is my favorite way to write, but because it is often the most powerful.) I anxiously await a single machine which can both walk and chew gum at the same time, and for which both shoes and chewing gum are available.

One really must wonder how long the existing workstation market can survive needing a continuing supply of customers who only want to run the FORTRAN and C CAD/CAM/CASE/VLSI programs and who *don't* need to do the other tasks associated with running a large technical project. I have heard it said that over 50% of the workstations in the world are being used to design new workstations. I hope this is wrong, because if it is right, it probably means there is no market out there for normal people.

In the worst case, I must agree with Steve Job's conclusions stated at the Summer 1987 USENIX meeting: if some serious software for the UNIX workstation market doesn't appear soon, software for tasks other than just programming, UNIX will go by the boards. This isn't a very bright prospect, so let's assume we have a problem. Now, how can we solve it?

7. YES!! We Have a Problem!

What is the problem? The UNIX community needs a visual language standard which can be implemented on a variety of window systems and is politically neutral enough that software builders (and maybe workstation builders?) can adopt it for their products without feeling that some competitor has an especial advantage for having promulgated the standard. Then the software builders must be convinced that adopting such a standard is actually in their own best interest, and that after adopting the standard, building software for the UNIX workstation market can be economically rewarding.

Most importantly, this new user interface visual language **must** be designed by an expert - a person trained and gifted in the design of visual languages for computer systems. This job certainly cannot be done by a committee, and as was shown above, it is not a job for hackers. How such a person might come to create this design and then what becomes of it will be discussed below.

The initial goal of the design effort would be the production of the equivalent of Apple's *Macintosh User Interface Guidelines*. This is a document which describes the lexicon, syntax, and semantics of the Macintosh interface in excrutiating detail and with absolute legal authority. The *Guidelines* creates a universe where great creativity can be exercised in building a user interface for a specific product within the limits of these constraints. But as with the Laws of Physics in the natural world, the rules of this universe (the behavior and placement of scrollbars, menus and their required contents, how dialogue boxes behave, etc.) are completely specified. One cannot change the speed of light on a whim, and programs which deviate from these guidelines are royally chastised in public and are generally considered to be "bad" programs.

An important adjunct to creating such a design and its supporting definition document is creating a library of code fragments which implement the "usual" way of handling parts of the user interface. The Apple document actually gives code fragments showing how the Macintosh User Interface Toolbox is used to produce the specified interface behavior.

This important task is rather more difficult for the UNIX community since our standard would be implemented on more than one window system. This will require significant, coordinated effort to induce *true believers* in each of the window system camps to implement the standard interface and then make the code available to their respective communities. This is a lot of work we would be asking people to essentially give away for the common good. This may or may not be a reasonable request.

8. But Where Do We Get the Brain?

One large obstacle remains: where do we get a quality user interface design? One solution is to somehow convince Apple that since they are about to introduce a UNIX machine, the Macintosh-II running A/UX, it would ultimately be in their best interest if

all UNIX window applications were to conform to their user interface specification, i.e., the Macintosh Guidelines. This approach has two serious problems: Apple probably wouldn't buy it since their system will definitely support the Macintosh Guidelines either as a UNIX library, or more directly by running Macintosh binaries within UNIX processes. Bringing the other workstations up to that level is probably not something Apple would find attractive. The other potential problem is that software builders might not be willing to adopt another company's design, particularly if it involved any kind of licensing arrangement. This clearly violates the political neutrality constraint stated above. Pity; Apple's interface is easily the best and most widely implemented visual user interface and anyone else would be *very* hard-pressed to do as good a job.

The other alternative is to somehow engage one of the practicing design professionals in the field to produce the design and documents and then make them readily available to the community. The only catch is that a minimal but credible effort would cost about $50,000, based on conversations with a real designer. This amount would produce a good point of departure, but more work would really be needed to even approach the quality and detail of the Macintosh specification documents.

Maybe this is a job that X/OPEN, or IEEE P.1003, or /usr/group, or even USENIX could fund initially, but it is rather hard to see how the first three could refrain from the terrible temptation to meddle with the result, and worse, meddle by committee. Other suggestions are certainly in order.

9. Conclusions

Window systems are not the answer to the problems of building good user interfaces for UNIX workstation products. Window systems may, in fact, obscure the problem by giving hackers a plethora of knobs to twiddle instead of doing the hard work of creating a powerful visual paradigm for the program being created. Building a good user interface implies creating a good visual language, a task not suited to programmers. If the number of visual languages is not carefully limited, the resulting chaos will astound even a UNIX user. The problem of how we get a quality visual language user interface standard created for the UNIX community is difficult, but we hope not intractable. The suggestions in this paper, however, are only that, suggestions. We need more minds seriously considering the problem and addressing its solution before it's too late.

MUSK - a Multi-User Sketch Program

Chris Crampton
Human Computer Interaction Section
Informatics Division
Rutherford Appleton Laboratory
Chilton, Didcot, Oxon
OX11 0QX, UK

cmc@uk.ac.rl.vd

ABSTRACT

In recent times we have witnessed an explosion in the use of personal workstations, but the development of interactive applications has lagged behind the capabilities of the hardware. This is particularly evident in the area of user-to-user communication.

The subject of this paper is a highly interactive sketch program that allows any number of users to participate in the sketching activity. In addition to sketching, facilities are provided for the inclusion of text and the pasting of images.

The paper starts with a brief retrospective of the user-to-user communication capabilities of UNIX† and progresses to describe the design and implementation of a multi-user sketch pad.

The paper concludes that such a facility is both feasible and useful although a number of questions remain about how multi-user and multi-media conversations should be managed.

1. INTRODUCTION

UNIX has always encouraged an open, informal, public environment which has helped it to acquire facilities for users to communicate by making utilities such as *mail(1)* and *write(1)* easy to implement. This paper is concerned with the synchronous, interactive style of communications programs characterised by the *write* program. *Write* is a very simple utility allowing two users to communicate over serial lines with lines typed at one terminal appearing on the other. No attempt is made to partition or tag the data to assist in identifying who said what.

As the UNIX programming environment has become richer the interactive communications facilities have also developed. The provision of interprocess communication (IPC) between separate machines and the ability to *select*[1] from several sources of input has made the implementation of networked communications utilities such as *talk(1)* possible. *Talk* allows two users, possibly logged onto separate machines, to communicate character based information. The concept of *talk* does not go beyond that of *write* except that the message data is partitioned to assist in identifying the originator. Also, the communication is unbuffered so that the data appears as typed. Other programmers have extended the *talk* concept to support conversations between more than two users [Waters87].

† UNIX is a registered trademark of AT&T in the USA and other countries.

[1] The *select(2)* system call provides programmers with a way of multiplexing synchronous I/O between several channels.

Today's UNIX environment is moving on from the idea that there are many users logged into a few mini-computer hosts via serial lines. More and more users now have dedicated workstations, each with its own CPU, high bandwidth communication channel and high resolution graphical display. However, the use of the high bandwidth communications capabilities has been limited mainly to the support of machine to machine data transfer to provide network file systems, network paging and swapping and file transfer. Little attempt has been made to exploit the bandwidth for highly interactive, user to user communication.

While this has been true for the UNIX community, numerous other groups have experimented in this area, notably the *Colab* work at XEROX PARC [Stefik87]. *Colab* is a computerised conference room where each participant has a networked station providing a kind of digital whiteboard. It is perceived as being useful in situations akin to meetings with a formality and structure being necessarily imposed.

This paper is specifically concerned with *musk* which is a highly interactive, multi-user sketching program which aims partly to fill the hole in the UNIX environment I have highlighted above.

Musk provides the following capabilities:

- Free hand sketching
- Circles, lines and boxes
- Text
- Copy and paste of text and bitmaps from anywhere on the screen
- A pointing mechanism
- A "talk" like sub-program
- Many users, connected by a Local Area Network (LAN)

The aim of the project is to experiment with a variety of user interface techniques to explore what users require and feel most comfortable with. Very little policy is imposed on the users' interactions with the system as we felt this to be the best starting point and seemed to be closest to "The UNIX philosophy!"

Although *musk* is currently only implemented for *SUN* workstations running under the *suntools* environment, it is expected that *musk* will be ported to other environments and machines with an interworking capability, *X* being the next likely candidate. As *musk* is implemented using a toolkit, porting should simply be a case of re-compiling once the toolkit is available in the new environment.

At the time of writing, *musk* functions in the homogeneous environment mentioned above with all of the features described. Users seem reasonably satisfied with the user interface and facilities.

2. FOUNDATIONS

My colleagues and I work in the Human Computer Interaction (HCI) section of the Informatics Division at the UK Science and Engineering Council's Rutherford Appleton Laboratory.

The work described here is one fruit of our extensive research and experimentation effort in the area of HCI. The primary goal is to support research into improving user interface design specifically within the realms of highly responsive applications running on single-user workstations. One task has been to provide a programming environment for ourselves and fellow researchers to pursue these research aims [Williams86].

The key component of this environment has been an ever-growing graphical toolkit, known as *WW* [Martin87]. This provides the sort of building blocks from which interactive applications can be assembled. This toolkit has a demonstrated portability across different versions of UNIX and different types of window managers. An almost identical programming interface is available on the *ICL PERQ* running *PNX* and the *Whitechapel MG-1* as well as the *SUN* running under *Suntools*.

Other work in the section has yielded a small library of routines to greatly simplify access to the Berkeley networking and IPC facilities.

Given these foundations and the readiness of their curators to make changes and additions as needed, I had the relatively straight-forward task of tying them together to produce a multi-user sketch program.

3. DESIGN AND GENESIS OF MUSK

3.1. Origins

Musk started out as a very simple, single-user, sketching program that only provided for free hand drawing. This was extended to support multiple users via a network connection An eraser and a simple text handling capability were also added.

At the same time, a colleague was working on a character based communication program with support for more than two users. This program, known as *Confer*, is similar to *talk(1)* in concept but uses a format akin to theatrical scripts to indicate who said what, see figure 1.

```
user1: [joins]
user1: user one's text
user2: [joins]
user2: user two's text
user2: more user two text
user2: [leaves]
user1: I am all alone
```

Figure 1. Output format of Confer

The experience we gained with these two tools indicated that a more versatile, integrated conferencing tool would be a worthwhile project. As a start, I added *confer* to *musk* as a sub-program to provide the forum for the sort of conversation one would prefer to conduct using voice, and indeed, a parallel telephone conversation can make this feature redundant. However, in addition to this tagged conversation feature, the *confer* sub-program allows users without a graphics capability to share to some extent in the conversation using the stand-alone *confer* program.

3.2. Extra Primitives

I noticed users tended to stick with fairly simple sketches that were built up from straight lines and curves. Therefore, I added lines, circles and boxes as extra primitives with local rubber banding providing feedback.

3.3. Copy and Paste

Although users continually asked for more sophisticated features such as splines, I felt it wrong to allow *musk* to keep growing to meet the never ending wish list of desirable features. Instead, a mechanism for including the output of more specialised programs was seen as the best way forward. This should allow the inclusion of spline curves, formatted multi-font text and anything else displayable within the *musk* workspace. Two mechanisms for copy and paste are provided:

(1) *Suntools* style "stuff" whereby a text selection is copied into the application as if typed by the user.

(2) Copy and paste of bitmap images using full screen access.

These features allow one, for example, to timestamp a session by simply pasting in a copy of a clock program's image.

In fact, the usefulness of this copy and paste goes beyond this example: many *musk* conversations are concerned with some aspect of work on a computer so it is very useful to be able to paste into the conversation the output of a program or even a code fragment or error message.

Text can be pasted into either the *confer* or the sketch sub-programs. The former also allows text to be selected for pasting into other applications which use the *Suntools* selection service. Text cannot be selected for copying from the sketch area as the picture is represented internally as an unstructured image.

3.4. Pointing

I noticed that a user would often wish to focus the other users' attention onto a specific area of the *musk* display. Using the sketching features to do this quickly led to a cluttered picture. Another problem was that if conducting a parallel telephone conversation users would often gesticulate with their cursors forgetting that the other users could not see their cursor[2] too.

What was needed was some form of shared cursor so I added a "pointer" device. An alternative would have been to echo everyone's cursor on each *musk* display all the time. However, this would have resulted in a cluttered display and been very distracting, although it would have helped participants to keep track of who was doing what. It is only occasionally that pointing is of use, so it is better that a more deliberate action is required. Possible confusion is limited by only allowing one pointer to be active at a time. This was also done to experiment with the idea of a *Woodstock*,[3] which had been considered for all interactions but was rejected on the grounds of being too restrictive. The idea is that a user can acquire the pointer which then ties a secondary cursor on all other participants' sessions to the local mouse. When the pointer is released it becomes available to other users again. For the duration of the pointing phase, the pointer icon is annotated with the user name of the current *pointing* party.

3.5. Concurrency and WYSIWIS

Throughout the development of *musk* a primary aim has been to achieve as near true *WYSIWIS* (What You See Is What I See) as possible. The *Colab* group [Stefik87] defined *WYSIWIS* as being where all participants "see exactly the same thing and where the others are pointing." They felt that their work had shown that *WYSIWIS* "creates the impression that members of a group are interacting with shared and tangible objects."

As *musk* imposes no ordering on users' interactions, maintaining *WYSIWIS* is complicated by all *musk* instances having to communicate concurrently so that remote changes are still reflected whilst local changes are being made.

3.6. Miscellaneous other frills

I made a number of other simple additions to *musk:*

- A "bell" which can be rung to attract the attention of users who appear to have dozed off or lost interest.
- Handling of iconise (and de-iconise) events. If a user joins a conversation while the local *musk* is iconised then it attempts to direct the user's attention to the fact.

3.7. Session management

This is undoubtedly one area that has been neglected with the consequence that session management is inadequate. Any user at any time may gate-crash a conversation and totally disrupt it. Our experience is that this is not a problem in a genial environment, but not everywhere is lucky enough to have such an environment.

---

[2] *Cursor* here refers to the graphical cursor usually tied to mouse movements rather than the text cursor.

[3] I once heard an unconfirmed anecdote about a multi-user database system being developed at *XEROX PARC*. This had no software locking to guard against corruption so the solution they adopted was that noone was allowed to update the database unless they held the *Woodstock*. This was a stick with some yellow feathers attached to the end.

Musk itself has no facilities for initiating a session and inviting other users to join in. However, it has been easy enough to implement a front-end which allows the initiator to specify parties, by user name, that he or she wishes to communicate with. The network can then be searched and a *musk* started up remotely, in iconised form. The remote parties can then join the session when it is convenient.

Undoubtedly this area needs consideration to devise a framework that is more secure and flexible.

3.8. User Interface

As has been mentioned, the group in which I work has undertaken extensive work investigating user interfaces onto interactive applications. Therefore, we already have ideas of what we think makes a "good" user interface. Other applications developed by the group have used a set of conventions to which *musk* must adhere to preserve consistency across applications.

Musk is entirely mouse driven with the keyboard only being necessary for entering message data. Assuming a three button mouse, the left button always initiates an action or makes a selection while the middle button always pops a menu. The function of the right button is not so clearly specified and is used by *musk* as an eraser in the sketch area and for extending a selection in the text area of *confer*.

The cursor picture is updated dynamically to provide additional feedback, for example a text bar is used when the cursor is in the text area. Probably not enough use is made of this as yet.

The display is divided into four regions, see figure 2. Along the top are four panels labelled "Sketch," "Lines," "Circles" and "Boxes." These are used to select the current sketch style which has a local effect, only. Down the right hand side there are series of icons for exiting, pointing, bitmap copying and bell ringing. Clicking on these invokes the corresponding function. The rest of the display is divided into an upper, larger region for the sketch pad and a lower region for the *confer* sub-program.

Window resizes are handled by redrawing the display. However, no scaling of the text and sketching is performed so larger drawings will be clipped in smaller *musk* windows.

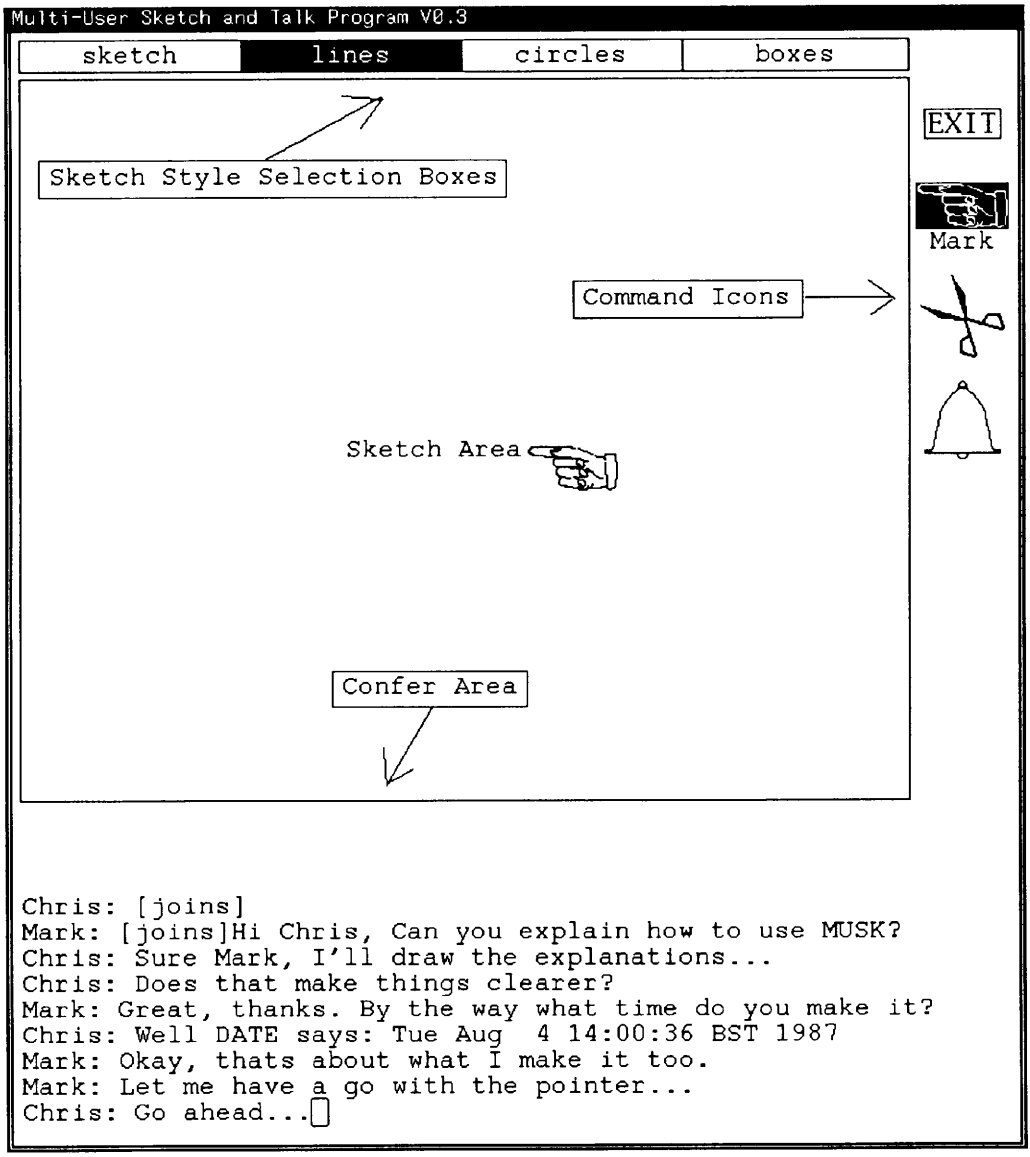


Figure 2. Display layout used by Musk.

4. IMPLEMENTATION

4.1. Basic Sketching

All graphics operations are performed using the OR raster-op function as this corresponds closest to what is expected of a sketch pad. Also, the final image is not dependent on the time ordering of operations which is relevant in the case of *musk* since time ordering is not preserved.

The *WW* graphics library provides routines for drawing circles, lines and boxes so these are used with an XOR raster-op function by *musk* to implement rubber banding feedback. Free hand sketching is ORed straight onto the display. All lines are one pixel wide and there is no provision for brushes or the filling of boxes and circles. A "clean sheet" function is provided on the sketch area menu. The eraser is implemented as a square brush which clears the area under the cursor as the mouse is moved.

4.2. Text

Text is drawn using a single fixed width font. It is important that all participating *musks* use a font of the same size to preserve *WYSIWIS* semantics. The *confer* sub-program uses a terminal emulator for output but the sketch sub-program paints the text onto the screen starting at the current cursor position. Consecutive character events are echoed across the screen, with handling for delete and carriage return.

4.3. Input

The input model of *WW* owes a lot to that of the earlier window systems on which it was built. The client has the locus of control and calls a routine everytime it wishes to wait for an input event. Lately, this model has been extended to allow the use of *select(2)* in *WW* applications. The main *WW* input wait routine in fact uses a function pointer to call the lower level input function and this pointer can be reset to call a client function instead. This allows a client to insert an extra layer of input handling which will be used by independently developed code such as a pop-up menu package.

The information about an event given by *WW* is rather limited. Apart from special cases, no indication is given as to what triggered the event, be it a button up, button down or mouse movement, although the state of the buttons and the position of the mouse are available. Special cases of input events include resize, iconisation, window enter and window exit.

4.4. Networking

A number of possible schemes were considered for networking *musk*, including the following:

Token Passing
: Each participating *musk* is linked in a ring of network connections with one read and one write channel per client. A token is passed around the ring and if a client receives an empty token it may place some data in the token which can only be removed by the placing client when it returns.

Broadcast
: Each participant has *n-1* read/write connections where *n* is the number of participants. All data is written to all parties by the originating *musk* and each client has to *select* from *n-1* sources for input.

Daemon
: Each client has one read/write connection to a central daemon. This daemon reads the data and echoes it to all other parties.

These alternatives each have merits and drawbacks. The first two would make it difficult for fresh parties to join a conversation after it has started and the ring may have problems with lost tokens and broken links. The daemon situation requires that an extra program be written and maintained and, like the token ring solution, needs one more IPC message per event when compared to the broadcast solution. The ring solution avoids concern over the time sequencing but propagation may take longer. Throughput could be improved by using more tokens but this would break the sequencing.

I decided to use the daemon solution as it was the simplest and had been demonstrated to work by the original *confer* program. This also meant that a daemon program was available which only needed customising for *musk*.

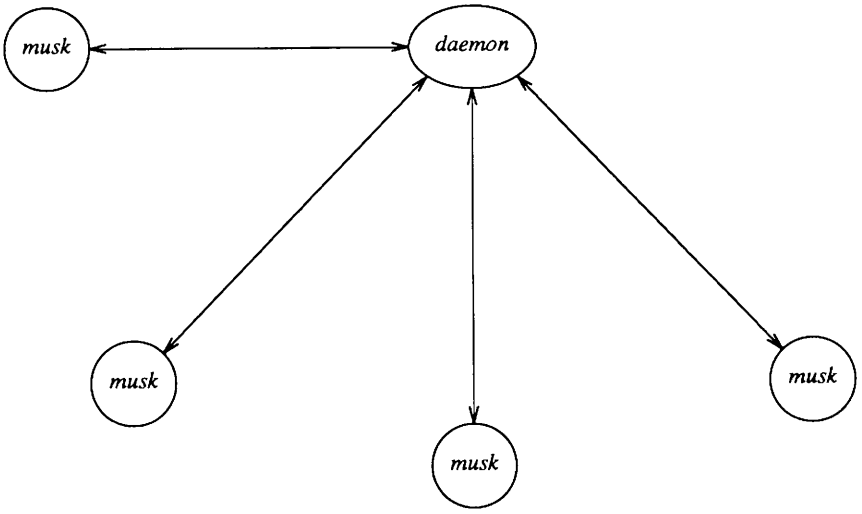


Figure 3. The network architecture used by Musk.

4.5. The Daemon

The daemon is implemented using TCP sockets although this is hidden beneath a library interface which provides the following calls:

```
int   ipc_server (port) int port;
int   ipc_accept_client (gateway, verify) int gateway, (*verify)();
int   ipc_client (hostname, port) char *hostname; int port;
```

The first routine is used by the daemon to open a file descriptor onto the specified port on the current host. The second routine is used by the daemon to accept connections from clients. The *verify* routine, if not NULL, is called with the hostname of the calling client to confirm the connection. *Ipc_client* is used by clients, *musk* in this case, to open a file descriptor onto the specified port on the specified host.

The original daemon needed no knowledge about the structure of the data it was handling. It simply selected for input on each channel and echoed any data read to all connections bar the originator.

The daemon can run on any machine, each *musk* simply has to know the machine and port number with which to connect. As the data is transferred in machine-independent form, the daemon can execute on a host-type different to those on which the *musks* are running.

As currently implemented one daemon is needed per conversation but this has not proved to be a problem as the usual scenario is that a user initiates a session causing a local daemon to be started which the other parties connect to. This system will break down should a user wish to initiate more than one session concurrently. The port and host are run time parameters so recovery could be made with a little work.

4.6. Extending Musk to be multi-user

There were two possibilities considered for extending the sketch program to multi-user operation:

(1) Use a high level protocol which would map onto remote procedure calls.

(2) Tag the low level input events with a source identification and propagate them.

I adopted the second approach for the following reasons:

- The code for handling the low level events was already implemented and only needed extending to be aware of the event source. This mainly required the use of vectors for state information.
- The networking could be mostly handled at the lowest level of the input code. The low level routine could *select* on the local input source and the network, with the local events being dispatched where they are read.

In fact, slightly more work was needed because of the problem of different display sizes. Events have to be tagged as intended for the sketch or *confer* sub-programs otherwise each *musk* could interpret the X and Y information differently. The event trigger (button up, mouse move etc.) is also reported.

Unfortunately, the functionality added to *musk* as it developed meant that higher level pseudo-events have to be generated for circles, boxes, bitmap paste etc. This has led to calls to the network code finding their way into the rest of the program.

Features of a user interface such as mouse feedback are time critical, so it is important that local interactions be as responsive as possible. The local response is improved, and the network loading reduced a little, by making local events short circuit the network and be handled directly. This also allows some censoring of the events broadcast as some only have local significance or no significance at all. For example, pop-up menu requests are not relevant to remote *musks*, but their effects are.

Sadly, this short circuiting has broken the *WYSIWIS* semantics: time ordering of events is lost. Despite the use of the OR raster-op function two operations are affected by ordering: erasing and "clean sheet." So far, this has hardly been noticed in normal use.

4.7. Copy and Paste of Text

Suntools, the only environment in which *musk* so far runs, has a *selection service*. This allows any text displayed by a cooperating client to be selected and copied. The *WW* library provides an interface to this service which *musk* uses. The *confer* sub-program makes displayed text be selectable for copying too.

Text which is pasted into *musk* is handled by both the sketch and *confer* sub-programs as if it is a string typed at the keyboard.

4.8. Copy and Paste of Bitmaps

The scheme here is that the user clicks and releases on the bitmap-paste icon (a pair of scissors) to invoke the function. *Musk* changes the cursor when it has acquired full screen access, using *WW*, and the user is then able to cut the desired bitmap. The source area is marked out using a rubber banded box and, after taking a copy of the bitmap, full screen access is released. The user is then able to drag the bitmap across the sketch area (using XOR) until it is placed at the desired destination and ORed onto the display. The bitmap has then to be copied to the other parties.

A problem with this is that all other communication is performed using discreet packets of a fixed size and a bitmap could not be made to fit into a packet. The initial solution was to use the *SUN's* network file system by copying the bitmap into a file and then sending the pathname to the other participants. This had problems due to the requirement of having a globally accessible and writeable directory - */tmp* is not globally available, at least with our file system topology.[4] Positional information about where to put the bitmap was encoded into the file name.

This implementation has now been replaced by one using the network connections used for the rest of the communication. A posting client sends a packet indicating that it is about to post a bitmap of a specified size. The bitmap is converted to a machine-independent string representation and written into the socket. The daemon reads the packet and buffer and then writes both to each of the other parties. Unfortunately, there is a complication: sockets have a limited buffer size so writes exceeding this size will block until the consumer at the other end reads data out. This can lead to deadlock. If the daemon was copying a bitmap to one client while the same client was trying to post a bitmap both would block. The solution used is that before a client can post a bitmap buffer it has to wait for a "go ahead" signal from the daemon. The client can then be sure that the daemon will consume as it produces. The daemon has the responsibility as arbiter for such requests.

Apart from the mechanics of actually transmitting the bitmap, there are problems with converting bitmaps to be of use to the other clients, even in an apparently homogeneous environment. Colour bitmaps do not (yet) copy sensibly. The problem here is working out how to collapse a colour image with, say, eight

[4] Directories used for temporary files cannot be shared between machines as programs depend on a unique process i.d. to generate unique file names. File system topology may still allow access to remote temporary file directories.

planes to a one plane monochrome image.

4.9. Pointing

The *pointing* state is entered by the user pressing on the pointer icon (a hand with index finger extended.) That user then has hold of the pointer until the button is released. Locally, the user sees the cursor change to look like the icon while everywhere else a second cursor appears and the pointer icon is annotated with the pointing party's user name.

SUNs have a ludicrously small cursor, only sixteen by sixteen pixels of one bit, which is not large enough for what I felt appropriate. Fortunately, previous experimentation by members of the group had shown that much larger cursors were perfectly feasible if implemented in software.[5] *WW* supports cursors of any size with an optional mask.

WW allows only for the single cursor tied to the mouse movements so the second cursor has to be implemented by *musk*. This was straightforward but needs care to ensure that the pointer is removed from the display before any graphics operations, and that the buffering is kept in sequence.

Only one pointer can be active at a time, so the clients have to negotiate with the daemon to acquire the pointer. If the pointer is already in use then the request is rejected. There is no queuing of requests. Once again, care is taken to make sure that clients keep in sequence with the daemon and parties joining while someone is already pointing are to be told of the state of the pointer lock.

4.10. Session Management

As was stated previously, this area has not been addressed in sufficient detail.

The daemon has to know how many connections there are and each musk needs to know the names of each user. The first packet that a new instance of *musk* sends is an initialisation packet giving the user's name. These are registered by the daemon and sent to the other parties. The daemon tells new parties about all existing connections. Parties leaving are de-registered.

The front-end for starting conversations is implemented as a pair of shell scripts. The *startmusk* script takes a list of users as its arguments and scans the network to locate which machines they are logged into. If the user is using the console then it is assumed that they have *suntools* running and an *rsh(1)* used to invoke the second script which executes a *musk* on the remote machine. Unfortunately this remote *musk* runs with the user id of the conversation initiator but an environment variable, **MUSKALIAS**, can be used to set the user name used by *musk*. An option for starting up iconised has been added to make the execution of a remote *musk* less intrusive.

[5] The bitmap dragging described above shows that large bitmaps can be moved smoothly if double buffering is used. It often amazes us to see applications suffering from terrible flicker which is quite unnecessary as double buffering is cheap and very effective.

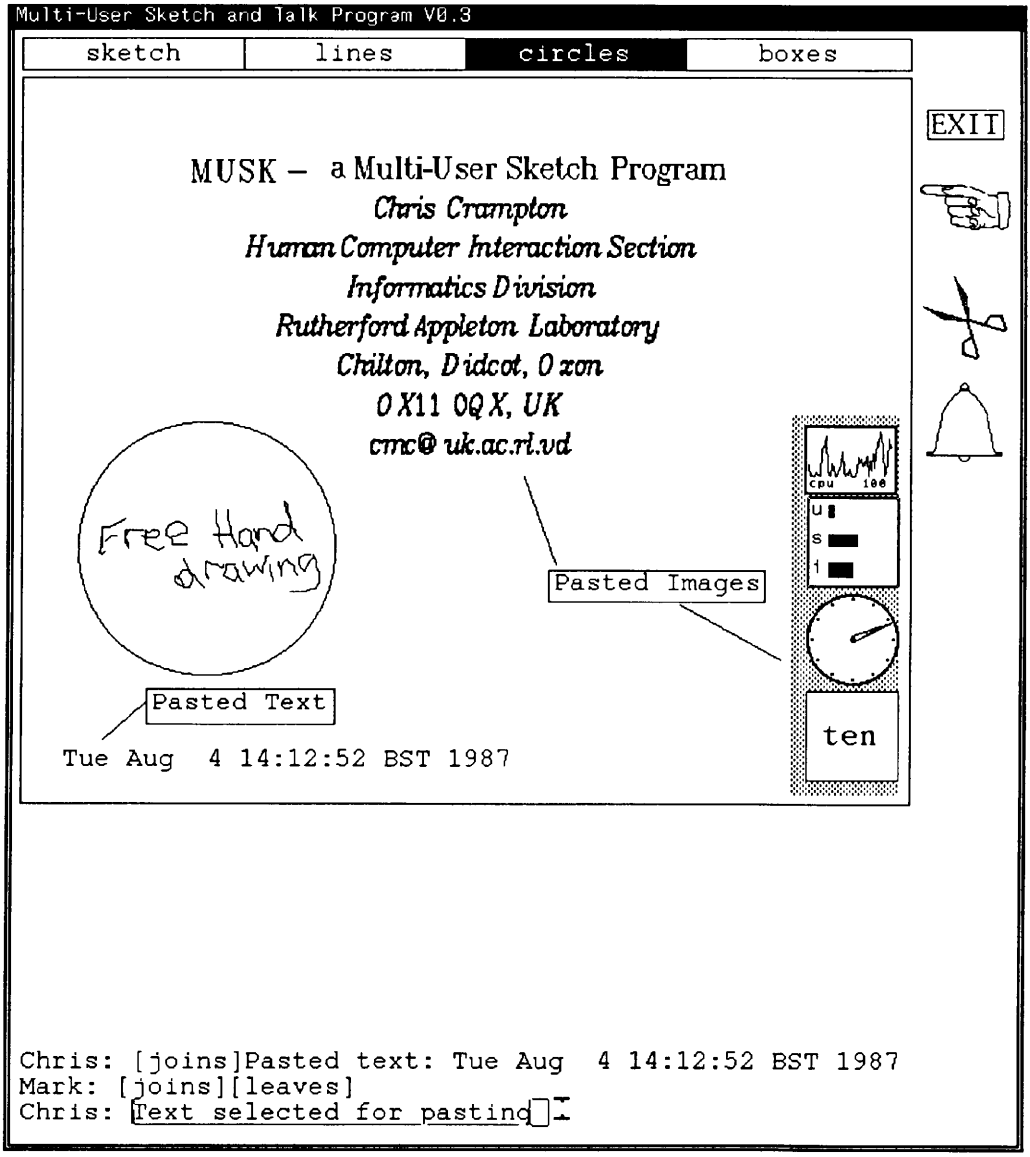


Figure 4. Illustration of Musk in use.

5. FOR THE FUTURE

In most respects, the *musk* program is as advanced as it is likely to get in the near future. What is now needed is a period of consolidation while *musk* is used so that comments can be collated to assess what changes and additions are required. However, some areas in need of attention are obvious and require experimentation to arrive at solutions.

Session management

Alternatives include using a chairperson with responsibility for session management or use of a more democratic voting system to make decisions, for example, whether to permit a new party to join or whether the picture can be wiped. Care will be necessary to avoid disrupting the flow of conversation with too much bureaucracy.

Colour

Currently, *musk* only uses two colours even when running on machines with a full colour capability. The copying of images on colour machines does not work. Apart from fixing this, *musk* could be made to exploit colour in useful ways:

- Extending the sketching capability to allow the use of coloured "inks"
- Use of different colours to distinguish each user's contribution

I have a number of other ideas that may be implemented if I find the time:

- Currently, when a user joins a conversation after it has started, he or she misses out on all that has been previously "said". It would be straight forward to extend the daemon to keep a history of the conversation since the last "clean sheet" so that the screen of a new user can be brought into line with those of the other parties. This could be a way of providing a playback feature too.
- There is no help or explain facility. Although the user interface is quite intuitive, some assistance for new users would be valuable. This could be simply in the form of a box which advises what the mouse buttons do at the current screen position.

6. CONCLUSION

If nothing else, *musk* has demonstrated the feasibility of highly interactive communication. Our initial reaction is that it has also demonstrated that the potential for such a tool is considerable.

After a period of exploration (usually less than a minute), the user interface is quickly understood by new users and the feedback has mostly been very positive. Comments seems to indicate that the user interface is successful by being intuitive, responsive and practical.

I don't pretend to have the answers for the deeper questions posed by the usage of such tools but we do have the capability to experiment and gain the experience which will allow us to ask these questions. Hopefully, a future paper will be able to give some answers.

A lot has been said about the "Desktop Metaphor" and this is symptomatic of the isolationist approach to computing so far taken, with a few exceptions. Now is the time to break out and think of more ambitious metaphors: the whiteboard, the beermat...

7. ACKNOWLEDGEMENTS

Many people have contributed to the design and implementation of *musk*. Foremost amongst this group is Mark Martin who is partly responsible for the *confer* sub-program and, more significantly, the *WW* graphics toolkit. He is the one who has wrestled with *SunView* to provide me with the facilities I needed. Crispin Goswell and Tony Williams have also contributed code and ideas. Indeed, everyone in the HCI group and many others at RAL have helped with development of *musk*, either with ideas or with tolerance at my repeated intrusions when I needed someone to "talk" to - debugging a multi-user, interactive program requires lots of cooperation or very long arms!

8. REFERENCES

Martin87 Mark Martin, "Foundations of a Toolkit", to be presented at a workshop on "High Level Tools for Window Managers" at EuroGraphics, August 1987.

Stefik87 Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning and Lucy Suchman, "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," Communications of the ACM, January 1987.

Waters87 Gill Waters and C W Tony Chan, "Three-party talk facility on a computer network," Computer Communications, June 1987.

Williams86 Antony Williams, "An Architecture for User Interface R&D," IEEE Computer Graphics and Applications, July 1986.

UBOAT
A Unix Based On-line Aid to Tutorials

Daniel V. Klein*
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh PA 15213
+1 412 268 7791
dvk@sei.cmu.edu

Abstract

Designers of computer aided education (CAE) systems are presented with the dilemma that course authors are required to also be expert in the field of computers. While this is a laudable virtue, it is very often the case that a skilled educator is not necessarily a skilled programmer, and conversely, a well trained programmer is rarely a good educator. Consequently, CAE packages are either rich in information but difficult to use, or else they are wonderful examples of human machine interfaces which convey very little real information to the student. This paper describes our work in developing UBOAT, a system designed to allow an educator with little computer expertise to develop an interactive, dynamic, and effective CAE system. UBOAT allows an educator to concentrate on his or her area of expertise, without requiring a great deal of distraction in coercing the computer to interface effectively with the learner.

Early attempts at generalized CAE generators such as *learn* or *PILOT*[1] provided an educator with simple interface to the student that allowed a predominantly top-down approach to presenting the educators materials. Little or no attention was paid to allowing for review, question and answer, ergonomic human-machine interface, or screen oriented display. *Learn* presented all of its text in serial form, and generally required modifying the source program to add features. *Pilot* allowed for simple graphics, but required that it be done in a terminal dependant way. Both these systems, as well as many others, had a cryptic, terse command syntax that was fairly unforgiving. The source files for the education scripts were often huge ASCII files, without any data compression or text reuse, and rarely, if ever was any debugging support or compilation facilities provided.

With the shortcomings of these systems in mind, we developed UBOAT, a system designed to allow non-computer experts to develop good CAE packages for their field of expertise. As a side effect, we also discovered that UBOAT could as easily be used as a menu system or a demo driver. The UBOAT system presents to the course author a easy to use source language that requires very little programming skills. Facilities for displaying text in a screen-independant fashion (also allowing for simple animation and partial screen updates) are provided, as well as a number of query-response interfaces, an integral "course debugger", pacing system, and an interface to the shell. Each mechanism has built-in defaults, so that if the course author wishes to query the student for a response of one letter from '**a**' through '**h**' the command in UBOAT is `query range "a-h"`. The query mechanism will handle prompting the student, filtering the response for errors (and printing appropriate error messages), timing out on non-response, and returning the response to the author. The course author, however, always has the liberty to override any or all of these defaults. The fact that the defaults are present, however, make the UBOAT system very easy to use.

*The work described in this paper was performed in the year prior to the author's employment at the SEI

[1] *Learn* was developed by Brian Kernighan and Michael Lesk of Bell Laboratories. *PILOT* is a copyright of Control Data Corporation

Course authors are presented with the model of an electronic book, combined with the display abilities of a teacher presenting information on-the-fly. To this end, the course author can segment information into "lessons", "units", and "pages", as well as including query-response, and automatic or directed pacing. To ease course development, an interactive "course debugger" is provided, that allow the course author to examine and affect the internal state of the course. The UBOAT system was designed to interface with most any terminal (through the use of *termcap*), rather than through a fancy windowing system. Our motivation for this choice of communication medium was driven by our experiences in teaching: the people who need training the most are the ones with the poorest computer facilities. Large screen, mouse driven windowing systems are usually held by the researchers, while the secretaries, clerical staff, and students generally have (or share) a simple terminal.

Although we designed UBOAT to present a simple, easy to use interface, we also recognized the need to allow for a great deal of flexibility in customizing a course, and allow the course author a great deal of leeway to 'tinker' with UBOAT's operation. To this end, we also provided access to the shell, although our implementation differs from *learn* in that our shell remains active, and thus can be used to save state information. We also have higher level programming features such as subroutines, flow control, multi-line textual variables, complex pattern matching facilities, an elaborate conditional execution system, and an override mechanism for every default in the UBOAT system. Thus, although a computer novice can develop an effective course using UBOAT, a person more experienced with computers can 'tweak' the system considerably.

The UBOAT system was written with portability (and wide distribution) as a paramount concern. Thus, although UBOAT was developed on a VAX running 4.2BSD, it also runs equally well under System V, System III, Version 7, Version 6, 4.1BSD, Mach, 4.3BSD, and all other derivatives we have been able to access. It has been successfully ported to the Sun, Masscomp, IBM RT-PC, Heurikon, PDP-11, MIPS M/500, Onyx, Zilog, and numerous other machines (our last 4 ports required 5 minutes of programmer effort). A Macintosh port is presently underway. The courses that we have developed are similarly portable. UBOAT has been used to create introductory Unix instruction, a user driven Ada demo, lessons in chemistry, a menu-driven shell, and an intermediate level presentation on linker/loader design.

1 Introduction

Before we can begin a serious discussion of the features of *UBOAT*, it is necessary to define a few terms. For the purposes of this paper we need to concern ourselves with three entities, namely:

1. The course author - this the person who deals with *UBOAT* as a "programmer", much in the same way that a programmer writes code in C. The intentions of the course author are expressed as the actions of the course.

2. The course - this is the "program" that the author develops, written in the "authoring language". In this paper, we will be treating the course as an active entity, since essentially it is an interactive system that responds to stimuli, providing stimuli of its own to the student of the program (or student).

3. The student - this is the person who deals with *UBOAT* as an end user, without concern for the language that was used to develop the program. As far as the student is concerned, a course written in *UBOAT* should be indistinguishable from a well implemented course written in C.

2 Text Variables and Graphics

One of the biggest problems with systems such as *learn* and *PILOT* is their lack of "program variables". Each of these systems have a simple mechanism for displaying text (the characters to be displayed are stored in place), with special commands being provided for extracting the information within responses. There is, in general, no concept of a "variable", so most of the internal mechanisms are fixed and unyielding. From a student's viewpoint, this is often of little concern, since a clever author can manipulate the authoring language to suit his needs. However, when we

designed *UBOAT*, we were also concerned with the welfare of the author, and thus took our cues from the Unix shell and other programming languages, to provide a more reasonable interface to the author.

All text is stored in variables, and variables can contain as many lines of text as can fit on a screen. This very simple generalization allows for a number of interesting features to be included in the *UBOAT* authoring system:

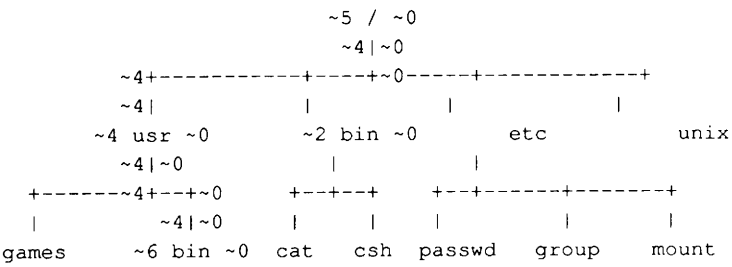
1. Text can be reused. There are numerous cases in a course where the same picture or set of words needs to be shown to the student a number of times on a number of different occasions. By storing text in variables, these pictures or sets of words can be referenced by name, instead of by content (simple text can be used once without needing a variable, if the author wishes). This also means that the amount of disk space that a course uses can be greatly reduced (an auxiliary program can also be used to compress courses to roughly 50useable).

2. Because variables can also contain strings of digits (i.e. numbers), these sequences of digits can be used by *UBOAT* for their numerical value. Thus, the placement of text on the screen can be at a variable location, instead of being fixed. In fact, anywhere a number could be used in the authoring language, a variable may be substituted.

3. Taking a cue from the C-Shell, variables can be used to alter the general behaviour of the authoring language, rather than resorting to specialized commands. For example, the variable **FLUSHMODE** selects whether student typeahead will be flushed, while the variable **INTERPMODE** selects whether the course is presented in "student" or "author" mode (the latter mode has the debugger enabled).

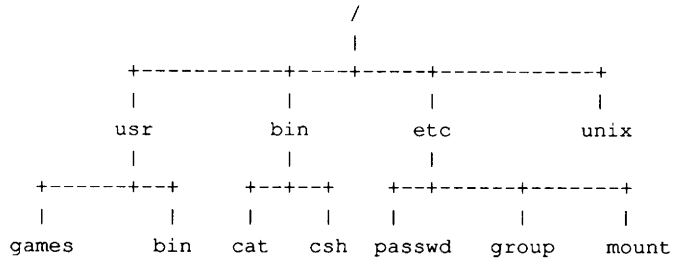
4. Variables can also be used to report to the author the status of the course. There are variables which tell the course the size of the screen, whether a timeout has occurred, what the status of the last shell command was, etc. Since variables are all multi-lined, multiple values can be stored in a single variable. The built-in variable **UBOAT** contains the version and revision information, with each component stored on a separate line. In this way, an author can select different behaviour for a course on System V Unix than on Berkeley 4.2 Unix.

The *UBOAT* system also takes advantage of a terminal's ability to display text in reverse video.[2] If the author desires that text be shown in standout mode, s/he brackets the text with a special standout mask. A boolean **and** is performed with this mask and a standout selector - if the result is **true**, the enclosed text is highlighted. Because text can be reused, it is very easy to use one variable to display the same information in slightly different ways. Consider the following example:



The bracketing pairs of "~n" and "~0" are the standout masks. Although they cause the variable to appear somewhat distorted, they are not shown on the screen when the variable is displayed to the student. When a standout selector of 0 is used, none of the boolean **and** tests result in **true**, so no text is highlighted, and the result would look like this:

[2]We have modified *termcap*, so that if the terminal suffers from the `sg` bug, characters can still be displayed in reverse video. (The `sg` bug's sympom is that the terminal requires more than one character position on the screen to change from normal to inverse video display). If the terminal does not have an inverse video feature, *UBOAT* uses special characters to enhance the desired text, so that in any event, the student sees highlighted text.



This, as you can see, is a standard representation of the Unix file tree. If we wanted to highlight the root of the tree, we would simply redisplay this same variable with a standout selector of 1, which would cause the '/' at the top of the tree to "light up" (of all the standout masks, only when 5 is **and**ed with 1 yields **true**). If a standout selector of 2 was used, then both of the **bin** directories of the tree would be highlighted (demonstrating that there can be multiple files in a directory hierarchy with the same name). When a standout selector of 4 is used, the path from the root to */usr/bin* would be highlighted (demonstrating how similar file names are distinguished by dissimilar pathnames).

3 Stepper Subsystem - Pacing the User

Generally, an author designs a course with the intention of allowing a student to pace through it at rate that is moderated by the student. One unfortunate aspect of *learn* is that it only allows a general top-down interpretation of the course material. Very often, the student will want to review a page, back up, or even cycle through an entire section again. In many authoring languages, this is a difficult task.

Many classical evaluations of CAE compare an unaccompanied CAE course to a textbook or a taped course, yet few CAE systems provide this basic analogy in the hierarchy of the system. We designed *UBOAT* around this analogy, and break down a *UBOAT* course into a set of components that parallel the breadown of a book. These components are the "lesson", which is broken down into "units", which is in turn divided into "pages". The advantage of pages displayed on a screen over pages in a book is that electronically displayed pages can be made to change dynamically, much in the same way that transparencies can be overlayed by a lecturer.

The authoring language automatically provides the student with a mechanism for stepping forwards and backwards across page boundaries. Should the author desire it, s/he can also implement mechanisms that step across unit or lesson boundaries. In this way, the student is not constrained to reading a course from start to finish, but can skip around much in the same way as a reader would skip from section to section in a book.

The *UBOAT* system provides much more than just an electronic book, although this in itself is a useful service. Since the authoring language is essentially a programming language (with variables and conditional expressions), we felt that there would be a need for higher level control structures - namely subroutines. Thus, should the author wish to create a subroutine that compares a set of prerequisites against a record of student experiences before entry to every lesson, this is an easy thing to do. In fact, although the *UBOAT* system is primarily an interpreted system, recursive subroutines are supported. Should the author wish to write a recursive evaluation of the Fibbonacci series, this would be possible (although it would not be recommended).

4 Query Subsystem - Getting Answers from the User

Responses from the student typically fall into a simple pair of orthogonal classes. The course is either interested in a multiple-choice answer, or some more elaborate string. *UBOAT* provides both of these classes to the author, utilizing a set of default prompting and error handling mechanisms that present to the student a consistent interface to which they can easily accomodate.

4.1 Multiple Choice Queries

The multiple choice class of response is so pervasive to all CAE systems that we felt it was a clear indication that *UBOAT* should have a built-in mechanism for processing them. Thus we broke this class of responses into three subclasses:

1. Range responses - this is where the course needs an answer in the form of a single character from (for example) '**a**' through '**f**', or '**1**' through '**5**', or '**G**' through '**K**'.

2. Subset responses - when the course requires that the student respond with a single character out of a set of (usually discontiguous) responses, such as '**a**' for append, '**i**' for insert, '**j**' for jump, '**q**' for quit, etc.

3. Yes or no responses - when all that is needed is a yes or no response. This is actually a simplification of the subset response.

In these three cases, *UBOAT* automatically prompts the student for the correct response by printing a message appropriate to the type of the query. It then filters the student's responses (performing the appropriate upper-to-lower or lower-to-upper case conversions), rejecting illegal entries. The course need only test the student's responses against the valid set of answers, since the query mechanism filters out all illegal answers.

As with most other features in *UBOAT*, the author can relegate all error checking and recovery to *UBOAT*, or catch them herself as desired. Thus, the author is free to concentrate only on the course materials and not on the mechanics of the presentation system. Alternatively, the author may also spend as much effort as desired on error recovery as she desires. For the most part, the default mechanisms suffice, but there are cases where it may be beneficial to enhance these. The course author has control over all of the following:

- Prompt message - the message that is printed to indicate that a response is desired. By default, this message reflects the set of characters that are expected.

- Case conversion - by default, *UBOAT* accepts both upper and lower case, and converts responses to match what the course expects. This may be disabled if desired.

- Error message - ordinarily, a simple error message is printed if an illegal response is given, although this message may be changed.

- Error handling - by default, *UBOAT* will cycle indefinitely until the student gives a valid response. Should the author wish to limit the number of errors, or have a subroutine process error conditions, this is easily accomplished. *UBOAT* also tests for the usual student mistakes, including typing the characters '<', 'r', 'e', 't', etc. when a legal response is *<return>* (i.e. the return key). This feature can also be disabled.

- Timeout - under normal operation, *UBOAT* will wait "forever" until the student responds to a query. An automatic timeout can be enabled on a per-query basis, or on a system-wide basis.

4.2 Unformatted Queries

Simple responses are not always needed from a student - often a single character (without any pre-checking) or a string of characters are needed. The *UBOAT* query system provides both of these features, with an optional limit on the number of characters that will be accepted from the student. As with the multiple choice queries, the author has control over the prompt and error messages that are printed, as well as the error handling and timeout characteristics of the query.

5 Conditionals - Checking User Responses

There is not much point in getting a response from the student unless it can be compared against what the course believes to be a legal input. In a *UBOAT* course, all student inputs are captured in variables, so that a number of high level manipulations can be performed of their contents. Because conditional expressions are implemented in a generalized form, comparisons can be made on pairs or variables, or on a variable and a constant (all that *learn* provides is a way of checking the last student response against a constant - this is very often inadequate).

The comparisons are provided include the usual set of relational operators, namely:

- Equality and inequality - these comparisons are done on a character by character basis, using the standard ASCII lexicographic comparisons.
- Less than, less than or equal, greater than, greater than or equal - these comparisons are also performed on a character basis. However, if both strings being compared are purely numeric, then the comparison is performed numerically instead of lexicographically.
- Pattern match - this relational operation uses the full set of regular expressions found in *grep* (although the function is built into *UBOAT*).

The standard relational operators are provided for checking responses. We included a pattern match facility was included when we observed that although the student was typing in a legal response to a question, the course was not recognizing it as legal. This was the case when the student did not type *exactly* what the course expected, and we felt that this was an inappropriate action for a modern CAE system.

It should be noted that the comparisons performed by *UBOAT* are still fairly simple. If the author wishes to include more complicated recognition facilities (such as the parsing facilities provided by *lex* and *yacc*, then these features can be incorporated as describe in the following section.

Once the student responses (or, as will be seen in the following section, the output of the shell) have been evaluated, *UBOAT* provides a standard repetoire of conditional expressions, including:

- Simple conditional - if the condition is true, then execute the following set of expressions. Naturally, an *else* clause is also provided.
- Case selection - select one of a set of alternatives based on the contents of variables. The actions may be based on the concatenation of multiple variables, so that highly complex, dynamically executable branching can be developed.
- Compound conditionals - simple conditionals and case selections may be nested to a reasonable depth, so that most any set of test conditions can be evaluated.

6 Interfaces with High Level Functions

There are many features that the course author may wish to have included in a course. Among these are arithmetic packages, data sorting, data filtering, date and time facilities, etc. Rather than implement each of these features directly inside of *UBOAT*, we instead decided not to re-invent the wheel, but rather to use previous incarnations of these tools. Most all of these features are available as user level commands from the shell, so what *UBOAT* possesses is an interface to an interactive shell. The choice of which shell to use was easy - we use the Bourne shell, primarily because every version of Unix that we have encountered has it, while not every system has the C-shell.

The casual reader may at first scoff at this idea, claiming that it is nothing new, and that many other programs (such as *ed*, *dc*, *make*, and *learn*) have a similar interface. The difference between these programs and *UBOAT* is that each of these other programs spawn a new shell for each shell command that is to be executed, while *UBOAT* creates a single shell, and communicates with it via a pair of pipes. The net result of this is manifold:

1. Because a *fork()/exec()* pair does not have to be executed for each shell command, the interface to the shell runs much faster.

2. Since the shell is not recreated for each command, the shell can be used to save state information (in shell variables or environment variables).

3. Since shell commands are not limited to one line, it is possible to execute complex shell commands (such as *case* statements). The mechanism employed by *UBOAT* is far more elegant than the hack provided by *make*.

4. Because the course is essentially connected to the standard input and standard output of the subshell, it is possible to take the contents of *UBOAT* variables (which can of course be multi-lined), feed them through a user level program, and capture the results in another *UBOAT* variable. In this way, the author automatically has full access to all of the Unix filters for arithmetic, sorting, keyword lookup, character translation, editing, etc.

5. As with interactive CAE systems such as *learn*, the *UBOAT* system allows the student's typed input to be executed directly, so that the output of the commands executed can be shown to the student. Unlike *learn*, however, *UBOAT* has the ability to capture the output of the shell comands and evaluate it before presenting it to the student. All that *learn* can do is examine the return status of the executed command, a feature used inconsistantly in Unix.

6. Many authoring languages provide a "native language" interface that allows authors to write in a subset of the host language (a restricted set of C, for example) to facilitate more complicated course actions. By providing a generic interface to *any* user level program, *UBOAT* allows an author to use *any* programming language when s/he feels that the command syntax of *UBOAT* is inadequate.

A simple very example of the power of the shell interface is shown in the following example. In this authoring language script, we demonstrate a binary counter. The shell is used for the simple arithmetic of adding 1 to a shell variable named **x**. This variable is captured in an *UBOAT* variable called **cnt**, which is used as a changing standout selector. As each of the standout masks in the string is **and**ed with the standout selector, the string will appear to "count" on the screen - each bit illuminating in proper sequence.

```
    # Initialize the shell variable x to zero.
! x=0
    # Provide a label for the goto at the end.
label loop
    # Capture the contents of the shell variable named 'x' into the
    # local variable named 'cnt'.
set cnt = `echo $x`
    # Display the contents of the variable 'cnt' followed by the
    # binary counter, centered on line 12 of the screen.  The standout
    # masks in the string are moderated by the contents of the local
    # variable 'cnt', which gives the appearance of a sequencing
    # binary counter as 'cnt' is incremented.
show "~(cnt)  ~8..~0  ~4..~0  ~2..~0  ~1..~0" 12 CEN_EACH ~(cnt)
    # Escape to the shell to increment the shell variable 'x'.
! x=`expr $x + 1`
    # Pop back up to the top of the loop.  No 'sleep' is needed, since
    # the execution time of 'expr' is a sufficient delay.
goto loop
```

Clearly, this is a trivial example, but consider that in other languages this simple script would take much more effort. More complicated examples are available, but these exceed the scope of this paper, and are presented in the *UBOAT* user's manual.

7 Commercial Considerations

UBOAT is an interpreted system. As such, it is naturally less efficient than a compiled system. However, in most cases, the speed of execution of the program is limited by the speed of the output device on which the student is learning. Since this is typically at a maximum speed of 9600 baud, the extra time required to interpret the *UBOAT* script is rarely noticed by the student.

There are some obvious advantages to an interpreted system, however. The author can make changes to the course and test these changes quickly, without waiting for recompilation. Because the authoring language is executed "as-is", it was very easy for us to include an integral debugger that reports on the state of the course and all of the internal variables, as well as providing a simple means of manipulating the source "on the fly".

However, an interpreted system also means that the course materials are stored in plain text form, and this has two important ramifications. The first is that the course can take up an awfully large amount of disk space - a problem at most sites. The second consequence is that the course materials are readable by anyone - even those who are not using *UBOAT* as a presentation medium. This means that theft of course materials are trivial, and this is a very real concern in the commercial environment. To counter both of these problems, a data compaction program was developed that not only reduces the size of the on-line course materials, but also renders them unreadable without the *UBOAT* system.

Another concern of commercial applications is that of unauthorized exit from the course. Since *UBOAT* can also be used as a driver program for a demonstration system, it would be potentially embarrassing (or damaging) if a user could escape from a demo and start perusing through other files on the system. *UBOAT* provides a number of different security modes (selectable through variables in the course) that can, if desired, completely disable unauthorized termination of a course or demo (including trapping of all signals), or ensure that correct shutdown procedures are followed before termination. In this way, a demonstration system written using *UBOAT* can be left running unattended, without fear of malicious user intervention.

8 Integral Debugger

When developing a course, usually the only way to test it out is for the author to play the part of student, and attempt to "break" the system. When this happens, the author typically installs instructions to test for the offending condition, and the problem is circumvented. However, when the unexpected occurs, and the course behaves in an unpredictable way, there is often very little that can be done in most authoring languages to rigorously seek out the cause.

UBOAT provides an integral debugger that allows a course author to examine the current status of the course, including the contents of all variables, location within the course, and subroutine nesting. The debugger also allows the author to single-step through, and to trace the execution of the course. It is also possible to enter the editor directly from the course to correct errors.

Since the authoring language is interpreted, no special considerations need to be made to enable the debugger - it is actually a part of the interpretive system itself. Once a course is deemed to be correct, simply setting a variable in the interpreter (via the course) disables the debugger, so that students do not inadvertenly activate it.

9 Conclusions

So far, all that this paper has discussed are the benefits to the author of a CAE script. Truly, *UBOAT* provides a highly flexible, dynamic, interactive medium for developing computer aided courses with a minimum of headache. In fact, owing to the flexibility of the system, *UBOAT* is not be limited to the development of tutorials - it has already been

used as a driver program for a hands on Ada compiler demonstration and as a user friendly menu driven shell. From the author's standpoint, *UBOAT* is a simple to use system that has default actions for most all conditions, and enables a course author to get information onto the student's screen with a minimum of effort.

The student also reaps from the benefits of the *UBOAT* system, by being presented with a consistent user interface. Additionally, since the author of the course does not need to be an expert on computers, the author can instead concentrate on their area of expertise. The net result of this is that the student gets a better course, since the author has had to spend less time jumping through hoops, and more time concentrating on the course materials.

Another benefit is one of portability. The *UBOAT* system was written with portability (and of course, wide distribution) in mind, so the source code for *UBOAT* can be ported to new machines with a minimum of effort. Because of this, both the presentation system (i.e. *UBOAT*) and the courses themselves can be readily distributed to a wide variety of systems, giving a greater amount of exposure than would be generated by other, less portable systems.

The final benefit, though, is derived from the fact that *UBOAT* was designed to run in a terminal independant fashion. There are a number of CAE systems on the market that run on windowed graphics systems, or that utilize special cassette tape interfaces, video disks, or "learning lab" facilities. While these are often excellent educational tools, they are often fairly expensive, and more importantly, present a bottleneck in that only 1 student can use them at a time.

The students who most need training are very often those who have the least money budgeted to training (i.e. secretaries, entry level programmers, etc.), and who have the least amount of free time to dedicate to the intensive training that these systems provide. The special systems also require special hardware, and often special tools to modify the existing courses, or develop new ones. This often locks the students (or the student's organization) into one type of hardware, or course provider - a bad idead no matter who is doing the providing. So while the sophisticated systems may be better on a per-student basis, they deliver much less "bang for the buck" when considered in the long term.

UBOAT therefore provides an inexpensive, easy to modify, easy to use system, that is easily affordable in terms of both monetary and time constraints, and can be used by the average student at their own terminal.

Developing Ada™ Software Using VDM in an Object-Oriented Framework

Chris Chedgey
Seamus Kearney
Hans-Jürgen Kugler
Generics (Software) Limited
7, Leopardstown Office Park
Foxrock
Dublin 18
Ireland

Abstract

A software development method encompassing a variant of Object-Oriented Design (OOD) and the Vienna Development Method (VDM) is described. The development is targeted at the Ada programming language. Part of the objective of the work described here is to produce a method for constructing Ada software - a method which follows sound engineering principles in order to improve software quality as well as improving productivity on the part of the developers.

1. Introduction

This paper describes the use of the Vienna Development Method (VDM) [Bjorner 78] in the context of the development of software to be targeted at the Ada programming language [Ada 83]. Part of the objective of the work described here is to produce a method for constructing Ada software. In this context a *method* is an orderly set of techniques and guidelines, designed for repeated use in developing products of similar properties. The authors are involved in examining the suitability of VDM in this context. ESPRIT project 510 *ToolUse* [Horgen 86] is examining the relationship of VDM to the typical software life cycle. The use of VDM to derive Ada packages as directly as possible is addressed within ESPRIT project 496 Papillon [Chedgey 86].

The paper discusses some of the observations made about VDM highlighting the advantages to be gained and the inherent problems that were encountered when applying it. A solution to the specified shortcomings is proposed incorporating VDM within a generalised development method.

VDM is combined with a variety of other techniques, derived from Object-Oriented Design (OOD) [Booch 83] [Booch 86] and Jackson System Design (JSD) [Jackson 83] [Cameron 83] [Cameron 86], to improve the support during the system specification and design phases of the software lifecycle [SLC 80]. Subsequently the refinement of data types with specific reference to their implementation in Ada is discussed.

™ Ada is a registered trademark of the U.S. Government, AJPO.

The experience of the authors stems from applying OOD, JSD and VDM to a number of case studies [Ryan 86]. They are applicable to a software engineering user of those methods, rather than a theoretician.

2. Developing Ada Software

The choice of which programming language to use within a development always merits careful examination. The following requirements are normally high in priority when assessing languages and generally apply to most application domains. A programming language should contribute to the effective management of the project whether it is small and simple or large and complex. It should aim to increase programmer productivity, reduce errors, improve performance and facilitate maintenance [Myers 86].

Ada is one programming language that supports the above features. Other languages such as Fortran, Cobol, C and even Pascal fall short of these demands. Most existing languages tend to emphasize syntax and features that are used to describe relatively small program units such as statements and procedures [LeBlanc 82]. When applied in larger projects these languages tend to complicate problems rather than resolve them.

Ada is a programming language suitable for building large, yet reliable systems. Ada was originally targeted at embedded systems but has, more recently, been applied to a wider range of commercial applications specifically those systems where complexity and size are major concerns.

Ada was designed to enhance program reliability and maintenance as these are the largest cost areas in the software lifecycle [Ada 83]. Programming as a human activity was considered by placing an emphasis upon helping the programmer to manage the complexity of software solutions. The language was required to support efficiency. Any language construct whose implementation was unclear or that would require excessive machine resources was rejected.

These benefits cannot be realised by simply applying the programming language. The provision of some form of design support is required to guide the structuring of systems in a disciplined manner [Mickel 84].

The proper use of languages like Ada, C++ and Smalltalk require a different approach to design than the approach one typically takes with more functionally oriented languages. These latter languages are best suited to functional decomposition methods, which concentrate upon the algorithmic abstractions.

Functional decomposition methods are not sufficient for developing Ada programs. In general, such methods do not explicitly consider information hiding, and data abstractions are difficult to enforce. Functional methods fall short in taking advantage of Ada's expressive power. Ada embodies many concepts that are not provided by traditional languages (e.g. packages, tasking and generics) and, as such, these approaches could not fully exploit the power of the language. There are also a selection of problems that require knowledge to be represented as part of the system as opposed to the use of purely imperative processes. Such a trend implies a move to object-oriented problem solutions (which can be expressed with Ada packages).

What we require then, is a method that fully exploits the characteristics of Ada as a programming language. This should preferably incorporate an object-oriented approach and support data abstraction and information hiding. The constituent techniques should be based upon sound engineering principles so as to improve the quality of the software as well as improving the productivity of developers.

In general we can view a development method as comprising three major components:

Notation - the notation in which the design is expressed

Guidelines - guidelines for producing the design

Analyses - rules for checking the internal consistency and completeness of the design steps

A number of desirable characteristics of a method can be identified. The techniques, incorporated within the method, should cover a wide spectrum of the software life cycle. The approach should be applicable to small and large scale developments and provide some degree of formality i.e. formal guidelines on applying the method should exist. The overall method should not be purely top down in nature but make provision for iteration and change propagation throughout the design process.

As it stands, there is no one method suitable for constructing Ada software that supports all of the above criteria. The possibility of combining a number of techniques that are complementary in nature was considered. The constituent notations should be compatible and the various design representations output from the methods should be understandable on their own. There should be no gaps, clashes, overlaps, or difficult transformations needed to carry the results of one method forward to the next. It should also be possible to verify that the design documents produced during a development are consistent with each other.

2.1 Vienna Development Method

The Vienna Development Method (VDM) is a well established formal development method for the specification and design of software systems [Bjorner 78] [Jones 86]. VDM promotes a rigorous approach to software development based on the gradual transformation of abstract specifications into concrete specifications. At the uppermost level, the specification is given as an abstract model. More concrete descriptions are then derived by transforming abstract entities into ones reflecting more detailed aspects of the system to be implemented. This process is repeated until the specification can be transcribed into code.

VDM is interesting since it provides comprehensive support during the design stages of the lifecycle [Chedgey 87a]. The method is based upon an underlying formal specification language, called Meta-IV [Bjorner 80]. Meta-IV is a notation for describing models and provides a number of basic data types and type constructors out of which new data types may be modelled.

This approach is complementary to the formulation of packages implementing abstract objects and abstract data types in Ada [Jackson 85].

The meta-language combined with the reification technique illustrated in detail in [Bjorner 82a], [Cohen 86] and [Jones 86] directly supports stepwise refinement of specification documents by operation transformations (including functional decomposition) and object transformations (known as data refinement or reification). Proof obligations [Jones 85] can then be generated to demonstrate the correctness of transformations and to show that invariants or other properties are still maintained. Guidelines exist on how to apply the notation and reify the various design documents produced during a development.

There are a number of drawbacks to using VDM as it is currently defined. VDM does not support concurrency. In an effort to overcome this the possibility of combining VDM with other formal methods that model concurrency issues is being examined [Hoare 85] [Milner 80].

There are problems with the notation of the meta-language when considering languages supporting data encapsulation, such as Ada. The basic concept in Ada is the package, encapsulating data and operational abstraction. The "traditional" version of VDM does not allow for grouping of domain definitions and operations into modules, nor does it support expression of information hiding other than through structuring of the specification document and careful choice of reification levels. Realisation of this drawback led to some modification in the style of writing VDM in our design method and will be discussed later.

2.2 Object-Oriented Design

The Vienna Development Method provides strong guidance during the design stage but does not maintain this support during the requirements and system specification stages where the developer is fully responsible for formulating the problem description [Kearney 86]. A full description of the functionality of a system must be at hand before VDM can be properly applied. This prompted a review of other design techniques to identify a method that would combine well with VDM and provide support for the earlier stages of the life cycle.

Object-Oriented Design (OOD) was examined within this context. Due to its object orientation, OOD appeared to tackle data oriented problems well and was a potential candidate for use alongside VDM since it supports the tasks of problem formulation and description. Moreover, OOD was known to incorporate mechanisms that support the specification of packages and tasks, something rarely addressed in other methods.

OOD is a design approach based on the principles of abstraction, information hiding and abstract data types [Guttag 77]. The object-based approach characterises a system as a collection of abstract data types (ADT's) or abstract objects. An object, in this sense, is an entity whose behaviour is characterised by the actions that it suffers and that it requires of other objects [Sommerville 85].

Objects themselves are characterised by associated data structures and a fixed set of sub-programs which are the only operations defined on the

object. Emphasis is placed upon identifying essential properties while omitting superfluous information. Subprogram bodies, dependencies and implementation details are hidden.

OOD has been used successfully in a large number of projects, e.g. a Satellite Tracking System developed at General Electric and a large (30,000 lines of code) application developed at Ford Aerospace.

2.3 Mechanics of OOD

The design strategy in OOD is globally top-down and recursive and starts when a requirements analysis document is available. OOD does not incorporate any requirements acquisition techniques. Therefore we must assume that there has been some prior analysis and that the software engineer has a basic understanding of the problem. Existing approaches incorporating data flow diagrams or SADT diagrams could support this process. An alternative approach would be to modify the entity-action and system modelling steps in JSD and incorporate them as a front end for OOD. Jackson System development would be particularly suitable since it is intended primarily for real-time systems.

Using VDM notation, the major steps can then be summarised as follows:

OOD :: Problem-Definition
(Informal-Strategy Formalise-Strategy)

Where the leaves of the tree are elaborated in the order above.

Problem-Definition = (Problem-Statement Analysis)

The design begins by describing the problem to be solved. The designer is responsible for acquiring any available information and for constructing a concise English language description of the problem. Emphasis is placed upon helping the designer to structure the requirements analysis information towards an object view. The support for this step is very weak in OOD. The authors are currently experimenting with parts of JSD and SARS [Koch 83] to improve the guidelines which can be offered to the designer.

The **informal strategy** step of OOD identifies the relevant entities and actions, thus forming the basis of the model for the system specification. In traditional OOD the informal strategy relies entirely on a natural language description of the problem, with an emphasis on highlighting

- the objects which are subjected to actions or which initiate actions
- the operations which can be performed on objects, and
- constraints which apply to objects and operations

There are detailed guidelines for writing such text. The relationship to the entity/action and entity structure steps of JSD is obvious and can be exploited [Connolly 86].

Much of the analysis about the objects and their relevant operations is left to the formalisation step:

Formalise-Strategy ::

(1) Identify Objects and their Attributes
(2) Identify Operations and their Attributes
(3) Establish the visibility of Objects
(4) Define Interfaces
(5) Implement Objects

The first step, identify the objects and their attributes, involves the recognition of the major actors, agents and servers from the project description and the identification of their role in our model of reality.

All operations suffered by or required by each object are identified next. This step serves to characterise the behaviour of each object or class of objects. The static semantics of the objects is determined by the operations that may be meaningfully performed on the objects or by the objects.

The attributes determine the dynamic behaviour of each object by identifying the constraints upon time and space that must be observed. The information may detail sequencing and concurrency considerations which may later influence the decision between subprogram and tasking implementations. An operation's attributes will also influence the definition of the interfaces of the operations.

In the third step the visibility of each object is established in relation to the other objects. This involves identifying the static dependencies among objects and classes of objects (i.e. what objects see and are seen by a given object).

A module specification for each object is produced, which defines the interface for each object and its associated operations using some suitable notation. The interface is defined with respect to the clients of an object and the object itself.

The final step of implementing the module definition for each object involves choosing a suitable representation for each object or class of objects and implementing the interfaces from the previous step. This may involve decomposition or composition. If an object is found to consist of several subordinate objects the design steps are repeated to further decompose the object. In other situations an object will be implemented by composition (i.e. the object is implemented by building on top of existing lower objects or classes of objects).

2.4 Combination of OOD and VDM

Many of the detailed guidelines given for OOD are directly translatable to guidelines for building specifications using VDM's meta-language. With some minor modifications the recommended design steps can be used as a front end for VDM. Using this approach, objects in a system are identified and

their structure and the semantics of their operations are modelled in Meta-IV. From this stage the method of refinement and the addition of more and more detail in successive specifications is the same as the traditional VDM approach.

Formalising the Strategy progresses from the informal problem and strategy statements to identifying domains of relevant objects. In VDM terms, this corresponds to moving from the syntactic domains in the initial iterations to the semantic domains. The semantic domains are then used to define the behaviour of the syntactic ones.

The emphasis on producing the objects of interest first before specifying the operations is very close to the usual ("traditional" or "Bjorner") style of VDM, i.e. specify the domains of interest and then the operations. As a model based technique, however, VDM describes many operations implicitly by specifying an abstract or representation model for the domain of interest. If the previous steps of object identification used abstract models, then it is important to consider the implicit operations here as well. OOD attempts to make all design steps and decisions explicit. The two steps of identifying objects and operations are clearly interrelated and the use of abstract models and implicitly defined operations has a major impact on the grouping of objects and operations, possibly causing iteration over a number of sub-steps.

The attribute descriptions will eventually contribute to the full definition of the interfaces of the operations. Therefore, this step will provide the material necessary for constructing argument and result domains of the operations. They will help ascertain whether the operation is a complete or partial function, or a state changing operation, what the pre- and post-conditions are, and what conditions are to be fulfilled by the implementation of the operation (proof obligations).

The VDM meta-language does not offer (at least not in the "Bjorner" style) any linguistic means to express associations between objects. The meta-language has therefore been extended to support Ada, as the target language, by introducing a piece of notation [Chedgey 86] which is a generalisation of the type definition of Affirm [AFFIRM 81] and it's VDM style described in [Beech 86].

When producing the module specifications for each object the representations and the associated operations are defined using Meta-IV.

Implementing the model is the recursive step of reapplying OOD, just one level further down. This corresponds closely to the combined data refinement and operation modelling of VDM, leading to a new specification using the reified domains [Bjorner 82a] [Bjorner 82b] [Jones 86]. The module definitions for each object can then be implemented by specifying a suitable representation for each object or class of objects using the Meta-IV domain types and specifying the bodies of the interfaces from the previous step. These representations are then refined through a number of stages until the design is detailed enough to be coded.

3. Case Study

In order to clarify the approach described above, part of a simple case study will now be described. The example application which is used is that of project planning and scheduling.

A system is to be generated which will enable a project manager to enter the various activities in a project, as well as the dependencies between them. The system will schedule the project in order to calculate the start date of each activity.

As described in previous sections, the Object-Oriented Design method is first applied in order to identify the various application's objects and object types, and their associated attributes and operations. These objects and types are then formally specified using a standard template and the VDM meta-language.

The result of this method is now illustrated by presenting some of the object and type specifications associated with the above application.

3.1 Specification of Abstract Data Types

Figure 1 shows a stripped down specification of the type "Date". The first two lines of the specification indicate that an Abstract Data Type called "Date" is being specified. The **Needs Types** field lists any ADT's on which Date depends. The **Interfaces** field gives the signatures of the functions defined for the Date type (note that it is only through these functions that objects of type Date may be manipulated). These first three fields of the specification then define the "signature" of the type Date, but give no indication as to the semantic behaviour.

The **Model** and **Functions** fields contain a formal Meta-IV specification of the ADT. In the **Model** field the Date data structure is modelled as a Meta-IV type. In the **Functions** field the semantic behaviour of the Date functions is specified in terms of operations on the data structure. Note that there may be functions specified in this section which do not appear in the **Interfaces** section and as such are not visible outside of the specification (for example "is_valid_date").

A Specification of the Activity type is shown in Figure 2. No **Interfaces** or **Functions** sections have been included in this specification. This indicates that the operations available for the Activity type are those defined on the Meta-IV type in the **Model** section (i.e. **mk**-Activity, **s**-id, **s**-duration, etc). This kind of specification is used when no extra semantic layer is to be defined for an ADT.

A Project plan type is specified in Figure 3 as mapping from activities onto the set of dependent activity identifiers.

```
Type
     Date
Needs Types
     INTG
Interfaces
     Make_date:    INTG  INTG  INTG -> Date
     Sel_day:      Date -> INTG
     Add_days:     Date INTG -> Date
     "<":          Date Date -> BOOL
     ...
Model
     Date ::      s-day   : INTG
                  s-month : INTG
                  s-year  : INTG

Functions

1.0  Make_date (d,m,y) =
 .1    (is_valid_date (d,m,y) -> mk-date (d,m,y),
 .2     T                     -> Undefined)
     type: INTG INTG INTG -> Date

2.0  Sel_day (d) =
 .1    s-day (d)
     type: date -> INTG

...

end Date
```

FIGURE 1

```
Type
   Activity
Needs Types
   QUOT, INTG, Date
Model
   Activity ::       s-id :          QUOT
                     s-duration :    INTG
                     s-description : QUOT
                     s-start :       Date
                     s-end :         Date
End  Activity
```

FIGURE 2

```
Type
   Project
Needs Types
    QUOT, Activity, Date
Interfaces
    Empty_project :                          -> Project
    Add_activity :    Project  Activity -> Project
    Add_dependency :  Project  QUOT  QUOT -> Project
    Schedule :        Project  Date -> Project
    ...
Model
    Project = Activity m-> QUOT-set

Functions

1.0  Empty_project = []
     type: -> Project

2.0  Add_activity (p,a) =
 .1    (a ε dom p -> Undefined
 .2     T          -> p U [a m-> {}])
     type: Project Activity -> Project

...

End Project
```

FIGURE 3

3.2 Specification of Abstract Objects

By slightly modifying the specification template used for ADT's abstract objects can be introduced. For example, Figure 4. specifies a single project instead of a project type.

Here the specification starting with **Object** indicates that an abstract object is being specified. In the **Interfaces** part, most of the functions have side-effects (indicated by the "=>"), that is they have become "procedures". In the **Model** part a state variable is declared and this variable is modified by the various operations. Note that the state variable can only be modified by an invocation of one of the operations defined in the Interfaces, unlike the case in unstructured Meta-IV.

```
Object
    Current_project
Needs types
    QUOT, Activity, Date
Interfaces
    Empty_project  =>
    Add_activity   Activity  =>
    ...
Model
    dcl p type: Activity m-> QUOT-set := [];

Functions

    1.0  Empty_project = c p := [];
         Type:  =>

    ...

End  Current_project
```

FIGURE 4

3.3 Specification of Related Subprograms

Occasionally subprograms need to be specified, which operate on one or more Abstract Data Types or Abstract Objects which have already been specified. Such a general collection of subprograms is called a **Package** and the specification starts with this heading.

These specifications do not generally export any types, but are used to extend the interface to already specified types or objects or to include application-specific operations.

4. Implementation of Specifications

When a first description of the system objects has been formalised in the form of Meta-IV specifications, the VDM reification process may be applied to transform these into concrete, implementable specifications. The form of the specification templates closely reflects that of Ada packages. Therefore, if we provide a mechanism for implementing the basic Meta-IV types [Jackson 85] (Herefordt 87] [Mac an Airchinnigh 87], the task of generating an Ada implementation is rendered simple and may even be automated.

4.1 Implementation of Basic Meta-IV Types in Ada

The simple Meta-IV types, i.e. INTG, QUOT, BOOL and TOKEN (also include is REAL), may be implemented directly as Ada packages. For example:

```
package INTG_model is
    type INTG is limited private;
    function "+" (i1, i2 : INTG) return INTG;
    ...
    private
       type INTG is  ...
end  INTG_model;
```

The composite types, however, will require package generators in order to construct Abstract Data Types for different subtypes as required. For the types set, tuple and map the generics facility of Ada may be used to construct Abstract Data Types from a generic template. For example:

```
generic
    type element is limited private;
   ...
package set_model is
    type set is limited private;
    function Empty_set return set;
    ...
    private
        type set is ...
end set_model;
```

Such a generic template may be easily instantiated for different element types as follows:

```
package INTG_set_model is new set_model (INTG);
```

Two composite types, the tree (or record) and the type union (typ_A = typ_B | typ_C | ...) cannot be generated using Ada generics, because they would need a variable number of generic parameters and some function names vary with the subtypes. For these types package generator programs are provided which take the specification of the type as input and generate the corresponding Ada package. For example, to generate the tree for the Activity type, we would create a text file containing the specification:

```
Activity :: s-id :       QUOT
            s-duration : INTG
            ...
```

This file would be passed to the tree package generator to produce the package (specification and body):

```
package activity_model is
   type activity is limited private;
    function mk_activity (id: QUOT; dur: INTG;
                          desc: quot; s, e: date) return activity;
    function s_id (a: activity) return QUOT;
    ...
    private
      type  activity is ...
end  activity_model;
```

4.2. Implementation of Package Specifications from Object Specifications

It is now fairly straight-forward to implement the package specifications of our application object types from their specifications. The first step is to implement the base type using one of the basic Meta-IV type packages or package generators.

For the type Date, the tree package generator is used to generate package "Date_imp_model" which implements the tree definition in the **Model** part of the Date specification. The package specification for Date is now easily constructed from the **Type, Needs_Types,** and **Interfaces** sections of the object specification and looks like this:

```
with INTG_model, date_imp_model;
use  INTG_model, date_imp_model;
package date_model is
    type date is limited private;
    function make_date (d, m, y : INTG) return date;
    ...
    private
        type date is new date_imp;
end date_model;
```

As there is no additional interface to the Activity type, package activity_model can be directly implemented using the tree package generator.

Package specification project_model is implemented by first using the generic map and set packages to generate package project_imp_model, and then constructing the package specification as in date_model above.

4.3. Implementation of Package Bodies from Object Specifications

Implementation of the package bodies involves for the most part a transcription of the **functions** part of the specification making any required syntactic alterations. For example function add_activity in the Project specification becomes:

```
function add_activity (p: project; a: activity) return project is
    begin
        if memb (dom(p),a) then raise undefined;
        else return merge(p, make_map(a, empty_set));
        end if
    end add_activity
```

Obviously since Meta-IV is a specification language and Ada a programming language there are some constructs which cannot be directly translated. Such constructs may have to be replaced by functions which produce the same result in an algorithmic way. The following section demonstrates, however, that Ada provides a mechanism for implementing some of these constructs which is not available in other programming languages.

4.4. Implementation of More Complex Constructs

ADT's which closely implement the pure Meta-IV types are useful for the prototyping of specifications. However, if an efficient production quality implementation is required, then some additional subprograms must be provided.

Significant improvements are made by including functions which replace more than one Meta-IV function. For example, *is_defined (p,a)* replacing *memb (dom(p), a)* avoids the creation of a temporary set.

We also provide generic subprograms which avoid the need for iterative or recursive scanning to perform selection or processing functions on sets, tuples, and maps. For example the construct

```
let a ε dom(p) and s-id(a) = id
```

may be implemented by instantiating the generic function

```
generic
   with function predicate (d:domain) return BOOL;
function select_one_from_domain (m:map) return domain;
```

as follows

```
declare
    function check_id (a: activity) return BOOL is
       begin
       return s_id (a) = id;
    end check_id;
    pragma in line(check_id);
    function select is new select_one from_domain (check_id);
 begin
    assign (a, select (p));
 end
```

As many as 30 such generic subprograms are provided for each of set, tuple, and map in order to improve efficiency and ease of programming with these types.

5. Analysis of the Method

Experience has shown that the recommended guidelines provide a systematic approach for developing Ada software. The method was found to be successful in providing a high degree of formality to the previously informal Object-Oriented Design on the one hand, while providing structure to the previously unstructured VDM meta-language on the other.

Designers found the clarity brevity and precision of the meta-language valuable for consolidating ideas and detecting inconsistencies at an early stage. The ability to generate a complete system specification before the start of implementation virtually eliminated subsequent redesigns.

The specification documents generated during the design and specification phases provide a well documented development history, as well as a sound foundation for the implementation. Even during system maintenance, the documents were often first referenced in preference to the code.

The basic Meta-IV packages and package generators proved to provide a powerful reusable base considerably reducing the implementation effort.

In projects with more than one implementor the use of the basic Meta-IV and Abstract Type of packages yielded software of a consistent style and quality. This significantly improved the ease of integration and subsequent maintenance.

It was also found useful to be able to generate early prototypes directly from specifications. The behaviour of these inefficient implementations could be observed in order to validate the specifications before introducing more details and efficiency considerations.

6. Conclusion

The combination of various techniques has been developed on the basis of previous experience with the constituent components themselves. The task was not to develop a unifying semantic base for the combination effort, but more to use each technique to its fullest advantage. Complementary work is performed to lay better foundations for such an approach in the ToolUse project in the ESPRIT framework.

Another direction of current work is the development of tools to support the method. Like in any manufacturing process the productivity and maintainability can be increased substantially by using powerful specialised tools in a well integrated manner.

Consistency and completeness checks, recording of development steps and problems encountered, up to semi-automatic code generation from specifications are functions to be supported by tools. Several toolsets are under development, mainly independently for the various components of the method described here. The Esprit Raise project is developing an extensive toolset for a derivative of VDM, extensive management support is the aim of SPMMS, and knowledge-based techniques are being employed to develop advisory tools to support the application of the method.

The Generics perspective sees advisory tools governing the application of the method described here, as well as others, with the individual steps supported by corresponding specialist tools, e.g. interface generators [Chedgey 87b] [Chedgey 87c], requirements analysis, code instrumentation etc. A framework for guaranteeing the interoperability of such tools exists in the Portable Common Tool Environment, for instance [Kugler 87].

References

[Ada 83] *Ada Language Reference Manual*, ANSI/Mil.Std 1815A, January 1983.

[AFFIRM 81] *AFFIRM Reference Library*, 5 Volumes: Lee, S. and Gerhart, S.L. (eds.), USC Information Science Institute, Marina Del Rey, California, Version 2.0, February 1981.

[Beech 86] Beech, D. (ed.), Gram, C., Kugler, H.-J., Newman, I., Stiegler, H., Unger, C., *Concepts in User Interfaces: A Reference Model for Command and Response Languages*, Lecture Notes in Computer Science 234, Springer verlag, 1986.

[Bjorner 78] Bjorner, D. and Jones, C.B. (eds.) *The Vienna development Method: The Meta-Language*, Lecture Notes in Computer Science 61, Springer verlag, 1978.

[Bjorner 80] Bjorner, D. and Oest, O.N. (eds.), *Towards a Formal Specification of Ada*, Springer verlag, Lecture Notes in Computer Science 98, 1980.

[Bjorner 82a] Bjorner, D. and Jones, C.B. *Formal Specification and Software Development*, Prentice-Hall, 1982.

[Booch 83] Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[Booch 86] Booch, G., *Object Oriented Development*, IEEE Transactions on Software Engineering, Special Issue on Software Design Methods, Vol. 12 (2), February 1986, pp. 211-222.

[Cameron 83] Cameron, J.R., *JSP & JSD: The Jackson Approach to Software Development*, IEEE Computer Society, 1983.

[Cameron 86] Cameron, J.R., *An Overview of JSD*, IEEE Transactions on Software Engineering, Special Issue on Software Design Methods, Vol. 12 (2) February 1986, pp. 222-240.

[Chedgey 86] Chedgey, C. et al., *Technical Annex to 4th Interim Report of ESPRIT 496*, Generics (Software) Ltd., Dublin, November 1986.

[Chedgey 87a] Chedgey, C., Kearney, S., Kugler, H.-J., *Using VDM in an Object-Oriented Development Method in: VDM - A Formal Method at Work*, Proceedings VDM-Europe Symposium, Springer Verlag, Lecture Notes in Computer Science 252, 1987.

[Chedgey 87b] Chedgey, C., *Papillon - Software Tools for Development of Graphical Software*, Proceedings of the EUUG Autumn '87 Conference, Dublin, September 1987.

[Chedgey 87c] Chedgey, C., *Papillon - A Support Environment for Graphical Software Development*, Proceedings of the 4th ESPRIT Conference, Brussels, North-Holland, Amsterdam, September 1987.

[Cohen 86] Cohen, B., Harwood, W.T., Jackson, M.I., *The Specification of Complex Systems*, Addison-Wesley, 1986.

[Connolly 86] Connolly, P., *Experimental JSD Rule System*, Project ToolUse, Insight.PC86h, Vector Software Ltd., 1986.

[Hoare 85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, London, 1985.

[Guttag 77] Guttag J.V., *Abstract Data Types and the Development of Data Structures*, CACM Vol. 20 (6), June 1977, pp. 395-404.

[Herefordt 87] Herefordt, H.M., Villadsen, P., *An Ada Package Supporting the Use of VDM for Ada Program Development*, Proceedings of the Ada-UK Conference, York, January 1987.

[Horgen 86] Horgen, H. et al., *Workplan for ESPRIT Project 510 ToolUse*, Phase 2, October 1986.

[Jackson 83] Jackson, M., *System Development*, Prentice-Hall, 1983.

[Jackson 85] Jackson, M.I., *Developing Ada Programs Using the Vienna Development Method (VDM)*, *Software Practice and Experience*, 15(3), 305-318, March 1985.

[Jones 85] Jones, C.B., *The Role of Proof Obligations in Software Design*, in Proceedings of the TAPSOFT Conference, Berlin, 1985, Springer Verlag, Lecture Notes in Computer Science 185, 1985.

[Jones 86] Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

[Kearney 86] Kearney, S., *OOD/VDM Methodology*, Project ToolUse, Generics,sk86t, Generics (Software) Ltd., November 1986.

[Koch 83] Koch, G., Epple, W., Sars, *A System for Application Oriented Requirements Specification*, 1983.

[Kugler 87] Kugler, H.-J., Lynch, B., *Uncle - A Case Study in Constructing Tools for the PCTE*, Proceedings of the EUUG Autumn '87 Conference, Dublin, September 1987.

[Le Blanc 82] Le Blanc J., Goda, J.J., *Ada and Software Development Support : A New Concept in Language Design*, IEEE Computer, May 1982, pp. 75-82.

[Mac an Airchinnigh 87] Mac an Airchinnigh, M., Burns, A., Chedgey, C., *Reusable Units - Construction Methods and Measure*, Proceedings of the Ada-Europe International Conference, Stockholm, Cambridge University Press, May 1987.

[Mickel 84] Mickel, S.B., *Experience with an Object Oriented Method of Software Design*, Adatec Conference, Brussels, 1984.

[Milner 80] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Sprinter-Verlag, 1980.

[Myers 86] Myers, W., *Ada catches on in the commercial market*, Editorial SoftNews, November 1986, pp 81.

[Ryan 86] Ryan, K.T. (ed.) et al., TCD.KR86B, *An Experimental Basis for ToolUse-Task 5.2 Report, Project ToolUse*, Trinity College Dublin, Dec. 1986.

[SLC 80] Freeman, P., Wasserman, A., *Tutorial on Software Design Techniques*, Third Edition, IEEE Computer Society, April 1980.

[Sommerville 85] Sommerville, I., *Software Engineering*, Addison-Wesley, 1985.

Papillon - Support Tools for the Development of Graphical Software*

Chris Chedgey
Generics (Software) Ltd.,
7 Leopardstown Office Park,
Foxrock,
Dublin 18,
Ireland.

ABSTRACT

The Papillon project (ESPRIT 496), entitled "A configurable graphics subsystem for CIM", has produced a prototype system to help the application developer build well-structured software which is initially devoid of representation or user-interface information. Graphical representations for application data types may be subsequently described without direct coding. Calls made to the application by "driver" software such as process control or simulation programs will now be reflected in the display. A user interface may be automatically generated to enable a human user to "drive" such applications as production planning and scheduling. The approach has its foundations in Object-Oriented Design (OOD), the Vienna Development Method (VDM) and the programming language ADA+.

1. Introduction

The Papillon project has produced a prototype configurable graphics subsystem for Computer Integrated Manufacturing (CIM). An important part of this subsystem is a collection of tools which on the one hand reduces the effort required to generate well structured application software, and on the other removes the need to implement a user interface.

While the example application domain used in the project is CIM oriented, the toolset itself is essentially application independent and should prove useful to software engineers from many disciplines. It is also important to note that while software engineering principles, including formal methods, have influenced the design of the toolset and as such the toolset complements the use of these techniques, it has been a requirement throughout that the toolset be usable by software developers who are not users of formal methods.

The paper will briefly explain underlying technology, a method which combines Object-Oriented Design, the Vienna Development Method and the programming language Ada. The main components of the toolset will then be introduced and their function described. The way in which the toolset is used

---

* The work reported here was partly funded by the Commission of the European Communities under the Esprit Programme.

+ Ada is a trademark of the U.S. Government, AJPO.

to rapidly generate a new application will be clarified by example. Finally, an analysis of the effectiveness of the tools is presented, followed by a discussion of future enhancements and the integration of the toolset into a broader software development environment.

2. The Software Development Method

A software development method which combines Object-Oriented Design (OOD), the Vienna Development Method (VDM) and the programming language Ada is described in detail in [Chedgey 87a] and [Chedgey 87b]. The user of the toolset is confined only to the informal parts of the method (OOD and Ada), while the use of the formal part (VDM) is optional. A brief description of the method is now given; the details may be obtained from the referenced papers.

2.1. Object-Oriented Design and Ada

Object-Oriented Design (OOD) is a system design methodology which is becoming increasingly popular in both software [Booch 83] and hardware [Organick 82] design.

Traditional control-based software has been data/procedure oriented, whereby programs are composed of a set of data and a set of procedures. With this approach, much of the expected benefit of decomposition is not realised because of the mutual dependency of many subprograms on the form and integrity of common data [Robson 81].

With the object-based approach to software a single type of entity, the object, represents both the data and procedure entities. Data structures are associated with a fixed set of subprograms which are the only operations defined on the object. Subprograms bodies, dependencies and implementation details are hidden. Such entities are sometimes called Abstract Data Types (ADT's) or Abstract Objects.

Programming languages which provide encapsulation and abstraction facilities are particularly suitable for this type of object-oriented programming. Ada provides these facilities through the "package" construct.

2.2. The VDM Meta Language

The Vienna Development Method (VDM) is a well established formal development method [Bjorner 78] [Jones 83]. At the kernel of the method is a formal specification language, one dialect of which is called Meta-IV [Bjorner 80a]. Based on sound mathematical foundations, Meta-IV has proved itself in a number of important software developments such as the DDC Ada Compiler [Bjorner 80b].

Meta-IV is a model-oriented language, providing a carefully selected collection of data types out of which new types may be modelled. The

semantics of the operations on the new data types is then specified in terms of operations on the basic Meta-IV types. It is possible to implement these basic Meta-IV types as ADT's in Ada [Jackson 85] [Mac an Airchinnigh 87] [Herefordt 87].

2.3. Combining OOD, VDM and Ada

One drawback of OOD is that it lacks formality and a disadvantage of the VDM meta language is its lack of structure. By combining the technologies and using the meta language to specify objects, formality is introduced to OOD while a rigid object-oriented structure is imposed on Meta-IV. A collection of Ada packages, generic packages and package generators (the latter operate on package schemata) renders the basic Meta-IV types and thus the Meta-IV object specifications easily implementable in Ada.

3. The Structure of the Toolset and Generated Application

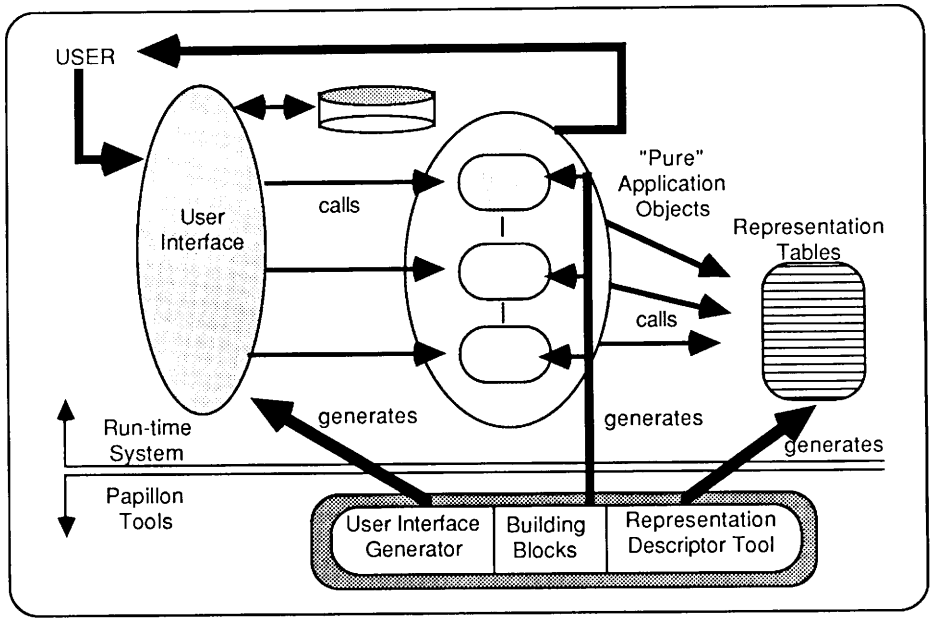


FIGURE 1

Figure 1 depicts the various components of the toolset and the structure of the generated run-time system. The horizontal divide between the toolset and the generated application is indicated. The toolset contains three basic

tools which are used by the application developer and user interface designers to generate three corresponding parts of the run-time system.

3.1. The Building Blocks

The Building Blocks are similar to the collection of packages, generic packages and package generators which implement the basic Meta-IV types. The way these are used is described in [Chedgey 87b]. Extensive use is made of generics within these components to render them an extremely flexible and reusable programming base.

The *simple* types, which are implemented as Ada packages, are integer, (INTG in Meta-IV), character string (QUOT), REAL (an extension to the Meta-IV types), and TOKEN.

Generic packages are used to create the *composite* types set, list (tuple) and map. These can be readily instantiated for different sub-element types (eg. set_of_INTG).

Package generating programs are used to generate records (tree) and type union (typ_a = typ_b **or** typ_c **or** ...). These package generators take the type specifications as input and generate the corresponding packages.

The difference with the Building Block types is that it is possible for instances of the types to be given a graphical representation, and any operations performed on the objects will be reflected in the display. This is achieved by extending the package specifications of the types to include graphical operations and by modifying the package bodies so that calls to subprograms also update the relevant graphical information and the display. The application developer, however, will generally use the Building Blocks as if these alterations had not been made. That is, he is unconcerned at this stage with the graphical representations.

To implement application Abstract Data Types their basic data structure is implemented either as one of the *simple* types or as a *composite* of already implemented types using the relevant Building blocks. Some of these types will require an additional higher level package or "semantic layer" in order to perform checks on parameters and/or to replace or extend the interface. These extra packages contain the "pure" application algorithms and are the only packages which require actual programming.

The user of VDM will first specify the application objects and their semantics using Meta-IV and, when the reification process is complete, will transcribe the specifications into Ada. The non-user of VDM will encode the semantic packages directly.

3.2. The Representation Descriptor Tool

This software tool enables the user interface designer to describe a number of graphical representations for each application object type and enter them into the representation tables (see Figure 1.).

The types of representations which it is possible to define for an object type depends partially on which Building Block type it was implemented from. The *simple* types will generally have a number of pre-defined representations available, although it is always possible to define new representations. Representations for these may differ according to size, font, colour etc.

When defining representations for *composite* types, the previously defined representations which are to be used for the sub-types are specified. Positioning rules must also be specified, for example whether the sub-objects are to be automatically or manually positioned.

A static piece of graphical data (such as a surrounding box) may be associated with any representation and will be displayed for all objects of the particular type. This is entered by means of a graphical editor.

Once representations have been defined for any application object type, it is possible for instances of the type, which are created by "driver" software to be displayed (though the driver software need not be aware of this). As the driver software modifies objects, their representations will be modified accordingly.

3.3. The User Interface Generator

Many applications require that a human user be able to "drive" the application. The User Interface Generator creates a handler which allows a user to create, manipulate and observe instances of the application object types. The result is a mouse/window/menu -driven interface which will be described in section 5.

4. Development of an Example Application

The chosen application domain is that of Production Planning and Scheduling (PPS). A very brief description of this is given here.

The user of the application is the manager of a production unit in a discrete manufacturing plant. The manager receives orders from customers which he must fulfil using a pool of resources available to him. He wishes to allocate the resources in such a way that they are as near to being fully utilised as possible, but not being over utilised.

The manager will wish to input a plan for each order in the form of a Pert chart, with the nodes of the chart representing individual activities and edges indicating which activities must be completed before others can commence. He will also wish to view the activities in the form of a Gantt chart with the current time line displayed. In order to achieve optimum resource allocation he must be able to examine the resource loadings as bar charts. Depending on the manager's task in hand, he may require different pieces of information displayed on the screen, and he should have maximum control over this.

4.1. Construction of the Application Software

Figure 2 shows how the "pure" application object types may be easily modelled using the Papillon Building Block data types. The object types are presented in hierarchical order. Meta IV is used in Figure 2 only as a convenient notation. The following brief description should adequately explain the implementation.

```
Order_plans = STRING m-> Order_plan
Order_plan  = Activity m-> Connection_set
Connection_set = Connection-set
Connection :: s_delay : INTEGER
              s_activity_id : STRING
Activity   :: s_id    : STRING
              s_dur   : INTEGER
              s_desc  : STRING
              s_start : Date
              s_end   : Date
              s_resources_used : Resource_descriptors
Resource_descriptors = Resource_descriptor-set
Resource_descriptor :: s_resource_id : STRING
                       s_amount : REAL
Resource_pool = Resource-set
Resource :: s_id : STRING
            s_amount_available : REAL
            s_requests : Request_set
Request_set = Request-set
Request :: s_start : Date
           s_end   : Date
           s_amount_requested : REAL
Date :: s_day   : INTEGER
        s_month : INTEGER
        s_year  : INTEGER
```

FIGURE 2

The type Order_plans is a mapping from unique identifiers onto order plans. An Order_plan is a mapping from activities to the set of dependent activities (activities which cannot commence until this activity has been completed). A Connection_set is a set of connections. A Connection is a record containing the time delay before the dependent activity can commence and the identifier of the dependent activity (these fields may be selected by the functions "s_delay" and "s_activity_id"). An Activity is a record containing a unique identifier, a duration, a textual description, a start and end date and a set of resources required by the activity.

The meaning of the remaining object types should now be clear to the reader. Once the structure of the application object types has been decided they can be implemented directly using the Papillon Building Blocks. Some of the object types will require an additional semantic package. For example not all triplets of integers create a valid date, and an interface to the Order_plan type containing such operations as add activity, delete activity,

schedule, etc is required rather than the interface to the basic map type with operations such as merge, domain, apply etc. The higher level operations are implemented as calls to the operations provided by the basic types.

4.2. Defining Representations

The representations for the object types may now be defined with the Representation Descriptor Tool. Because of the hierarchical nature of object type representations, it is usual to start with the objects lowest in the hierarchy and work upwards. In the example one would start with types INTEGER, STRING and DATE. For such low-level and frequently used types, it would be normal to use previously defined (standard) representations (though it is possible to define new ones).

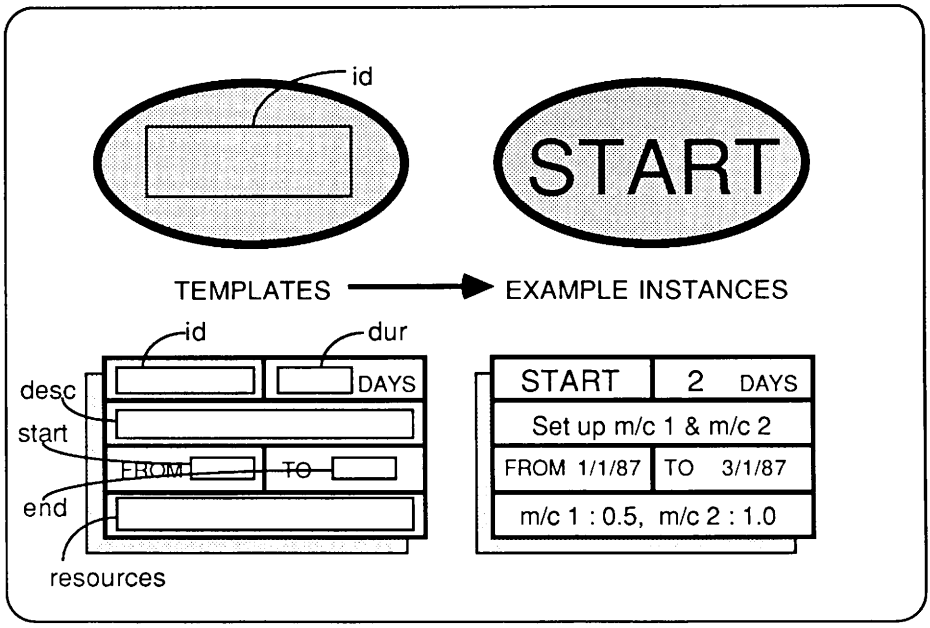


FIGURE 3

Two possible representation templates and corresponding instances for the Activity type are shown in figure 3. The labelled boxes in the templates define the positions of sub-objects of the Activity type. The rest of the templates comprise the static data. When we come to describe representations for the Order_plan type we can use the upper Activity representation to generate an easy-to-read Pert chart and the lower for a detailed chart. It is possible to specify that the user locate the Activities in the Order_plan (Pert chart format) or that they are listed in rows and columns, perhaps in a

specified order (directory format). We can also specify a bar chart representation for the Order_plan type. One of the possible representations for the Resource type is a histogram showing the resource loading over time.

The run-time system can be completed now by using the User Interface generator to create the software modules which will enable the end user to manipulate the application objects.

5. The Run-time System

A possible snap-shot of a run-time system generated for the example application is depicted in a simplified form in Figure 4. The arrows in the figure indicate relationships between the windows and do not appear in the actual display.

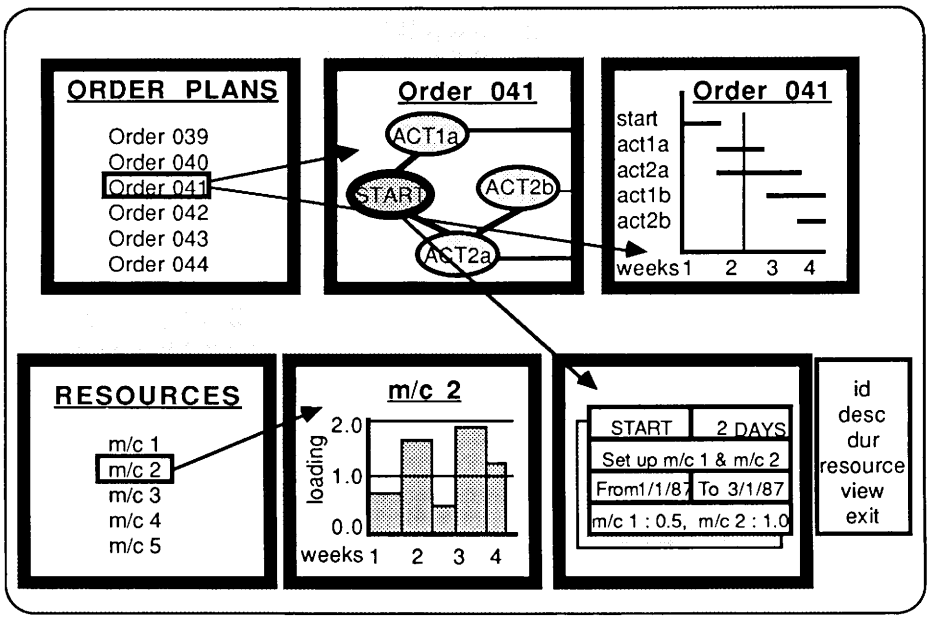


FIGURE 4

The user has loaded up the order plans and associated resources. Both of these have been chosen to be displayed in a dictionary format. He has then chosen to view one of the order plans (Order 041) in both Pert chart and bar chart format. When he modifies the order plan, both representations will reflect the change. He has then chosen to observe/edit one of the Activities in the plan ("START") and this has appeared in detail in its own window. At

present this is the window into which commands are directed and the menu of available commands for Activities is displayed.

The user has also chosen to view one of the resources (m/c 2) in the form of a histogram. As order plans are modified and the resource loadings for m/c 2 change, the histogram will also change.

The user may switch between windows; for example, from working with activity "START" he may move to resource "m/c 2". He may change the representation of an object or open a new window containing another representation. He has other viewing operations available such as panning and zooming on the Pert chart or shifting the displayed time slot on the Gantt chart or histogram.

When the user has finished observing/editing the activity, the activity window will close and the order plan will be updated. When he has finished with the order, the order windows will close. Eventually, when all windows have been closed, the user will be able to transfer the modified Order_plans and Resource_pool to permanent storage and perhaps load some other objects for modification or observation.

6. Analysis of the Papillon Prototype

6.1. Reconfigurability of the User Interface

This is an important aspect of the toolset. Because the user interface software is independent of the application software, the representations of object types may be edited, replaced or added to at any time (conceivably even at run-time). It may occasionally be advantageous to supply the Papillon tools with the application code in order that the user interface may be modified by OEMs or even end users to suit their needs as closely as possible.

6.2. Suitability for different applications

An analysis of the type of applications for which the Papillon tools are most suitable will require more experimentation. However, it can be expected that the approach will work best for those applications which are amenable to Object-Oriented Design. Such applications tend to be data-oriented rather than control-oriented.

Although the prototype currently supports the various object representations associated with a particular application domain, many of these representations will also be relevant to other applications. However, in order that derived tools be useful for a wide variety of applications, a set of representations must be provided which fully supports each application. A certain amount of investigation has been undertaken which indicates the particular suitability of the tools to model-oriented applications such as parts of process control or simulation.

6.3. Comparisons with User Interface Management Systems (UIMS)

The work undertaken on the Papillon project is sometimes compared to that on User Interface Management Systems. There are however some important differences between the Papillon toolset and the more traditional type of UIMS [Pfaff 83].

In a UIMS, the user interface designer usually describes the user-machine dialogue and the overall form of the user interface in considerable detail by means of a fairly complicated description language. The application implementor then either "fills in the blanks" where the UIMS will call application routines, or writes his applications software to call the UIMS when input or output is required.

This approach means that the application implementor must write a certain amount of user-oriented software, although this will be minimised by the UIMS. There is generally little encouragement by the UIMS towards the generation of well-structured software which is likely to enhance reliability and reuse. Also, one can generally say that the tighter the coupling between UIMS and the application software, the less the ease of configurability of both the application and the user interface.

With the Papillon approach on the other hand, the application implementor makes no user-oriented calls, rather he is free to concentrate on the pure application algorithms. He is encouraged and indeed helped to generate his software quickly and in the form of well-structured, reusable and easily reconfigured components. As there is complete separation of application from user interface, the configurability of the user interface is maximised.

7. Enhancement of the Toolset

For the remainder of the project and the subsequent exploitation phase, during which the product AnimAID will be developed by Generics, a number of advancements will be made to the toolset. Some of the envisaged areas of improvement are now mentioned.

The number of possible representations for the various Building Block types will be expanded. The available Building Block types themselves may be expanded upon to provide new types or higher level composites of existing types.

The Representation Descriptor Tools could be improved to provide more checks for valid representations and more assistance to the user interface designer. A facility for specifying and editing prompt and error messages should be provided. Another useful feature would be the support of generic representations for generic object types.

As is typically the case, there are a number of important areas for improvement of the generated user interface. Examples include the incorporation of more direct manipulation for input, the improvement of error handling, the introduction of "explosion levels", and the ability of the user to create his own commands as a sequence of existing commands.

The relative importance of the various possibilities for improvement are becoming clearer with experimentation.

8. Incorporation Within a More Comprehensive Tool Environment

While the tools described here form a powerful development aid in their own right, there are implicit links with the broader software development method described in [Chedgey 87b]. Other tools which support this method are currently under development within Generics. It is envisaged that these tools will be combined within a common environment to provide support for the entire software life cycle. The ESPRIT program provides a basis for such an environment through the Portable Common Tool Environment (PCTE) and Generics is currently involved with PCTE developments [Kugler 87].

9. Conclusions

The toolset described in this paper makes use of Object-Oriented Design principles in order to simplify the normally complex task of semi-automatic user interface generation. Because the generated user interface closely adheres to the application object structure, the benefits associated with OOD are retained in the generated graphical application.

The potential for reuse of software components in very high. Once objects have been implemented for one application they are easily reused in another, along with the representations which have been defined for them. The reusability of the basic Building Block types themselves greatly reduces the programming effort when implementing new objects. When the reconfigurability of software components is combined with that of the user interface, the result is a highly flexible system.

The prototype in its present form is immediately valuable as a tool for prototyping and testing of Ada packages. Further development in the areas mentioned in section 7 are expected to result in a powerful toolset for the development of highly configurable graphical software of production quality.

Acknowledgements

I wish to thank all those in Generics and Trinity College Dublin who have worked on the implementation of the toolset.

References

[Booch 83] Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[Bjorner 78] Bjorner, D., and Jones, C.B., (Eds.), *The Vienna Development Method : The Meta-Language*, Lecture Notes in Computer Science, Vol. 61, Springer Verlag, Berlin, Heidelberg, New York, 1978.

[Bjorner 80a] Bjorner, D., *Reference Manual for the Meta-Language* in Toward a Formal Description of Ada, Lecture Notes in Computer Science, Vol. 98, Springer Verlag, Berlin, Heidelberg, New York, 1980.

[Bjorner 80b] Bjorner, D., and Oest, O.N., (Eds.), *Towards a Formal Description of Ada*, Lecture Notes in Computer Science, Vol. 98, Springer Verlag, Berlin, Heidelberg, New York, 1980.

[Chedgey 87a] Chedgey, C., Kearney, S., and Kugler, H.-J., *Using VDM in an Object-Oriented Development Method for Ada Software*, in VDM - A formal Method at Work, proceedings of the VDM-Europe Symposium 1987, Lecture Notes in Computer Science, vol. 252, Springer Verlag, 1987.

[Chedgey 87b] Chedgey, C., Kearney, S., and Kugler, H.-J., *Developing Ada Software Using VDM in an Object-Oriented Framework*, Proceedings of the EUUG Autumn '87 Conference, Dublin, September 1987.

[Chedgey 87c] Chedgey, C., *Papillon - A Support Environment for Graphical Software Development*, Proceedings of the 4th ESPRIT Conference, Brussels, September 1987, North-Holland, Amsterdam.

[Herefordt 87] Herefordt, H.M., Villadsen, P., *An Ada Package Supporting the use of VDM for Ada Program Development*, Proceedings of the Ada-UK Conference, York, January 1987.

[Jackson 85] Jackson, M.I., *Developing Ada Programs using the Vienna Development Method (VDM)*, Software-Practice and Experience, 15(3), 305-318, March 1985.

[Jones 86] Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

[Kugler 87] Kugler, H.-J. and Lynch, B., *Uncle - A Case Study in Constructing Tools for the PCTE*, Proceedings of the EUUG Autumn '87 Conference, Dublin, September 1987.

[Mac an Airchinnigh 87] Mac an Airchinnigh, M., Burns, A., and Chedgey, C., *Reusable Units - Construction Methods and Measure*, in Ada components: libraries and tools, Proceedings of the Ada-Europe International Conference, Stockholm, May 1987, Cambridge University Press, Cambridge 1987.

[Organic 82] Organick, E.I., *A Programmers View of the Intel 432 System*, McGraw-Hill, New York, 1982.

[Pfaff 83] Pfaff, G.E., (Ed.) *User Interface Management Systems*, Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1983.

[Robson 81] Robson, D., *Object-oriented Software Systems*, Byte, August 1981.

EXPERIMENTS WITH THE USER INTERFACE FOR UNIX MAIL

Peter R. Innocent[*], *Gerrit C. van der Veer*[**] *and Yvonne Waern*[***]

[*]School of Mathematics, Computing and Statistics,
Leicester Polytechnic, UK.

[**]project coordinator,
COST-11-ter working group HUMAN FACTORS IN TELEMATIC SYSTEMS,
Department of Psychology, Free University of Amsterdam, the Netherlands.

[***]Psychology Department, University of Stockholm, Sweden

ABSTRACT

Designing the user interface and application interface are only two aspects of the complex work that has to be done to provide users with systems that they both are able to use, and like to use. An interdisciplinary approach is necessary to define, design, and construct these interfaces. For the COST-11-ter working group HUMAN FACTORS IN TELEMATIC SYSTEMS electronic mail was used as an example to illustrate the way to develop user interfaces. As we wanted to start from an existing situation (an application system in actual use), we chose an application that is certainly neither the best, nor the most advanced. It is, however, a standard component of UNIX which is becoming a standard operating system for a growing diversity of generally available machines.

Beginning with an analysis on task level, we restricted the task domain to the core functions of electronic mail proper, i.e. the preparation, sending, receiving,and administration of electronic messages, and the learning of this facility in interaction with the system (not aiming at a special learning program). We explicitly excluded from the task domain in our study all functions regarding file management outside the mail system, network protocols and addresses, and editing.

We collected experiences of novices and expert users with the existing system. Listing strong points and weak points, inconsistencies and notions on feedback, we drew conclusions on the function of attention, evaluation, memory, interpretation and generalisation. We derived a definition for a "better" functionality (the semantic level) for an application interface. We defined the conceptual model of the system (or the user virtual machine) in terms of objects, attributes, actions, and metacommunication. In the next phase of our project, we will construct an application interface (including communication between the system and the user) and a user interface (including metacommunication), applying different description methods and definition languages, and we plan experiments on the effects of functionality of different interfaces. At this moment we concentrate our efforts on the comparison of several alternative systems of mnemonics for commands, prompts, and messages, on which we will report the first experimental results.

1. INTRODUCTION

This contribution describes the progress of the working group "Human factors in telematic systems", a project within the COST-11-ter Human Factors action, sponsored by the Commission of the European Communities, DG XIII, information technologist and telecommunications task force. The project coordinates research on the feasibility of human factors (cognitive psychological) requirements and guidelines for the definition and design of human-computer interfaces in OSI (Open Systems Interconnections). An important role in this effort is played by the distinction between categories of users and between categories of tasks or task elements. The goal of the project is to contribute to the standardisation of definition languages for user interfaces in OSI environments, enabling the application of human factor guidelines by the designer.

Work on the development of open systems standards is continuing at every level in the OSI model. Such work often takes the form of devising agreed protocols for communication within a layer and between layers. The protocols are then specified formally and subject to testing and approval. The application and presentation layers present certain problems in that people are involved and cannot be formally represented in this strategy. This is particularly important to the user interfaces of both applications and the other visible components of an open system. Prior work under the COST 11 bis established a human factors working party comprising a multi disciplinary group. This group developed ideas which have been published. The working group has thus far completed work on the inventory of theoretical and empirical knowledge on human factors in OSI (Van Muylwijk, Van der Veer and Waern, 1983; Wheeler and Innocent, 1983), and on the feasibility of design criteria (Hannemyr, 1983). A representational framework and design strategies have been developed (Hannemyr and Innocent, 1985), and a feedback model for metacommunication inside the virtual machine (Van der Veer, Tauber, Waern and Van Muylwijk, 1985).

Trying to solve the problem users face when confronted with a system they do not know in all its details (and which they may never be able to know completely), we will have to answer the following four questions:

a. Is it possible to **describe** the user interface part of the conceptual model of an existing real life computer application.

b. Is it possible to **define** the conceptual model for the user interface component of a real life computer application that is acceptable from cognitive psychological and cognitive ergonomic viewpoint, taking care of interactions at the user interface, individual differences, and learning processes.

c. Is it possible to **construct** a user interface according to the conceptual model as we have defined it.

d. Is it possible to **demonstrate** that the new user interface leads to "better" man-machine interaction than does the already existing interface.

The group works along two interconnected lines:

a. user modelling
   This title covers the field of human factors in relation to human computer interaction. The activities along this line are directed at the analysis and definition of the requirements of users in an OSI applications environment. These have to be structured and described in a form that is suitable for incorporation in an ISO (International Standards Organisation) standard. An important aspect of this is the distinction between the human-machine interaction proper (directly aimed at the delegation of sub-tasks to the machine), and "metacommunication" consisting of off-line and on-line interaction concerning the interaction proper (e.g. documentation, help-facilities, error messages, implicit metacommunication by the wording of commands or the choice of icons).

Both the interaction concerning delegation of tasks to the computer and on-line metacommunication are located in the user interface. The conceptual model of this interface should be defined in relation to both the task space and the intended user-group. In the definition of the conceptual model of a system adaptation to relevant user variables and individual differences may be taken care of. The metacommunication facities in the user-interface are designed to evoke the development of a mental model in the user that is consistent with the conceptual model of the system.

b. language for defining user interfaces
This activity aims at developing and testing the feasibility of implementation of a flexible language for the definition of user interfaces for OSI application, along the guidelines developed in the other line of the project. The distinction of functional different levels in the command language (c.f. Moran, 1981) will be a useful method to locate the source of inconsistency between the user's mental model and the actual user interface, and to develop an adequate structure in the interface to cope with requirements of learnability, ease of use, and the avoidance of interactive deadlocks (A situation in which specific human-machine interaction cannot proceed because both the user and the machine are unable to infer what to do next).

The human factors requirements will have to be translated into design standards, to be incorporated into the layers of the ISO model for OSI, with the help of an appropriate language interface.

To answer the basic questions we have chosen an example of an application that makes sense in an OSI environment: UNIX mail.

It is important to realise that the version of the mail utility used was chosen because it is a standard component of UNIX systems and not because it is the best program for handling mail under UNIX. There are many different mail service programs available under UNIX with great varieties in user interface and functionality. For our experiments we needed generality and availability. UNIX is being more widely used by none computer specialists and in a network environment. Hence, our interest in approximating open systems usage by novice users in a mail application centred on UNIX mail. Our goals in the first phase of this work are:

a. to analyse the electronic mail application from a users point of view.

b. design and run multi site pilot experiments into novice use of an electronic mail application.

In order to achieve these goals new representation and simulation languages and other tools had to be developed. These are described in detail in van der Veer et al (1987). This paper describes some of the main results of the work with reference to the interests of Unix users and experiments in using Unix mail.

2. ANALYSIS OF AN ELECTRONIC MAIL APPLICATION

2.1. cognitive ergonomic aspects of electronic mail

In order to analyse the users' requirement on a particular application, it is useful to understand what general requirements are posed by the human cognitive system, what these requirements amount to, and what they actually mean when a particular computer system is concerned. We will consider general cognitive psychological requirements, applied to the situation where a computer user wants to use a mail system in an open systems interconnection (OSI). The general cognitive requirements are schematically represented in Figure 1 below.

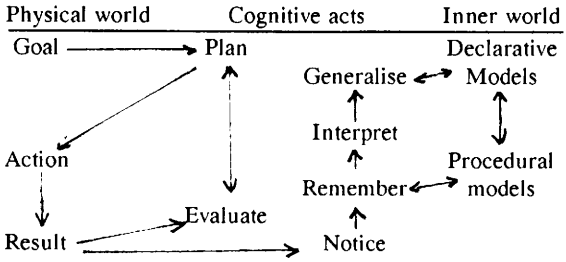


Figure 1. The interaction between physical world, inner world and cognitive acts.

This figure represents a crude simplification of the cognitive complexities of the human being. This simplification is made to highlight the most important aspects, when computer use is concerned.

The designation "physical world" corresponds to the perceptible aspects of the current situation. The perceptual and motor acts which are required to deal with these aspects are not covered in the figure. The denotation of "goal" refers to a goal as described from the "outside", for instance in terms of a particular configuration of symbols at the screen to be attained by the user. By "action" is meant an action performed by the user with respect to the system, e.g. a command given, or a menu selection. Other kinds of actions by the user are covered by the notion "cognitive acts". By "result" is meant the feedback given by the system, either in terms of a direct echo of the keys pressed (as in typing letters), or in terms of the effect accomplished by a command menu selection or direct manipulation. The result as here referred to is in some way be perceptible by the user, and does not refer to the computational processes in the system.

The cognitive acts mediate between the physical world and the inner world. We can regard them as taking place within the constraints of the working memory. The different cognitive acts will be described further below.

The inner world refers to the content of long-term memory which is relevant for the use of the computer system in the particular task. The notion of declarative and procedural models will be described further below.

The situation to be analysed here is goal-oriented. Pure exploratory activity will not be covered in this analysis. In a goal-oriented situation, a *goal* is posed (either by the user himself or by somebody else). This goal can be reached by performing a sequence of *actions*. The *result* of the action is noticed and evaluated with respect to the goal. The cognitive acts related to goal-handling consist of *planning* what actions to take in order to achieve the goal and *evaluating* the results with respect to the goal. Eventual discrepancies between results and goal are processed by other cognitive acts.

Related to the evaluation is the cognitive act of *noticing*. The user will notice the results which lie within his focus of attention. This means that there is no guarantee that all relevant feedback from a computer system to a user really is noticed by the user. It also means that some irrelevant information may be noticed, if it lies within the users' focus of attention. Knowledge of factors affecting human attention are thus important in order to be able to design "user-friendly" systems. Some relevant factors are the following:

a. The users will attend to aspects which are related to their plans, i.e. their visual focus of attention will be at the place where they expect the results to turn up. This means that the users well may miss information which is out of focus.

b. The user will suppose that the information contained within the visual field in some way is related to his current goals and actions. This means that the user can risk to attend to information which is no longer relevant, if the screen is not continually updated with respect to the current goal.

c. The user will attend to aspects which are related to their "model" of the system and task they are performing. This means that feedback which is not compatible with the current model will risk not to be noticed, and that eventual irrelevant feedback which is compatible with the current model will be noticed. The notion of "model" will be further explained below.

Next cognitive act is related to *remembering* the actions performed and their results. This is crucial for the users' possibility to learn the system. There is no guarantee that all that which is noticed will be remembered at a later time. The conditions for memory have been extensively investigated by psychologists, and all details cannot be presented here. The most important generalisations are the following:

a. The meaningfulness factor.
That which is compatible with prior knowledge will be easily remembered. This factor is related to the models of the system and task, since models represent one aspect of prior knowledge.

b. The repetition factor.
Repetition gives a higher probability for remembering.

c. The consistency factor.
Regularities facilitate remembering. This factor is related to the ability of human beings to use variables and abstractions. Users will try to find regularities between commands or menu selections, and systems which capitalise upon recurring similar action sequences will be easier to learn.

In the figure, the cognitive act of *interpretation* is separated from remembering only for clarity. In effect, interpretation is intimately related to remembering. The (conscious) memory of a noticed act and its result is always retained in an interpreted form. The interpretation is intimately related to the model of task and system, which will be described below. By interpreting the acts and their results the users make sense of the system. When the result of the act does not correspond to the result expected, the interpretative activities will be particularly prominent. Several different interpretations may be put forward and tested in renewed plannings and acts. The interpretation arrived at does not necessarily correspond to the "true" way to describe the actual act-result sequence. Instead, it is often the case that users arrive at interpretations which deviate from the actual functioning. These deviations may or may not be detrimental for the users' understanding of the system. If they lead to wrong or nonefficient plans and actions, they are of course detrimental. If they inhibit the user from creating useful generalisations, they are also detrimental.

The final cognitive act here described is concerned with *generalisation*. By this is meant the activity by which different experiences of the system and different interpretations are collected together under a common concept and treated as similar. Such generalisations are always attempted by human beings, and contribute, as was said above, to the ability of human beings to learn. Generalisations are derived by utilizing semantic and structural similarities. This activity is related to the use of models, both by being derived from them and by contributing to the creation of models.

Models can be defined as the more or less connected set of long term memory knowledge a user has available about a system and the task to be performed. The model can either be given to the user from the "outside" (by a teacher, system designer, etc.) or it can be derived by the user himself by his experiences with the system. It is here suggested that models can contain knowledge of two different kinds which sufficiently different to merit separate consideration. The distinction is based upon the well-known differentiation between declarative and procedural knowledge.

A *procedural model* contains the particular procedures to be performed in order to perform a certain task in the system. A user who has a procedural model does not necessarily understand WHY the procedures work, nor how to combine the procedures into more general methods. A *declarative model* implies an understanding of the structural aspects of the system, where concepts are related to each other without being tied to any particular procedure. The most important thing to notice is that these different types of models may not be related to each other. A user may well have a declarative model, without being able to apply it in a particular task. He can also well know the procedure, without being able to describe it.

The only knowledge which a user evidently applies in order to cope with a system is procedural knowledge. It is not always necessary to provide users with declarative explanations of why or how a system works. This is the main reason behind so called "direct manipulation" systems. However, there are conditions which make direct manipulation systems impossible or impractical. Hardware and software restrictions (particularly in the Open System Interconnection situation) hinder the implementation of direct manipulation. The task itself may benefit from a declarative model, particularly when it is concerned with abstract entities, variables, or repetitions.

The cognitive acts of interpreting and generalising from the actions performed and their results serve to transform the procedural experience into a declarative model. By using declarative models the users may get the procedures to fit within the constraints of working memory. However, the problem with declarative models consists in the difference between their form and the form of a procedure. A procedure most often has the form of a conditional sentence: if you want to accomplish this, then do that. A description is a declarative model. It may, however, take the form of an explanation: You can do this, because..., or a metaphor: This is similar to... Declarative models therefore have to be mapped into procedures in order to be of practical use, and procedures have to be transformed into declarative sentences in order to be understood.

A complex interaction is going on between the outer world of goals, actions and results, and the cognitive acts of planning and evaluation, noticing, remembering, interpreting and generalising, which all feed into and are fed back by the models. Problems can easily surface at one place of this complex interplay and hereby affect all other elements involved. We will investigate some of these problems by taking a concrete case as point of departure. An "ideal" computer system should at least avoid known problems.

Some methods are based on the believe that the easiest way to minimise problems consists in tutoring the users at the points where they lack procedural or declarative models. This reflects a very simplified view of education. We should know be now, that education is no guarantee that the student will arrive at the "right" knowledge. For instance, a currently used solution is to give the user information about the procedures to perform in the system (Breuker and De Greef, 1985). This kind of tutoring relieves the problem of noticing (here noticing the relevant information in the manual), but does not provide real help, when the acts of remembering, interpreting and generalising are concerned. Current discussion about intelligent tutoring systems shows that these problems are neither trivial, nor easily solvable.

2.2. UNIX mail from a user's viewpoint

The present section tries to categorise users' observations and provides some (modest) guidelines to people who might wish to construct an electronic mail system. At first glance criticism seems to dominate the paper. But its goal is to summarise experiences, not to derogate a powerful and useful tool. The specific system which was studied was the seventh edition of the UNIX system, update

2.1. For obvious reasons not all the characteristics of the UNIX mail system (UMS) are classified. Only a summary will be presented.

2.2.1. strong points

The UMS gives (especially in combination with "write") tremendous opportunities for inter-user communication. The strong points of the UMS in this respect are:

a. power
   A few examples of the power of the UMS:
   - It is relatively easy to send mail to other institutes (when UUCP is installed).
   - It is possible to send secret mail (using xsend).
   - The UMS provides sophisticated and fast mail handling (e.g. looking through your mail in reversed order).

b. speed
   To send a short letter to another user can be done within 15 seconds.

c. complete integration with the UNIX operating system. For instance:
   - Sending of files from your directory to other users (using pipes or re-directs).
   - The possibility of giving commands to the shell while looking through your mail.

2.2.2. weak points

The term "weak points" has two limitations. Firstly, it doesn't mean weak in an absolute way but only relative to the rest of the system. Secondly, the only weak points that are considered in this paper have cognitive ergonomic nature. That is, they refer to user friendliness, not to technical aspects.

a. inconveniences
   The word "inconveniences" is used for aspects of the UMS which are not vital shortcomings but do cause loss of time, increased error rate etc. The aspects mentioned below are of course an arbitrary choice but probably most "heavy users" will recognise them as unhandy.
   - When sending a message fails the message is saved in the file "dead.letter". When another mailing fails this file is overwritten without informing the user.
   - When a letter prepared with "mail" has been sent successfully there is no possibility to save a copy of this letter (except when you send it to yourself).
   - A user is informed if he has any mail just after a successful login. But there is no (fast) way of finding out who has sent the mail or how many letters there are.
   - When entering the UMS there is no way of knowing how to invoke the help-menu before you have seen it.
   - While working with your mail 'x' and 'q' turn out to be not the only ways to go back to the shell. Control d gets you back. And if you are viewing your last message so do '+', 'd', 'n', 's', 'm' and return.

b. inconsistencies
   With "inconsistencies" are meant events, feedback and commands which are not logical or consistent with the rest of the system. Some examples:
   - When using the UMS the command "ddddghtyt&&543*" will delete a message. Obviously the shell does more error checking. Strangely ."sfgrthg" is not equal to "s" but "nfhytnvc" is to

"n".

- If a message is sent from the shell to two users who are not listed (Ziggy and Elton) the following error message appears:

  mail: can't send to Ziggy.
  mail: can't send to Elton.
  Mail saved in dead.letter.

  If you do the same from the UMS (using m) you get:

  mail: can't send to Ziggy.
  Mail saved in dead.letter.
  mail: can't send to Elton.
  Mail saved in dead.letter.

  Although not entirely cryptic, it is more confusing.

- When the user wants to see the previous message in the UMS and there is no previous, then the same message is shown again. If (s)he requests the next and there is no next (s)he is sent back to the shell automatically.
- "mail xxxxx" (from shell) and "m xxxxx" (from UMS) are essentially the same commands. So why not give them the same name? (typing "mail xxxxx" in the UMS causes an error)

c. lack of feedback
The UNIX operating system and all its utilities give no feedback if a process is started correctly. The only evidence of a successful completion of the process is the re-appearing prompt. This approach is called "silent mode". Especially novice users would like to have more feedback when they use the UMS. ("But how do I know my message is sent?") Before we criticise this unfriendly behaviour it is good to know that UNIX, when it was developed in the early seventies (!), was intended as programmer's workshop. Its users therefore, would be well-trained professionals who do not need and want "useless" feedback messages. But today the system is used by people with very little experience and the lack of feedback is often a problem.

2.2.3. evaluation

The UMS is still a powerful and useful utility but these characteristics are not sufficient. Today, many non-experts are using the UMS and sortlike programs. Their demands are of cognitive ergonomic nature: user friendliness, learnability etc. If software is developed without considering the human factors, it will take longer to learn and will result in more errors by users. The great advantage of UNIX and its components is adaptability. Until now this advantage has only been used to improve the system technically. Since the system is becoming popular very fast in the world of office-automation maybe this is the right time for improvements concerning the area of human factors.

2.3. an example of various problems encountered in UNIX mail

An exploratory study was performed in Stockholm, where one subject who did not know anything about computers was asked to perform some simple tasks in a simulated part of UNIX mail. The subject was given a short introduction to UNIX mail. The subject was not given the manual (the reason is that in an open system you might not have all manuals available). The subject was told that

he could get help by pressing the "?" key. At the operating system level, help was given orally (according to some simple rules). At the mail level, only the in-built help was used.

The following difficulties were encountered (in order of severity):

a. The subject did not understand how to send a letter (once he had written it). From the subject's comments and the registrations of the interactions, the following reasons for this difficulty were found:
   - The subject could not find any way to check that the letter really was sent, even though he tried hard. (There is no way).
   - The subject could not understand the description of commands in the help menu. In particular, he tried to use commands in "send" mode which were only applicable in "read" mode. (For instance he tried to "remail message", when he was through writing.)
   - The subject had no way of knowing "where he was" (i.e. in the operating system, in "send" mode or in "read" mode).
   - The subject was uncertain about the difference between received mail and written mail. This uncertainty showed up when he tried to check that the letter was sent by checking his own letters.
   - The subject did not know how a written letter is treated by the system, i.e. he did not know that finishing the mail session by "ctrl-d" also sends the letter.

b. The subject did not understand when the help-text could be derived. This difficulty is related to the layout of the screen. In fact, the help-text is only available in "read"-mode. However, in this particular case, the subject had first read a letter, and then asked for help. Thereby the help-text was available all the time he tried to send his own letter. It was this help-text that the subject used to find out how to send the written letter.

The subject's difficulties in handling the system can be summarised in the words of the letter written by the user:

> "This lousy system makes me crazy!"

Why were the problems so difficult to overcome? One of the main reasons lies in the fact that the system does not make the most important distinction sufficiently clear: the distinction between "send" and "read" mode. The very faint hint lies in the prompt, which consists in a "?" in read mode and nothing in send mode. Combined with the cluttered screen, which showed the "read" commands also while the user was writing, it is rather evident that the user did not notice the important distinction. Not noticing it, it was of course impossible for the user to understand which actions lead to which results, and thus to make a coherent plan for writing letters.

2.4. conclusions for an "ideal mail" system

It is often hold that an "ideal" system should conform to users' expectations. However, users of new computer systems do not usually hold any particular expectations on the system. The new system offers new facilities, and it is not necessary that these comply with the facilities which already are familiar to users.

Instead, it is necessary that the system as a whole complies with the users' cognitive requirements:

a. attention.
   Human beings' attention is restricted. It is therefore essential that attention is directed to the right place. Users' prior knowledge will make them attend to particular things (and neglect

others). If a system includes an important distinction (such as between "read" and "write" mode as here illustrated) users should also be made to attend to that distinction.

The "ideal" system would to start with "refresh" the screen when changing "mode". This would avoid the confusion the user encounters when nonrelevant helpmessages are still visible on the screen. It would also be advisable to invoke some active help to make the users attentive to the necessary distinction. The actual method to draw attention to the relevant facts has still to be determined.

b. evaluation.
The user will evaluate the result achieved with reference to his goal. It is thus essential that the system gives some information about the result. One of the main difficulties in the UNIX mail system is that it normally operated in "silent mode". It does not give any indication at all about the outcome of a certain command once it has been given by the user. This absence of feedback was responsible for most of the problems encountered by our example user.

An "ideal" system should strive for as much feedback as possible to novice users (and maybe less to experienced users). A good example in this respect is the COM system, developed at QZ, Stockholm Computer Center, where all messages, once they have been sent are accompanied by the information: "Message sent, No. XXX".

A question for research is how much feedback should be given when and to whom. We here encounter the problem of trade-off: A great amount of feedback will compete with the user's attention when other aspects are concerned. The user's planning of the next action may for instance be hampered if he has to wait for the feedback to appear and has to process it before he can continue. Frequent users of the same system may not need much feedback at all.

c. memory.
When new and unfamiliar procedures are presented, people will have difficulties memorising them. Users will differ in terms of what procedures that are unfamiliar to them, and will thus need different reminders.

The "ideal" system should thus have the possibility to tailor the information about the system (metacommunication) to the needs of the particular user. The knowledge the user brings to the particular situation can be assessed in advance by some simple questions. This assessment should cover not only information about whether or not the user knows this particular system, but also information about what other systems the user is familiar with. It has been found that both negative and positive transfer results between different systems (Waern, 1986).

There are some different ways of helping memory: giving concrete, procedural reminders and trying to fit the procedures into an overall structural model. We do not know what way is best, but suppose that different kinds of users may need different kinds of memory helps.

d. interpretation.
Users have to make sense of what they themselves and the system are doing, both in order to ease the working memory load and in order to plan future actions. We here encounter the big problem of what "models of the system" users create, and how these models may differ from the most adequate model in the particular situation.

In this particular project we have found that it is no trivial task to define even an "adequate model" of the UNIX mail system, even less to define the different conceptions which users might create from the metacommunications and interactions with the system. An "ideal mail" must therefore specify the objects and actions which are concerned in the particular system.

Next step concerns how to specify these objects and actions so that prospective users will interpret them in the intended way. This is a question which has educational implications, and will be covered by further studies within this project.

e. generalisation.
Whereas interpretation refers to particular procedures to be performed to achieve a particular goal, generalisation implies that some rules can be found which can generate different procedures for different circumstances.

In the UNIX mail study we found that one particular difficulty was related to the inconsistency of the commands used to invoke the "read" and the "send" mode respectively. Whereas the command "mail" would be expected to refer to an invocation of the mail system as such (as in DEC-type systems), in UNIX (and other types of mail systems as well) this command leads the user directly to the "read" mode. To write, the user has to give the command "mail", followed by a user-identity. This command structure is not very consistent and contributes to keeping the user to the procedural type of knowledge, where the difference between "read" and "write" modes is not clearly understood.

An ideal system is built on consistent rules to help the user with his generalisation attempts. A careful consideration of the functions which the system should perform can help system designers to create a consistent "grammar" (S. Payne, 1985). Starting at the functional level, further developments of the systems can then be made by specifying the corresponding command structures and the interaction rules (Moran, 1981).

2.5 The consequences for systematic experimentation.

Most of this analysis is based on introspection and observation of a small number of people trying to use the functionality for an artificial set of tasks. This is sufficient for establishing the problem areas that naive users are likely to have. Among the many questions that arise we chose the following as focal points to start with:

- are these problems general for all computer naive users
- to what extent can these problems be alleviated by alternative user interface design (mnemonics and making the system less silent).
- are these problems a consequence of lack of functionality with respect to the users perceived needs.

3. A PILOT EXPERIMENT WITH AN ELECTRONIC MAIL SYSTEM

3.1. Objectives

The objectives of this experiment were:

a. to establish a multi-site basis for future large scale experiments based on Stockholm, Amsterdam and Leicester.

b. to design and run an experiment with a multi-disciplinary team of cognitive and computer scientists with the prime objective of finding out to what extent naive users problems with unix mail can be alleviated by alternative user interface design. The main null hypothesis is that **different mnemonics and making the system less silent will not result in a different system from the users point of view.**

c. to systematically explore within the context of the null hypothesis:
   1. representations that naive users have
   2. the task performance of naive users
   3. the relationship (if any) between user performance and representations used and the spatial ability of users.
   4. the perceived mental load and mood of users when using the mail system.

3.2. Experimental Details

A simple design was adopted based on controlling for order effects due to users learning two different systems - the standard system (UMS) and the system with an alternative user interface (UMAI). All users in the experiment were selected to be naive to electronic mail. Subjects are randomly assigned to one of two groups depending on the order of learning: UMS first or UMAI first.

	time →		
Session No:	1	2	3
	training phase A	trial B	trial C
group I	UMS	UMS	UMAI
group II	UMAI	UMAI	UMS

The training phase is to teach users the functionality which is invariant for the following trials of the two systems. Equal numbers of subjects were used in each group. The experiment for each subject consisted of:

a. introductory information, collection of details about, for example, prior computing experience.
b. completing a mood and mental load questionnaire
c. written instruction about the system
d. training phase of guided and free exploration, terminated with a timed task completing a second mood and mental load questionnaire relating to d
f. repeating c-e with a new task inserted after c (trial B)
g. repeating c-e on the alterative interface, with a new task inserted after c (trial C)
h. spatial ability test

In total the experiment took over 4 hours, including coffee breaks between the sessions. The interaction between the subjects and the experimenter was carefully controlled with respect to giving help to users.

The following data were collected in the experiment:

a. questionnaires on mood and mental load, scored on a ranked scale
b. a time stamped log made during all the mail sessions from which objective task performance time and accuracy figures could be derived.
c. a "teach back" questionnaire relating to the users representation of the systems used and knowledge learned of using the systems.

3.3. Unix mail (UMS): Table of restricted command set considered

Some commands in UMS may appear in different notation, although the syntax is in principle equivalent. For the experiments we chose one of the alternative notations only to teach the users.

Command	**Function**
d	mark current message for future deletion, move onto next message now
w {<file>}	append current message without heading to <file> now mark current message for future deletion move onto next message now
s {<file>}	append current message as is to <file> now mark current message for future deletion move onto next message now
x	exit mail without updating mailbox
q	quit mail after updating mailbox by removing marked messages
?	give help now return to current message
+	if not last message move onto next message now else do x
-	if not at first message go back to previous message now
p	print the current message now
m {<user>} {<user>}	forward the current message now to <user> mark for future deletion move onto next message now

<user> is a valid user, default is the sender of the mail.
<file> is a valid filename or defaults to "mbox" in the user directory.

Command prompts	**possible user response**
?	All commands valid
$	UNIX prompt on leaving mail process means all UNIX commands valid

Command responses :

1. to any unrecognised command	"usage" followed by help information
2. to forwarding problems (m)	message that mail cannot be sent, then saves file in "dead.letter"
3. to file problems (w,s)	message that mail cannot be written to file

3.4 The Unix mail alternative interface (UMAI):

In order to test the null hypothesis, a version of unix mail had to be designed and developed which differed only in the user interface characteristics. A number of possibilities were explored around a well defined functionality. These were:

a. 1 character commands
b. 2 character commands
c. Full commands
d. Metaphorically based mnemonics
e. Semantically based mnemonics (hybrid of a to e)

The semantically based commands were chosen for the experiment.

3.4.1. derivation of semantically based mnemonics

The single commands of the standard UNIX mail as described above do not correspond to primitive operations. They can be organised into semantic groups and listed as follows:

operations on current message	navigation through mail	mail session control	Metacommunication
print now on screen	goto next message now	abort session now	give help now
mark for future deletion	goto last message now	leave mail application now & update mailbox	
append to <file-as-is>/ <file-as-text> now			
forward current message now			

This simple semantic analysis shows there are two themes.

Spatial: involving objects and places (files, messages) and associated operations (path traversing)

Temporal: events that happen immediately after invocation, or in the future and associated operations, (marking/signaling, ending a mail session)

Mnemonics should enable these themes to be easily organised and recalled with respect to operations an objects.

a. Temporal theme:
Default semantics for users to learn are that all commands are executed immediately (direct commands) unless they are special temporally marked commands (indirect commands). Some commands are both direct and indirect. The mnemonic principle is that commands that are both direct and indirect always contain a capital letter, else they are always lower case letters.

b. Spatial theme:
Default semantics are that after operations on the current message, the next message becomes the current message. i.e no path is specified by the user. When the current message is the last message in the mailbox, any current message operation (or going to the next message) is followed by a normal exit from the mailbox by the system i.e. the system makes a path from the last message to outside the mail session.

Paths must be specified for certain commands and these are organised so that the syntax of the message specifies the path (e.g. append current message to file is "a <filename>").

3.4.2. semantically based command set (UMAI)

Using the mnemonic principles and the results of a simple semantic analysis outlined in the previous section, a set of mnemonics was developed.

Mnemonic command	Function
operations on current message	
s(how)	print the current message now
D(elete)	mark current message for future deletion, move onto next message now
p(ut)D(elete) {<file>}	append current message without heading to <file> now

	mark current message for future deletion move onto next message now
p(ut)t(itle)D(elete) {<file>}	append current message as is to <file> now mark current message for future deletion move onto next message now
m(ail)D(elete) {<user>} {<user>}	forward the current message now to <user> mark for future deletion move onto next message now

<user> is a valid user, default is the sender of the mail.
<file> is a valid filename or defaults to "mbox" in the user directory.

navigation through mailbox

n(ext)	if not last message move onto next message now else do x
b(ack)	if not at first message go back to previous message now

mail session control

q(uit)	quit mail without updating mailbox
q(uit)C(hange)	quit mail after updating mailbox by removing marked messages

metacommunication

h(elp)	give help now return to current message
Command prompts	**possible user response**
Command:	All commands valid
:	Input next line of message to be sent, or end-of-file symbol
You left mail	
$	UNIX prompt on leaving mail process means all UNIX commands valid

Command responses :
same as for UMS

3.5. implementation

The controllability of the experiment and the need for multi-site implementation requires that the unix mail (UMS) and its alternative (UMAI) are simulated. Such simulations also serve secondary goals of achieving consistency (some versions of UMS have minor differences in functionality) and ensuring that the functionality is well understood and formally represented. It is necessary that the UNIX environment is well defined and controlled in the experiments as well as the mail systems themselves.

An extended BNF grammar was used to formally represent the interface components of both UMAI and UMS in a UNIX environment for the purpose of simulation. Simulations were developed in FORTRAN, LISP, PROLOG, C and SYNICS (a user interface manager language). Some simple evaluations were made of these on various criteria: portability, reliability, response times, etc.

A detailed guide for experimenters was developed and standard instructions for users in 3 languages

were used. Analysis programs are being developed as necessary for use on each site. However, the analysis of the user's representation and mental model of the system is an issue we are still working on.

4. RESULTS

So far the general achievements of the project resulted in a basis for multi-disciplinary multisite working, and a consistent representational framework within which to describe users models of the task and the system.

On the subject of user interfaces for electronic mail systems, the first series of experiments has been partially completed. At the time of writing we are analysing the data from two out of the three experimental sites. Other aspects are to be reported and further details are available from the project coordinator.

4.1. Site 1: Stockholm

In the following analysis, the factor experience denotes the number of sessions that subjects have completed in the experiments. Thus subjects are inexperienced before session 1 and (relatively) experienced after session 3. The second factor in the experiment is the order in which the two mail interfaces have been studied. The following discussion and results, based on 20 subjects (most of them university students), are preliminary only. We applied an analysis of variance technique, from which (with one exception) only statistically significant results ($p < .05$) will be reported.

4.1.1. Subjective data

a. mental load
   mental load generally decreases as experience increases, but it always is reported to be well above the base rating (before the introduction to the first system). There is an interactive turn over effect between sessions 2 and 3, depending on the order of the systems: when going from UMS to UMAI there appears to be a greater mental effort perceived by the user than when going from UMAI to UMS. A possible explanation is that applying UMAI requires a greater effort because the mnemonics are more complicated than the UMS. If it is assumed that the mental effort is mainly based on the mnemonics rather than the conceptual understanding of the system (both systems are semantically equivalent), this would give rise to an interaction effect such as that observed in the data.

b. mood
   - rating on dimension Happy-Sad
   We only found a significant interaction effect which is difficult to interpret. It seems that subjects who started with the UMS system get more happy when they learn it, whereas the subjects who started with the UMAI system get less happy during learning.
   - rating of confidence
   The main effect of experience is significant, the subjects were not too confident to start with, but grew more confident as the experiment went on. The interaction effect is only significant at 10% level, which leads to the speculation that subjects who started with the UMAI system got more confident, even when they turned to the UMS system, whereas subjects who started with the UMS system only got more confident within that system. This would be expected if the UMAI system made subjects confident about the functionality the systems have in common, by

providing more consistency in command set than in the case of UMS.
- rating of relaxation
At the first task, the subjects were a little more tense than to start with, a tension that quickly disappeared in the second session. But for subjects going form UMS to UMAI the effort of learning the new mnemonics clearly made them less relaxed. The other subjects going from the UMAI to UMS became more relaxed. This could be because the mnemonics for UMS are easy once the functionality of the system has been well understood by prior use of UMAI. The significant interaction indicates that the introduction to UMAI made the subjects feel more tense, which agrees well with the result presented above regarding the ratings of mental load.

4.1.2. Objective data.

a. task performance
We find that the tasks were pretty difficult to perform, with only about half of the subjects performing them correctly to start with. However the subjects easily learn, which is shown by a significant effect of experience on performance. Even when the subjects changed system in session 3, the performance either stayed constant or increased.

b. mental model representations
The subjects' free explanations of how to perform five different tasks (assumed to be an indication of their mental model) were rated concerning correctness and completeness as a whole. Correctness of the subjects' representations was on the lower half of the scale at the beginning, but the subjects were very close to correct at the end, even when they shifted systems, correlating with the results from the performance measurement. In the same way, completeness in explanations increased significantly with experience.

c. questionnair performance
In the second part of the questionnaire, the questions were divided into questions which primarily touched the semantic aspect of the system and questions which primarily concerned the syntactic aspect. Only the amount of experience is significantly positively related to the number of correct answers for both types of questions.

4.2. Site 2:Amsterdam

The analysis of the data collected in Amsterdam is not finished at the moment of writing. First of all we focussed on the representations users develop of a new application. In the second place we investigated the relation between this and individual differences in spatial ability. The "teach back" situation in Amsterdam was much more free than it was in the Stockholm experiment. So we applied a descriptive analysis, based on 22 subjects from a comparable population as those in Stockholm. Most of these had finished secondary education and were university students or graduates, some were in the final phase of secondary school. The experimental factors were again experience and order of mail interfaces to be studied.

4.2.1. Description of teach back results

After all three phases of the experiment we asked our users to explain, to somebody who did not know any computer system, how to delegate certain tasks to the electronic mail system. Subjects were allowed to apply all written means, either graphical, verbal, or mixed.

a. levels of description

We scored the representations by level of description (Moran, 1981). About half of the descriptions explicitely mentioned some notions of task delegation from user to computer, nearly all incorporated notions both on the level of semantics and of syntax, and 3/4 of the protocols contained some description at keystroke level. We found some extreme ways of representation, e.g. one subject only produced lists of keystrokes (including some indications of loops) although there were a lot of incorrect sequences in the representations. Another presented a complete and correct verbal description of task delegation to the system, of semantic structure, and of corresponding syntax, already after session 1.

b. structure of representation
About half of the users structured their knowledge as lists of actions (often not completely correct, as some kind of loops or diadic choices had to be included in the interactions to be described). One of every 4 users represented his knowledge as a set of (verbally described) production rules. Another quarter of the subjects gave more or less complete algorithms, sometimes applying graphical schedules or even full grown flowcharts.

c. quality of representation
Both the correctness and the completeness of representation in the teach back procedure increase with the order of the sessions. There was no clear relation with the order of the systems studied.

d. spatial ability
In the teach back data this factor seemed to be related with the correctness of description, not with the completeness or with any other characteristic of the representations.

4.3. Overall summary of experimental results.

The preliminary analysis of some of the Stockholm and Amsterdam data gave some idea of the greater mental effort and tension reported by the users in dealing with the mnemonics of UMAI, accompanied with a greater confidence (about the knowledge of the functionality, we assume), compared with the application of the traditional UNIX mail commands.

In general the mental load decreases with experience, whereas confidence and relaxation increase. Overall, both interfaces (with identical functionality) could be learned in a couple of hours, which was evident from task performance, from answers to questions about syntax and on semantics, and from representations of the system produced by the users.

The levels and structure of representation showed a lot of individual differences. Spatial ability seems to be related to correctness of free representation.

5. Conclusion and Discussion:

Thus far we have only made some exercises regarding the four basic questions asked in the introduction to this contribution.

a. It is possible to describe the user interface of an existing application in a complete and consistent way

b. The conceptual model of a new user interface may be defined from an analysis of the semantics of an existing application.

c. From this model an actual interface may be constructed, in a way that supports portability and experimentation.

d. A methodology has been developed to do feasibility studies on alternative user interfaces.

Regarding the specific questions we posed in 2.5 we are only in the pilot phase of our study. The two alternative interfaces we compared did not make too much systematic differences in user behaviour or subjective experiences. On the other hand we collected experiences about representations that naive users have, about their perceptions of mental load and mood, and about their performances for this kind of tasks.

For the future we established a multi site base for large scale experiments, and developed a methodology of multi-diciplinary cooperation. The impact of a more user oriented functionality for the design of application interfaces, combined with notions on metacommunication still to be worked out, will be the focal point of the next studies.

REFERENCES

Breuker J. & de Greef P. (1985) Functional Specification. Teaching and Coaching Strategies for EUROPHELP. 4.2 ESPRIT Project 280. Memorandum 49 of the Research Project: The Acquisition of Expertise.

Hannemyr G. (1983) Human factors standards. The design of conceptual language interfaces to open computer network application and management systems. Behaviour & Information Technology, 2, p. 335-344.

Hannemyr G. & Innocent P.R. (1985) A network user interface: Incorporating human factors guidelines into the ISO standard for open systems interconnection. Behaviour & Information Technology, 4, p. 309-326.

Moran T.P. (1981) The command language grammar : a representation for the user interface of interactive computer systems. International Journal of Man-Machine Studies, 15, p. 3-50.

Muylwijk B. van, Veer G.C. van der & Waern Y. (1983) On the implications of the user variability in open systems. An overview of the little we know and of the lot we have to find out. Behaviour & Information Technology, 2, p. 313-326.

Payne S.J. (1985) Task action grammars: the mental representation of task languages in human computer interaction. Unpublished PhD thesis, University of Sheffield.

Veer G.C. van der, Tauber M.J., Waern Y. & Muylwijk B. van (1985) On the interaction between system and user characteristics. Behaviour & Information Technology, 4, p. 289-308.

Veer G.C. van der, Guest S., Haselager P., Innocent P., Lammers E., McDaid E., Oestreicher L., Tauber M.J., Vos U.& Waern Y. (1987) Human factors in telematic systems - progress report for the COST-11-ter working group. Free University, Amsterdam.

Wheeler T. & Innocent P. (1983) Human factors in and requirements of the OSI environment. Behaviour & Information Technology, 2, p. 335-344.

Waern Y. (1986) Understanding learning problems in computer aided tasks. In: F.Klix and H. Wandke (eds.). Man-Computer interaction research Macinter-I. North-Holland, Amsterdam.

A SunView User-Interface for Authoring and Accessing a Medical Knowledge Base

Neil P. Groundwater

Sun Microsystems, Inc.
8219 Leesburg Pike, # 700
Vienna, Virginia USA 22180

Dr. Neil Bodick
Andre Marquis
Department of Pathology and Lab Medicine
2 Gibson Building
Hospital of the University of Pennsylvania
3400 Spruce Street
Philadelphia, Pennsylvania USA 19104

1. INTRODUCTION

A system named *Eidetic* has been built for the storage, retrieval, and examination of microscope slides. The foundation of the system is a Sun Workstation running the SunView window system with application programs built at the Hospital of the University of Pennsylvania in Philadelphia in cooperation with Sun Microsystems, Inc., and Elsevier Science Publishers.

The term *Eidetic* has been chosen for the system because as an adjective it is employed by cognitive psychologists to characterize the memory of images or the process of visualization. The complete system with image capture and display inspires the user to closely associate vivid color images with the process of medical diagnosis. *Eidetic* is intended to be used much like a reference journal, increasing the capacity of a physician to analyze relevant data.

Two major subsystems make up *Eidetic*: an authoring system and a searching system. After an author collects images and composes a knowledge base both get transferred to Compact Disk Read Only Memory (CD-ROM). Searchers can analyze the stored data to draw conclusions and assist their diagnostic process. *Eidetic* supplies "power tools" to the physician or medical student.

This discussion will cover the overall design of the system with the intention of highlighting the components which are affected by *SunView* programming techniques. To acquaint the reader with the goal of the delivered system, sections will also cover the world-view of the users, an overview of the knowledge base and its use, describe the authoring and user systems, and outline the organization of the CD-ROM storage. This is not intended to be an introduction to *SunView* programming therefore only pertinent details will be discussed. For the reader's convenience a short glossary of *SunView* terms is attached at the end of this paper.

2. WORLD VIEW

The users of *Eidetic* will generally be physicians or students in hospitals or medical schools. Although they may have been trained on other computers, our general assumption is that there has been little past experience and there may be no computer reference material present. The appearance to the user must never become too cumbersome to comprehend; while advanced use of the system does involve significant processing of the stored data, the user is encouraged to "explore" without fear of "getting stuck". *Eidetic* requires almost no keyboard input by the end-user and presents an interface in which only the appropriate choices and controls are visible at any moment. The mouse is used for almost all input and even the authoring system only uses the keyboard for the collection of the medical and clinical terms which comprise the vocabulary.

With the power of a workstation computer at hand, the user can expect quick response, both searching the database and retrieving images to the screen, but at the same time the system is tailored for utility rather than to overwhelm the user with flashy visual displays. There is the capability to

display 256 simultaneous colors, however that capability is **not** used to flash buttons and menus in gaudy colors just to impress and confuse the viewer. Color is only used when displaying the images and their annotations.

3. OVERVIEW

A physician's personal knowledge base tends to organize itself around the history of cases that the doctor has either treated or studied in the medical literature. Clinical categorization of case histories is the foundation of the *Eidetic* system in a three-tiered hierarchy:

Category The category is the highest level of information specific to a case. It divides a case into approximately twenty clusters of information ranging from patient descriptors to detailed clinical terms. In the former instance, *patient descriptors* such as *age* will probably not refer directly to images but in the latter, specific terms may well be reinforced by images.

Feature The elaboration of a *category* results in its *features*. To expand on the category above, *patient descriptors* including *age* and *sex* are features and each can be assigned a specific *grade*.

Grade At this level of case-documentation an image may be attached to illustrate a grade or more accurately, the degree to which a characteristic is visible. The author can draw an overlay on an image and directly connect that overlay to one or more *feature-grades*.

Other than the notion of the above levels of case-documentation, the end user is not restricted to a particular view of the data. *Eidetic* is a system which does much to encourage **exploration**. As is generally expected of visual-interface programs, the user, whether physician or student, can vary the flow and choose to:

- Show a case when given patient descriptors and clinical descriptors match.
- Show images where the diagnosis was malignant and the cell-groupings were in clusters.
- Given cases where both xx and yy were present, show the tendency for zz.

Usage is not restricted to a directed query-response formula but rather a "seek-and-ye-shall-find" hunt for pertinent information.

4. SUNVIEW

The *SunView* library includes routines to display *panel items* in a *panel* subwindow and respond to the user's selection and interaction by means of the mouse or the keyboard. The level at which the application responds to the user is based on *events* and can be developed with an *object-oriented* style. While the advantage of an object-oriented style is beyond the scope of this paper, it reduces the dependence on global variables and the programming anomalies introduced when one must program around special cases because there are a large number of exceptional situations. One feature of *SunView* that enables an object-oriented approach is the use of *handles* to *SunView* objects and their *attribute value lists*. Together they give a uniform way to access and manipulate the items on the display without undue reliance on user-maintained state. Another feature is the PANEL_CLIENT_VALUE parameter which may be specified when declaring a *SunView* object. It takes an argument of type *caddr_t* which may point to a variable, data-structure instance, function-name, etc. Once given a handle to an object, one may recall and access the PANEL_CLIENT_VALUE supplied parameter.

Although *SunView* provides many features as part of its generic behavior, most of the features may be enhanced or overridden if different responses are desired. Because of the target users for *Eidetic*, the standard Sun-supplied behaviors were in several instances deemed to be too complex for the casual user. Historically, sales of Sun Workstations have been to a technical market where the users are accustomed to a high-powered and sophisticated UNIX system and their window-based tools have had an equally sophisticated interaction-level. While third-party software developers for many of today's workstations have been prone to entirely replace the UNIX interface and manufacturer's "look and feel" by defining their own model "from the ground up", with the release of *SunView* accompanying SunOs Release 3.0 the was made to use and adapt the "standard" interface.

Some of the behaviors provided by *SunView* were appropriate just as supplied, some were either eliminated or simplified, some new *panel items* were built from existing parts, and some changes may be made in the future. The primary notion that *buttons* can be "pressed" has been kept; however, the appearance has been altered to display an image or icon as a button which visually implies the action one may expect when activated:

- An arrow-button indicates the "flow" of the interaction and pressing on the button may generate a pop-up menu.
- A button made of a color picture will display the microscope slide with that image, generally with annotation in the form of an overlay and text.

4.1. Traditional Features

Within the scope of standard *SunView* features there is tremendous flexibility to define the displayed user interface without needing to create complex *event procedures* to handle interaction. Perhaps the most useful of these techniques is the ability to supply images, both 1-bit and 8-bit, to be used as a PANEL_CHOICE_IMAGE. A sample set of buttons and the associated image for one of them are shown in Figure 1. In the user system, 1-bit (black and white) arrow-buttons are used to indicate flow of control and the 8-bit (color) buttons are used as selectors for the display of a case's images.

4.2. Custom Features

Several of the interactions with *Eidetic* require the user to choose among a list of medical terms. One technique for dealing with this is the use of pop-up menus which may be intimidating for inexperienced or casual users. When the appearance of menu requires a conscious action on the part of the user - namely pressing a mouse button - it should be strongly indicated that that operation is appropriate through the appearance of a particular *panel_item.*

Most of the choose-from-a-list operations in *Eidetic* appear instead as a *scrolling list* much like the Macintosh *SFGetFile* function provides in *Inside Macintosh*, the Apple reference manual. Macintosh users are most accustomed to seeing this item appear when they choose *Open* in the *File* menu in a Macintosh application. The code included in Figure 2 is the basis for such a device and its appearance is shown in Figure 3. This is a PANEL_CHOICE with the selected item highlighted by setting PANEL_FEEDBACK to PANEL_INVERTED, which causes the whole-text of the entry to invert to white-on-black. For the inexperienced or casual user the full range of selections may be reviewed by means of the scrollbar attached to the window. For advanced users the PANEL_CHOICE offers a menu when the right mouse-button is pressed. These items are dynamically constructed when a new category or feature is chosen; a linked-list in the database provides the values for the choices. When a selection is picked from the list, the PANEL_VALUE returns the offset of the selection and the PANEL_CLIENT_DATA points to the information related to that selection.

An additional function is related to the visual continuity when a pop-up menu is used to make a selection. If the scrolling list is used in the normal manner, the selection is scrolled into the visible region of the list by the user before being picked. If the pop-up menu is used, there is no internal feedback to normalize the list and make the choice visible because the scrollbar doesn't intrinsically "know" whether the choice is visible. In order to preserve fidelity when the menu is used, the *event procedure* compares the positional value of the choice against the value of the first visible entry and number of entries visible. If the selection is off-screen then *scrollbar_scroll_to* is called to reposition the scroll bar and redisplay the adjusted entries.

4.3. Advanced Features

A combination of the above functions was produced to provide a list of image-based choices. One of the advanced analysis features of *Eidetic* measures the interrelation of case-features based on a comparison composed by the physician-user. From the information retrieved from the database a histogram is produced to illustrate the correlation of two of the cases' *feature-grades.* Such a histogram is illustrated in Figure 4. Given the histogram the user may choose to review the cases which produced the statistics by selecting the relevant bar with the mouse and pressing the *List Cases* or *List Images*

button.

4.4. Unused Features

There are several *SunView* features that are notable in their absence. A few of the exclusions and their reasons follow:

[1] There are no *frame-menus* on the *Eidetic* tools like those which appear with standard *SunTools*; *Eidetic* intended for exclusive use on the workstation. Although there is no technical reason for restricting the use of *Eidetic* while other *SunTool* (UNIX) windows are in use, the current scheme begins when the user logs onto the workstation and completely controls the selections of tools and windows which may be used.

[2] The middle mouse-button doesn't allow the movement or resizing of windows or sub-windows because the screen "real estate" is used in a manner which mixes *tiled* and *overlapped* windows. The choice of windows present is designed to maximize the utility of the tools; at no time would it be opportune to "move a window out of the way" to expose another window. Rather, once the screen is filled with windows the user has reached a point where something should be "put away" before proceeding; the end of a particular investigative path has been reached.

[3] In general, only the left mouse-button is used. The normal *SunView* interface defines that the left-button is *select*, the center-button is *adjust*, and the right-button shows a *menu*, which might be difficult for the inexperienced or casual user. The only time the *Eidetic* user system requires the use of the other buttons is when scrolling through a list of choices and this is actually considered undesirable because there is some difficulty in explaining **how** scrollbars are operated. For the current system it was expedient to use the scrollbars as provided by *SunView* rather than invent new ones. As development of more *Eidetic* tools proceed, some other scrolling paradigms may be investigated.

5. AUTHORING SYSTEM

Creation of the knowledge base is performed on a Sun workstation which is coupled to a television camera mounted on a microscope. The author of a knowledge base for a particular field of medicine is an expert in that field. The two fields which have been documented to date are cytology and dermatology.

Four major phases are involved in creating a knowledge base:

[1] The descriptive terms for the field of interest are entered into a *thesaurus tool*. New terms can be added during the collection of images, however the discipline of a restricted vocabulary improves the ability to present a useful search strategy.

[2] The author(s) collect a set of microscope slides which covers the field of interest.

[3] The slides are previewed, captured, and stored on magnetic disk. Concurrently, the overlays are composed and associated with images and *feature-grades*. During this phase the author's expertise determines the quality of the final product since the annotation and overlays attached will appear in the final knowledge base.

[4] A completed knowledge base is pre-mastered into a form appropriate for producing a Compact Disk and sent out for production. The specifics of the disk layout will be presented later in an overview of the CD format.

5.1. Thesaurus Generation

A relatively simple *SunView* program helps the expert enter the vocabulary of terms which will be used to create the knowledge base and document the annotation drawn on the microscope slide images. This vocabulary is stored in a hierarchical database descending from category to feature to grade. The structure of the vocabulary is integrated into the knowledge base such that the information for each case parallels the structure of the thesaurus and both may be maintained in memory for fast search and retrieval.

It is the concept of fully qualifying the *grades* for each *feature* that allows the expert/author to ensure the level of accuracy that may be later used in searching. Given a new case, each feature will be marked *not reported* until the author fills in all of the details. Even "no information" may be entered since for each *feature-grade* there is a value reserved for *unknown*.

5.2. Image Collection

A Sun 3/160 is connected to an Ikegami *ITC 350M* video camera mounted on a Leitz *Laborlux 12* microscope and the RGB image from the camera is transmitted to three Matrox *MIP 512* boards in the Sun Workstation. The image capturing routines compress the 24-bit color image to 8-bit color and present the result on the screen in the authoring tool. Compression of the image-data consists of error-diffusion of each channel followed by colormap assignment and takes about 100 seconds; a few of the 256 colormap entries are reserved for the windows, their borders, and the overlays.

For image control, only an intensity adjustment is supplied by the software; its value varies the sensitivity of the Matrox boards to dim or brighten the image captured. Focus, magnification, and illumination are adjusted at the microscope. Once the image appears satisfactory on the screen it is "confirmed", stored on the magnetic disk, and made available for annotation. Since the microscope slides are stored at a fixed resolution, if there is an interest in examining a site more closely it is the author's responsibility to store it in the database at multiple magnifications: 100x, 250x and 1000x.

5.3. Drawing Tools

The annotation may be performed with three basic styles of paint-tools: The first resembles the MacPaint "paintbrush" type and just lays down paint while tracking the mouse, a second is for outlining and draw a square, a circle, or a line which "rubber-bands" to follow the mouse until the button is released. A third set of tools are arrow-stencils which deposit an arrow with its point at the mouse and its shaft trailing away in one of the four cardinal directions depending on which arrow-tool was chosen. The tools are a simplified rendition of those provided by MacPaint and the "paint" appears on the microscope slide in a color which contrasts with the colors of the slide and appears to flow "on top of" the image itself.

The mechanism by which paint is applied to the image involves the *canvas* subwindow-type. The drawing is performed via *pw_stencil* operations on the *canvas pixwin* and consist of capturing the *mouse-down* event in the canvas. At the same time that the paint is stenciled onto the canvas an off-screen *pixrect* is retained of the previous annotation in case the next operation is *undo*. Once the overlay is completed, an *Attach Overlay* button is activated to connect the overlay with the currently selected *feature-grade*.

In the painting *canvas* there is some advanced capability attributed to the middle and right mouse buttons. Although optional, it allows the use of the middle button to act like an eraser and the right button to undo the last bit of overlay that was drawn. Since the user of the authoring system is trained and not a "casual user", the increased capability speeds the annotation process.

6. USER SYSTEM

The user system presents the most simplified of the *Eidetic* applications by beginning with a near-blank screen containing just a few icons to choose among the analysis subsystems. Figure 5 illustrates one such window after some user responses have occurred and the subframes have all appeared. The gray arrows along the right side of the figure show the current course of input and it is clear that the boxes (*subframes*) have appeared from the top to the bottom. Once again the scrolling list *panel items* are present and the large arrows are all *panel buttons*. As the search proceeds the user can choose to call up images or cases based on the results of each inquiry.

7. CD-ROM Organization

Compact disks have been selected as the distribution media for *Eidetic* because of the low cost of reproduction and the static nature of the data. CD-ROM had some impact on the storage layout and retrieval techniques involved: the volume of data, the directory scheme for the disk, minimizing disk-

head-seek times, and the interface to the workstation.

7.1. Volume of Data

A survey determined that a typical field of medicine can be significantly documented by images from between 100 and 400 cases and the calculations which follow will show that the CD-ROM capacity is a good match. The images come from a television camera video output and provide 512 by 480 8-bit pixels after data compression, thus an image requires approximately 250K bytes. For other storage technologies it might be attractive to compress the data (a two-to-one reduction could easily be expected), but then the retrieval time would have to include decompressing the image too, and would affect the system's response times dramatically. Each case provides from three to eight images and each image has one to seven overlays. The above numbers result in about 700 images and 3000 overlays for a disk covering 125 cases and that will only occupy about forty percent of a compact disk. None of these numbers have an upper bound in the on-disk directory scheme, these particular numbers just represent two clinical fields thus far documented.

7.2. CD-ROM Directory

The CD layout is shown in Figure 6 and begins with a label and a copyright notice in clear-text. Following that, a simple directory on the disk needs only eight bytes per image (case number, image number, and disk-offset). Image retrieval would suffer considerably if all referrals commenced with a return to track zero (or other fixed location) for every search so the directory is read into memory only once and indexing is always fast. Once the image is located an *lseek* and *read* of the raw disk can transfer the image into the target *canvas*.

In addition to the directory, images, and overlays, the CD-ROM also stores "postage-stamp" iconic images which are used as color buttons representing the full-size images. Some of those images are visible near the top of Figure 1. During the user-search, when a case is "opened" on the screen, these icons appear to show all of the slides attributed to the case. The compression-scheme to create the reduced image is basically one of discarding-seven-of-eight-bits and it requires only four kilobytes on disk per icon.

In order to "lead" the user to select the "correct" image for inspection, the icon which illustrates a *feature-grade* can be outlined to indicate that it is "interesting".

7.3. Access Times

Although not specifically pointed to by the directory, an image's icon offset on disk can be trivially computed once the associated entry is found in the directory. All of the icons for a given case are stored contiguously on the disk so that they can all be quickly loaded when a case is opened up on the screen.

After an image is brought to the screen, the disk-head is in position to read the directory of associated overlays since they immediately follow it on the disk. In contrast to the images, the overlays are very suitable for compression since they represent a one-bit-deep image which likely illustrates a couple of arrows pointing out image features. Without compression, that data would require thirty-two kilobytes but with simple run-length compression the overlays usually are no longer than three to five kilobytes. As a nice side effect, the standard Sun Rasterfile access routine, *pr_load*, reads using *stdio* so when *stdio* is used to read in the overlay-directory the first one or two overlays are transferred to memory "for free" because of the *stdio* eight kilobyte internal buffer size.

The access techniques used for the CD-ROM are traditional UNIX techniques with the restriction that all accesses must be on a 2K-byte boundary. Thus a *read* is always preceded by an *lseek* to the appropriate starting point. It is known that the buffers for *stdio* accesses will always read an even multiple of 2K which then allows *stdio* to be used with only the proviso that *fseek's* are done to 2K boundaries too. Since there are no other restrictions on access it is convenient for testing or demonstration that a magnetic-disk file containing a small database in the CD-ROM style can be used with no software modifications.

7.4. The Sun Workstations

As described above, the authoring station is a Sun 3/160 with video frame-grabber boards added to collect images. The users' workstation is a Sun 3/110 connected to a CD-ROM drive through the SCSI bus which extends to disk and tape peripherals. The software driver for the CD player was adapted from software Sun Microsystems had already developed to support write-once (WORM) drives. Both the authoring and user workstations are standard offerings with appropriate hardware added for this application, so there is a long-term advantage that these systems may also be used as traditional workstations and *Eidetic* need not require dedicated resources.

8. FUTURE

The *Eidetic* hardware suite includes a frame grabber and sufficient computer power to allow the end-user to collect and annotate images "in the field". While there is a premium on read-write media of sufficient capacity to allow long term collection and storage of images, it is presumed that write-once or some other technology will deliver the means for the physician or student to build up their own image collection and knowledge base. In addition, there are several areas of image processing which can be provided to allow clinical research to extend the capabilities of the *Eidetic* system.

9. SUMMARY

Eidetic has been built for both composing and retrieving clinical information and microscope images. The system uses physicians' ability to interpret microscope images and connects that interpretation to a clinical knowledge base that may be examined via a mouse-based interface. *SunView* has been utilized to build the software components and their user interface with standard *panel items*, with *scrolling lists*, and with *PANEL_BUTTON_IMAGEs*.

Emphasis in the design of the man-machine interface is on decreased user training time and encourages the user to deepen the search by continuing to associate new references. There is no "conclusion" to the search; the final analysis is always subject to interpretation by the physician/user. A staged appearance of windows and their interface-objects leads the user through the analysis process.

SunView Glossary

PANEL_CLIENT_DATA

A pointer of type *caddr_t* to a user supplied data item. In the C tradition it may be any storage type or a routine name.

PANEL_NOTIFY_PROC

The procedure called when the item is selected.

PANEL_EVENT_PROC

If a behavior is not provided by default event processing a new behavior may be supplied. *Eidetic* uses these to alter the standard mouse-button handling to simplify and constrain the user interface.

PANEL_FEEDBACK

The feedback given when a choice-item is selected. One of: PANEL_INVERT, PANEL_NONE, PANEL_MARKED, etc.

PANEL_LAYOUT, PANEL_VERTICAL or PANEL_HORIZONTAL

"Relative" positioning policies. Positions can be enumerated and based on a parameter such as font-size so the layout dynamically adjusts if the default-font is changed.

PANEL_SHOW_ITEM, TRUE or FALSE

The buttons and text-items may be composed in advance and then dynamically shown when a particular mode or subsystem is activated.

PANEL_VALUE

When used with PANEL_CHOICEs it sets or retrieves the value (ordinal number) of the current selection.

References

Sun Microsystems:

System Interface Manual for the Sun Workstation
The SunView Programmers' Manual
The SunView System Programmers' Manual

Apple Computer:

Inside Macintosh

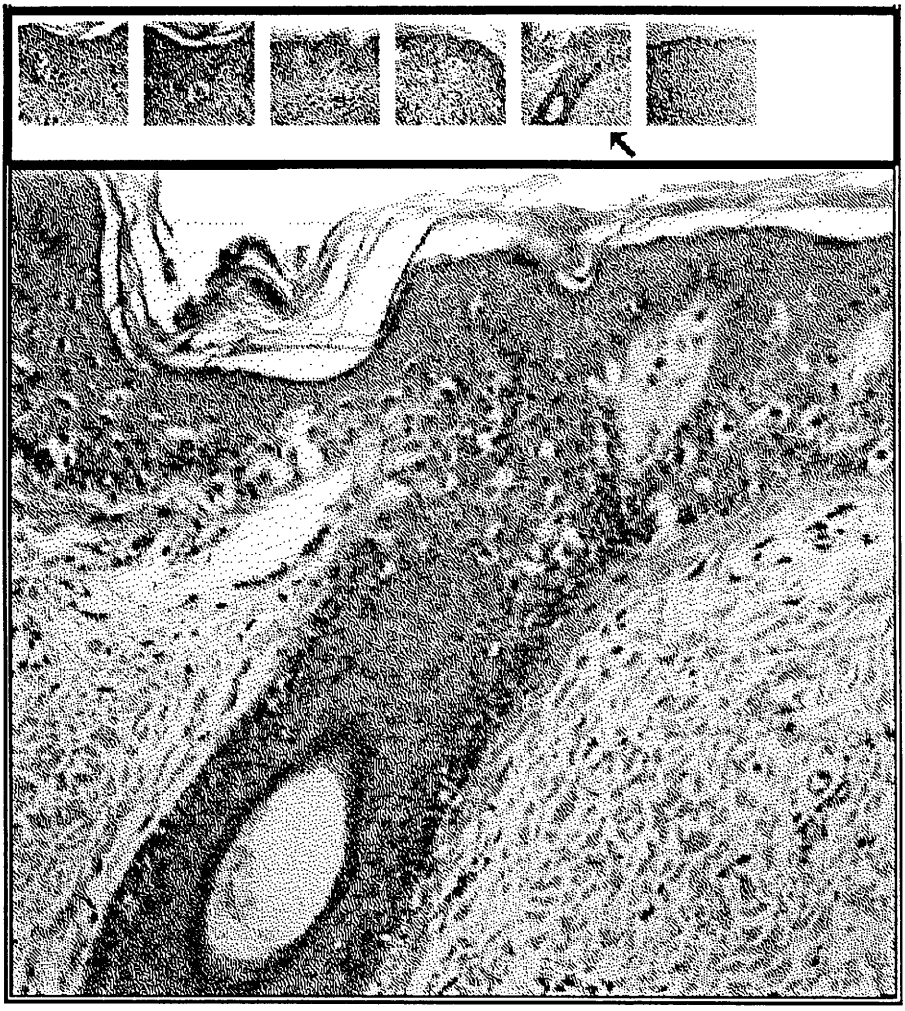


Figure 1. Panel with button-images (top) and an Eidetic image in a canvas subwindow (bottom)

```
static Panel_item
create_cat_list_item(cat_ptr, catnum)
     struct CATEGORYSTRUCT *cat_ptr; /* linked list of entries */
     int                   catnum;       /* entry # to be highlighted */
{
     int                   item_num, position = 1;
     Panel_item            cat_item;

     /* set it up to show the selection in reverse video */
     cat_item = panel_create_item(cat_panel, PANEL_CHOICE,
          PANEL_FEEDBACK,              PANEL_INVERTED,
          PANEL_NOTIFY_PROC,           cat_list_notify_proc,
          PANEL_CLIENT_DATA,           NULL,
          PANEL_SHOW_ITEM,             FALSE,
          0);

     /* put the strings into the "list" */
     for (item_num = 0; cat_ptr != NULL; item_num++) {
          panel_set(cat_item,
               /* index and string for choice */
               PANEL_CHOICE_STRING,   item_num, cat_ptr->categoryname,
               PANEL_LAYOUT,          PANEL_VERTICAL,
               PANEL_CHOICE_Y,        item_num, position,
               0);

          if (item_num == catnum)     /* the "current" selection */
               panel_set(cat_item,
                    PANEL_CLIENT_DATA, cat_ptr,
                    0);
          position += font_height;
          cat_ptr = cat_ptr->nextcategory;
     }

     /* make it appear */
     panel_set(cat_item,
          PANEL_VALUE,                 catnum,   /* highlight it */
          PANEL_SHOW_ITEM,             TRUE,     /* become visible */
          0);

     return (cat_item);
}
```

Figure 2. Scrolling-list sample code

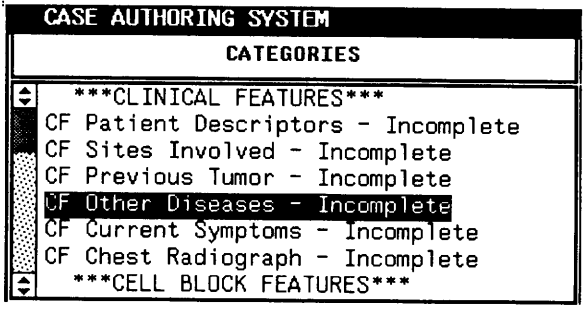


Figure 3. Scrolling list

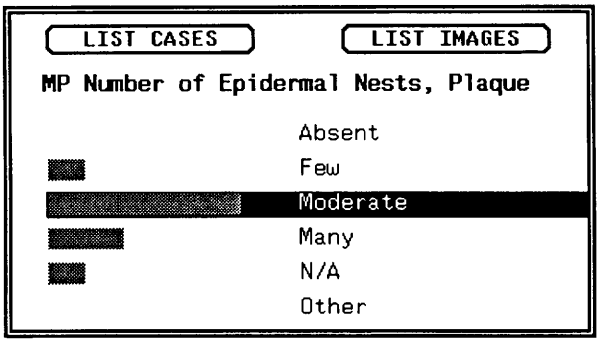


Figure 4. Histogram list made of PANEL_CHOICE_IMAGEs

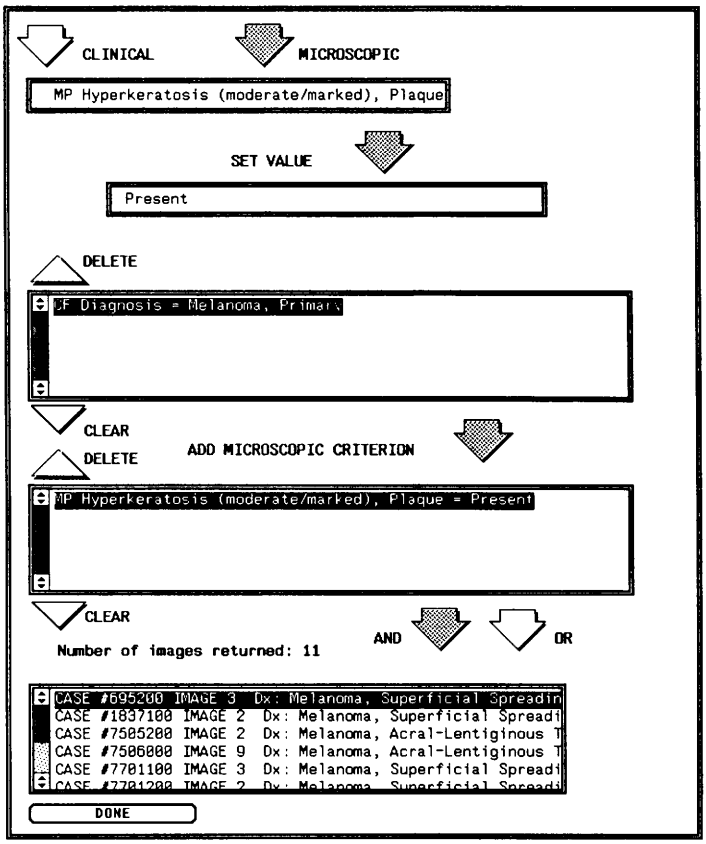


Figure 5. User system with highlighted arrow-buttons

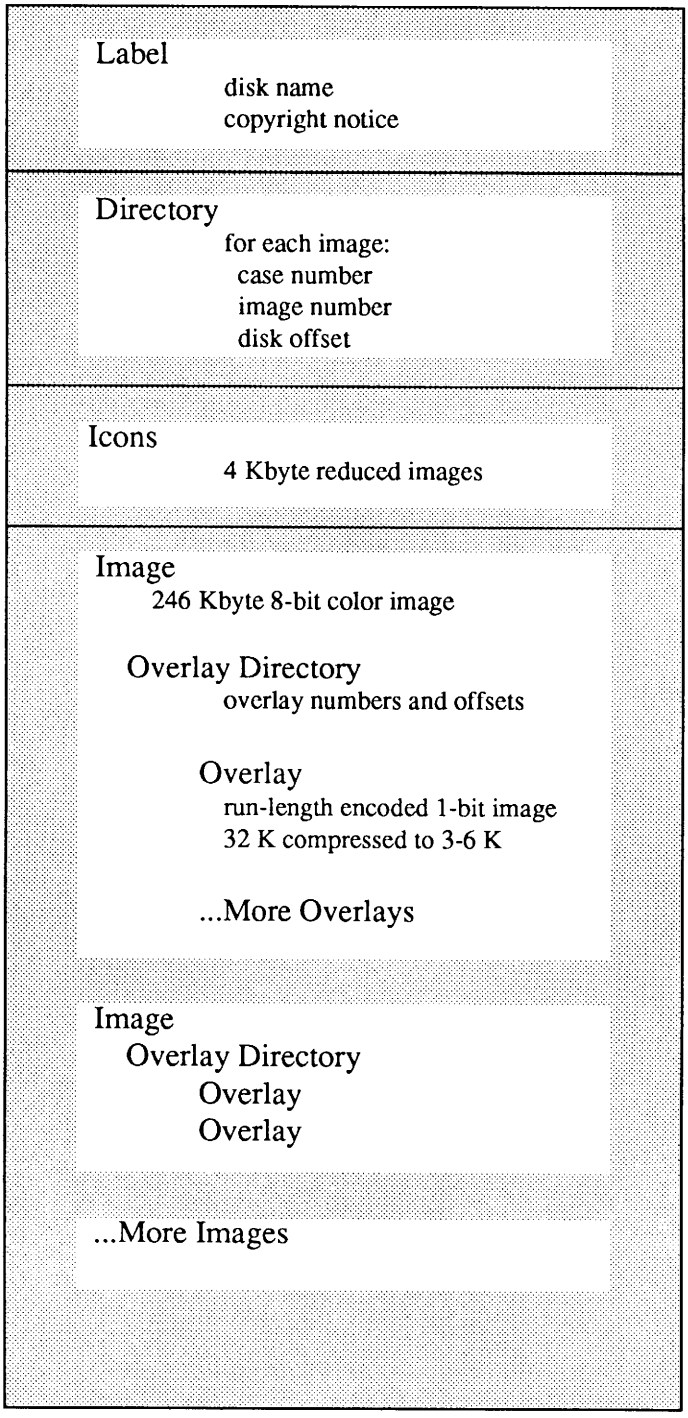


Figure 6. CD-ROM organization

The Analysis and Manipulation of BNF Definitions

Allan C. Milne

Dept. of Mathematics & Computer Studies
Dundee College of Technology
Bell Street
Dundee
Scotland

ABSTRACT

An extended BNF notation (EBNF) is formally defined and its use described. An associated software tool called LL1 provides analysis and manipulation functions for systems defined using this notation. The facilities provided by LL1 are particularly useful in the development of LL(1) definitions. This paper will also discuss the use of EBNF and LL1 in the development of language syntax and will consider some implications of using EBNF representations of Jackson structure diagrams.

1. Introduction

Backus Normal Form (or Backus-Naur Form) (BNF) is a widely used notation for the definition of programming language syntax. Although other notations such as syntax diagrams are also used, none match the formality, consistency and completeness of BNF. A BNF definition implicitly defines a grammar, the theory of which developed from work in linguistics and which has now matured into a well-understood, rigorous mathematical discipline. The theorems and axioms of grammar theory (see[Aho72,Back79]) can be directly applied to systems defined in BNF.

It is felt that the theory of grammars and the use of BNF have wider applications than that of defining the syntax (or structure) of programming languages. Most applications have an implied "language" in their human-computer dialogue, an appropriate and useful specification tool for this would be BNF. The structure of other notations which include recursive and iterative constructions may also be represented in BNF, one example of such s notation is the Jackson structure diagram.

Unfortunately, there is no a formal definition of the BNF notation and the original BNF does suffer from certain limitations. Many extensions have been proposed but these also have not been formalised and are often inconsistent. To encourage a wider use of BNF there must be a formal

definition of a useful extension together with appropriate tools to allow automatic manipulation and validation of specifications.

The extended BNF notation (EBNF) described in this paper is formally defined in itself and allows great flexibility in the style and structure of system specifications. The notation allows both recursive and iterative constructions and a distinction can be made between major structural components and the "micro-syntax" of more elementary entities.

The LL1 tool provides analysis and manipulation functions for systems defined using this EBNF notation. The facilities provided are designed particularly for processing LL(1) specifications although they can also be usefully applied to more general definitions. Analyses provided by LL1 include validating the correctness and consistency of a specification, computation of LL(1) director sets and a full cross-reference. Manipulations include transformation into BNF, factoring, substitution and removal of left recursion.

The application area addressed by EBNF and LL1 is the specification of system structure. This paper will describe their use in the design of language and dialogue syntax and will consider some implications of using EBNF specifications of Jackson structure diagrams.

2. BNF and Extended BNF

2.1. Terminology of BNF

The theoretical underpinning of BNF is the theory of grammars (or language theory). A grammar is a tuple (N,T,S,P)

N : A finite set of non-terminal symbols.

T : A finite set of terminal symbols.

S : The distinguished or starter symbol. This is a designated non-terminal from the set N.

P : A finite set of productions, each mapping a non-terminal to a string of terminal and/or non-terminal symbols.

Terminals are the basic symbols which constitute the entity being defined, for example the digits "0" through "9" and "." might be the terminals for a real number. Non-terminals are introduced by the designer to represent structural components of a system. The distinguished symbol non-terminal is the highest level component representing the system as a whole. A production of the form

```
non-terminal -> string
```

defines the string of terminals and/or non-terminals which can be derived from the non-terminal. Starting from the distinguished symbol, productions are repeatedly applied until a string of terminals is reached.

The notation BNF was first used to describe a grammar in the definition of the language Algol-60[Back59,Naur63]. A grammar is defined in BNF by giving a series of rules for the non-terminals of the grammar. These rules are made up of one or more productions. Each non-terminal must be defined by a rule of the following form:

```
non-terminal  ::=  production-list
```

The meta-symbol "::=" is read as 'can derive'. A production-list is a series of one or more productions seperated by the "|" ('or') meta-symbol.

For example, a binary number may be defined as

```
<bin-no>     ::= <bin-digit> <rest-no>
<rest-no>    ::= <bin-no> | <>
<bin-digit>  ::= 0 | 1
```

The meta-symbol "<>" is used here to represent the null production which derives the null string. The symbols '0' and '1' are the terminals.

This BNF specification defines the grammar (N,T,S,P) where

```
N = {<bin-no>,<rest-no>,<bin-digit>}

T = {0,1}

S = <bin-no>

p = { <bin-no>     ->  <bin-digit> <rest-no>
      <rest-no>    ->  <bin-no>
      <rest-no>    ->  <>
      <bin-digit>  ->  0
      <bin-digit>  ->  1  }
```

One very useful type of grammar is the LL(1) grammar in which rules must exhibit certain characteristics. These characteristics are concerned with parsing which is the process of determining if, and how, a specific string of terminal symbols can be derived from the distinguished symbol. A string which can be so derived is thus a valid sequence of terminals meeting the specification. For a rule to be LL(1) it must either have only one production or the choice of which alternative production to apply must be uniquely determined by the next terminal symbol to be processed, i.e. there is no requirement to look ahead. A grammar is LL(1) if all the rules are LL(1).

An example of an LL(1) rule is

```
<filename> ::= <letter> <rest-name>  |  . <rest-name>
```

A non-LL(1) rule is

```
<response> ::= yes please  |  yes thank you
```

2.2. The Extended BNF Notation

The extended BNF notation (EBNF) defined here is based on BNF. A formal definition of EBNF (in EBNF) is given in Appendix 1 to this paper.

Within EBNF there are three distinct types of symbols

Meta-symbols which are symbols of the EBNF notation itself.

Non-terminals which are bracketed by angle-brackets (<...>).

Terminals which are any other character strings.

Non-terminal symbols are bracketed by "<" and ">" as for BNF and must not contain any meta-symbols. Terminal symbols are strings of characters delimited by a meta-symbol, space, end-of-line or end-of-file. If a terminal is to include a meta-symbol or space then the terminal must be enclosed in quotation marks. The apostrophe is used as the escape character within such a terminal stribg to include quotes or apostrophes. Thus "'''" is the terminal '".

Rules in EBNF are of the same form as previously described for BNF but must be terminated by a period ".". The first rule of an EBNF specification defines the distinguished symbol of the grammar.

While in BNF a production is simply a string of terminals and/or non-terminals, a production in EBNF is a string of clauses of the following form:

```
symbol                  : a terminal or non-terminal
{production-list}       : production-list may optionally appear
{production-list}*      : production-list may appear 0 or more times
[production-list]       : used to bracket alternative productions
[production-list]*      : production-list may appear 1 or more times
```

N.B. the second and third clausal forms above are enclosed in curly brackets.

Using the extended clausal forms the binary number defined earlier in BNF might be alternatively defined as any of the following in EBNF

```
<bin-no> ::= <bin-digit> {<bin-digit>}* .

<bin-no> ::= [<bin-digit>]* .

<bin-no> ::= [0 | 1]* .
```

Clauses may be nested to any depth as in the following specification of a real number:

```
<real-no> ::= [<digit>]*  |  (<digit>)* ("." (<digit>)* ) .
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .
```

Another example is of a UNIX-type command:

```
<command> ::= <name> ( -<option> (<specifier>))*
                    (<parameter>)* .
```

The notation is free-format except for end-of-line delimiting terminals and non-terminals. Comments may be included in a specification anywhere that a rule or clause can occur. The start of a comment is indicated by a question mark "?" and it continues to the end-of-line.

BNF is a proper subset of EBNF with the exception that "." must be used to terminate a rule.

2.3. Macro- and Micro-syntax

One of the innovations of EBNF is the distinction which can be made between the macro-syntax and the micro-syntax of a specification. Rules occurring first in a system specification are taken to define the macro-syntax. Rules which define the micro-syntax must appear at the end of the EBNF definition and are seperated from the macro-syntax by the meta-symbol "%microsyntax%".

Macro-syntax rules define the main structural components of a system. These rules must be complete and formally define the constructs which represent the primary processing requirements. A system may be entirely defined within the macro-syntax. Formal definition of a complete system can be tedious and result in an over-detailed specification. The micro-syntax of EBNF allows more intuitive, descriptive or less formal definitions to be made. The rules within the micro-syntax may define entities which are elementary, pre-defined or generally accepted and which do not require a full rigorous specification.

A simple example of the use of this concept can be found in the definition of EBNF given in Appendix 1. To avoid listing all possible characters which can be derived from <char-string> and <any-char-string> these non-terminals are defined descriptively in the micro-syntax.

In the context of programming language design the macro and micro components might relate directly to processing phases within the compiler. The macro-syntax may define the major syntactic constructs which are processed by the syntax analyser or parser. The micro-syntax may define the tokens which are processed by the lexical analyser.

This concept allows the system designer flexibility in developing

specifications. The system definition can be built up in a stepwise manner, gradually moving definitions from the micro to the macro-syntax as the design detail emerges and can be formally defined. Another implication of this distinction is that it opens up the possibility of differentially processing rules. An example of this can be found in the LL1 tool where rules in the micro-syntax play no part in transformations, in fact LL1 treats non-terminals defined by these rules as terminals as far as its processing is concerned.

2.4. Advantages of EBNF

Using BNF to specify systems or parts of systems has certain attractions. BNF is a reasonably concise formal specification technique which has been used in programming language design for over 20 years. The theory of grammars which underpins BNF is well-understood and there are many useful results which might be applied to specifications.

One of the major disadvantages of BNF is that the designer is restricted to using recursive definitions. EBNF allows iterative constructions through its extensions to the clausal forms within productions. The designer is now free to choose an iterative form of definition which may be the most natural and which can have implications on the efficiency of implementation. It is important not to constrain the implementation through lack of flexibility in the design tool used.

The arguments of efficiency and flexibility in design and implementation also support the other functional extensions in EBNF. The distinction between macro- and micro-syntax can be used to hide detail from the main design and allows the specification to be complete in structure without being complete in detail. A stepwise refinement approach can then be used to develop the level of detail required.

Other advantages of EBNF relate to its increased utility as a specification document. The formal definition of EBNF ensures there can be no ambiguities in the form of the notation, while the variety of constructions available allows flexibility in the form of the specification. The free-format layout and provision of comments hopefully make a specification easier to write, understand and maintain.

3. The LL1 Software Tool

3.1. Introduction

The original motivation for LL1 came from work in designing the syntax of programming languages. The syntax specification should be correctly formed, consistent and complete. It may also be desirable that it exhibit certain specific characteristics, such as being LL(1). Manual validation of these criteria is certainly non-trivial and may be unrealistic. An automatic mechanism is needed to perform such validation and to guarantee that validity. A model for such an automatic tool can be found in SID [Fost68]. The functions of SID are similar to those of LL1 but while SID was innovative in 1968 there have been many advances since then which now

limit its usefulness. LL1[Miln86] is an attempt to revise, extend and structure both the facilities provided and the implementation.

The overall objective of LL1 is to provide a tool to assist in the development of grammars (especially LL(1) grammars). The grammars are to be defined using the EBNF notation described earlier. The operation of LL1 is to be flexible with the user controlling all display, analysis and manipulation functions.

The current implementation of LL1 is in the language S-Algol[Cole82,Morr79] which is a high-level orthogonal language developed at the University of St. Andrews. The language has very powerful program and data structuring facilities including support for list-processing. LL1 is heavily reliant on the list-processing environment since the two main data structures representing the grammar and the data dictionary are multi-linked list structures. The intention is to implement any future version in C for greater portability.

3.2. Functional Description

The main functional phases of LL1 are shown in Figure 1. All functions of LL1 are under the control of the user through a list of options supplied when invoking the tool. The EBNF notation is used as the definition language for the input grammar to allow flexibility in the specification and to ensure there are no artificial constraints. In the descriptions that follow all analyses and transformations are with respect to rules in the macro-syntax. Non-terminals defined in the micro-syntax are treated by LL1 as terminals as far as the defined grammar is concerned. The input grammar is checked for correctness and completeness of rule construction and non-terminal definition.

The theory of grammars relates to recursive rather than iterative definitions and therefore during input any iterative (EBNF) clause is transformed into an equivalent recursive (BNF) definition as follows :

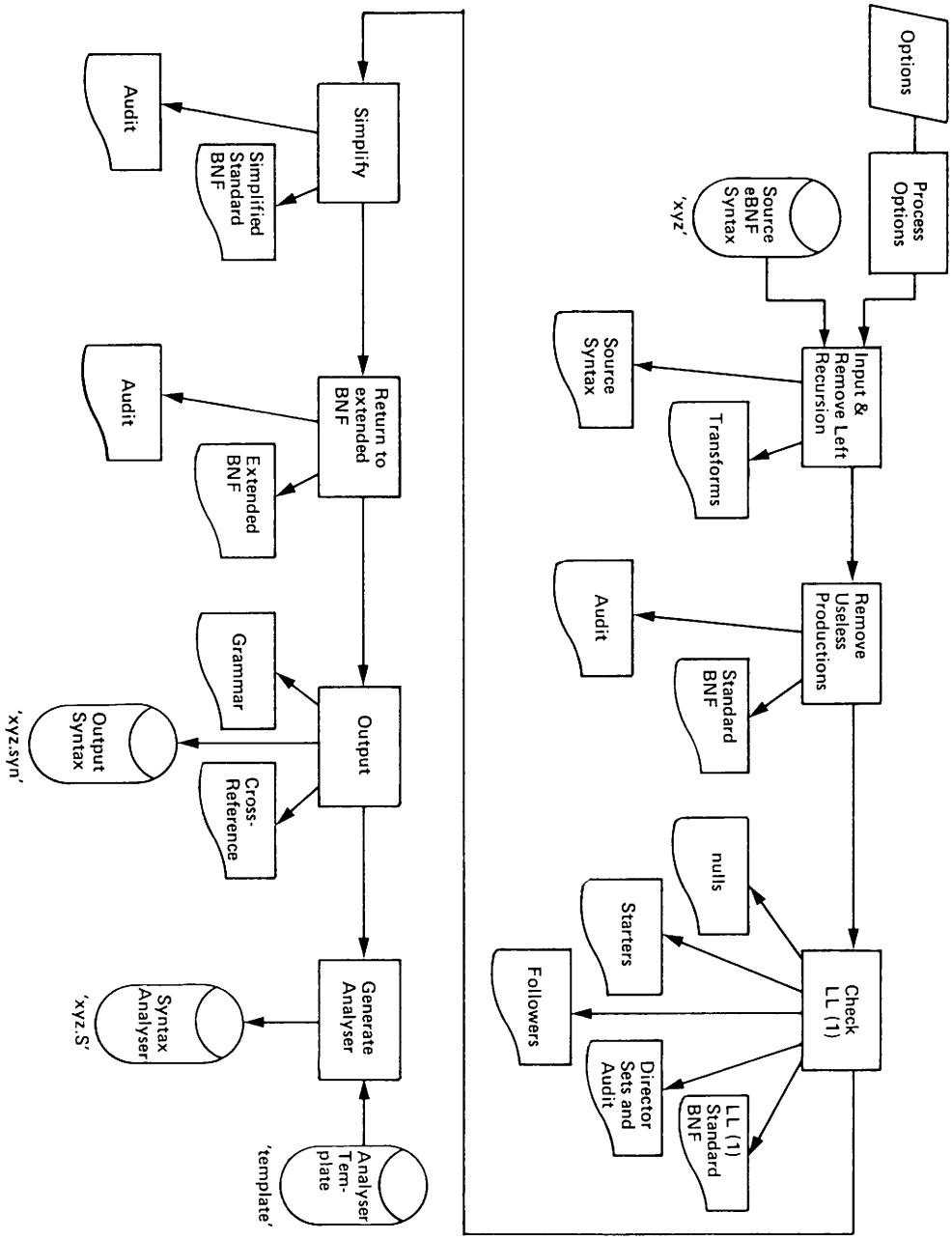
```
<a> ::= ...{s}... .          =>   <a> ::= ...<$$Option-1>... .
                                  <$$Option-1> ::= s | <> .

<a> ::= ...{s}*... .         =>   <a> ::= ...<$$Option-1>... .
                                  <$$Option-1> ::= s <$$Option-1>
                                                 | <> .

<a> ::= ...[s]... .          =>   <a> ::= ...<$$Group-1>... .
                                  <$$Group-1> ::= s .

<a> ::= ...[s]*... .         =>   <a> ::= ...<$$Group-1> <$$List-1>... .
                                  <$$Group-1> ::= s .
                                  <$$List-1> ::= <$$Group-1> <$$List-1>
                                               | <> .
```

Figure 1


Options
Process Options
Source eBNF Syntax
'xyz'
Input & Remove Left Recursion
Source Syntax
Transforms
Remove Useless Productions
Audit
Standard BNF
Check LL (1)
nulls
Starters
Followers
Director Sets and Audit
LL (1) Standard BNF
Generate Analyser
Analyser Template
'template'
Syntax Analyser
'xyz.S'
Output
Grammar
Output Syntax
'xyz.syn'
Cross-Reference
Return to extended BNF
Audit
Extended BNF
Simplify
Audit
Simplified Standard BNF

In the above, and in the following transformations, <$$Option-1>, <$$Group-1> and <$$List-1> are new non-terminals generated by LL1; "s" and "t" are arbitrarily complex production lists. Also during this input phase any direct left-recursion in non-terminal definitions may be removed by a rule transformation of the form

```
<a> ::= <a> s | t |... .        =>      <a> ::= <$$Group-1> <$$Option-1> .
                                        <$$Group-1> ::= t | ... .
                                        <$$Option-1> ::= s <$$Option-1> | <> .
```

Using these generated non-terminal names allows this to be transformed by a later phase of LL1 into the EBNF form

```
<a> ::= [t|...] {s}* .
```

A production is useless if it either cannot be reached from the distinguished symbol or it cannot (eventually) derive a string of terminal symbols. Such useless productions are normally the result of an error in specification and their occurrence can be reported and the offending productions removed from the grammar.

The grammar may be checked to determine if it satisfies the LL(1) condition. If it does not then it is possible to apply certain transformations in an attempt to arrive at an equivalent LL(1) specification. This is described in more detail in the next section.

Some elementary simplification of the grammar can be made at this stage to remove non-terminals which have only a single production and are referenced only once. The final grammar can then be transformed back into an EBNF representation. This is done in a manner which ensures that the final grammar is as close to the original as possible, taking into account any transformations which have been applied. The EBNF transformations are applied only to productions which include a generated non-terminal (<$$Group-1> etc.) and are exactly the reverse of those described for transforming from EBNF to BNF. The grammar can be output in the EBNF notation.

During LL(1) processing information is computed which defines the terminal symbols determining the application of alternative productions of a rule. This information can be used to generate a recursive-descent syntax analyser (parser) which will recognise terminal strings meeting the specification. The procedure used is described in the next section.

As can be seen from Figure 1 various analyses and audit listings are provided. One of the most useful analyses is a cross-reference report which, in effect, is a data dictionary for the system specification, it lists all terminals and non-terminals with references to their definitions and occurrences.

3.3. LL(1) Processing

The motivation for specifying a system as an LL(1) grammar lies in the ease of recognising valid terminal strings. Given a terminal string, at any stage in the derivation sequence the cortrect production of a rule to apply is defined uniquely by the next terminal in the input stream. Thus the input stream is processed one terminal at a time with no requirement for look-ahead or backtracking in the recogniser procedure. Consider the rules

```
<response> ::= <yes> <conditional>  |  <no> <conditional> .
<yes>      ::= yes .
<no>       ::= no .
```

The rule for <response> is LL(1) since the presence of 'yes' or 'no' as the next input terminal defines the production to apply. If a terminal other than these is present an invalid <response> has been made and an error procedure will be invoked.

In any realistic specification there will be a number of terminals which can validly define the application of each production. This set of terminals is called the director set[Grif76] of the production. It has been shown that a necessary and sufficient condition for a grammar to be LL(1) is that the director sets for each production of each rule be disjoint[Knut71,Grif76]. LL1 uses this result in its LL(1) determination. The director sets of each production are computed, this is not as straight-forard as it may appear since allowance must be made for null productions. An analysis of the director sets can be produced and this can be useful in manually validating the correctness of a specification. The structure of the LL(1) determination algorithm follows that proposed by Griffiths[Grif76], certain details borrow from Backhouse[Back79] and Warshall[Wars62].

Given that a specification is found not to be LL(1) (i.e. it has intersecting director sets) it is possible to attempt to transform it into an equivalent LL(1) specification. This cannot be guaranteed to succeed (this is theoretically unsolvable) and if LL1 detects looping or cannot apply further transformations it will terminate this phase for the rule in question. The transformations applied are factoring and substitution. Factoring is applied as follows

```
        <response> ::= yes please  |  yes thank you .

=>      <response>  ::= yes <$$Group-1> .
        <$$Group-1> ::= please  |  thank you .
```

The resulting LL(1) rules can be transformed later into the EBNF form

```
        <response> ::= yes [please | thank you] .
```

Substitution is applying a non-terminal definition to a production in which

the non-terminal occurs. It is of the following form

```
<a> ::= <b> r  |  s .          =>      <a> ::= t r | u r | s .
<b> ::= t  |  u .                      <b> ::= t | u .
```

This does not by itself make a production LL(1) but it is often necessary before an appropriate factoring transformation can be applied. The order of application of these transformations is important and an appropriate application algorithm is used[Miln86].

With an LL(1) grammar the passive specification can be directly transformed into an active recogniser for processing strings of terminals (in compiler terminology a syntax analyser or parser). The technique used is recursive descent compiling (for details see[Davi81]) in which, using a small set of primitive procedures, recopgniser procedures are derived for each non-terminal in the specification. LL1 offers the facility to automatically generate a recursive descent parser from an LL(1) specification. The parser is in the language S-Algol and incorporates error handling and an elementary lexical analyser, to build up terminal tokens from character input. The generation algorithm used is designed to allow parsers in other languages to be generated. The transformations between EBNF and the generated code are defined in Appendix 2. The elements of the parser which are not specific to any particular grammar are stored in a template to which the recogniser procedures and other information is added.

4. Application to Language and Dialogue Design

The specification of programming language syntax was the original motivation for the development of EBNF and LL1, however most systems have a "language" implicit in the design of their human-computer interface. EBNF is an appropriate specification tool to use for this dialogue and is,, for example, itself the input "language" for LL1.

Using EBNF gives the designer flexibility in a formal specification and imposes no restrictions on the design, its features also make it self-contained for documentation purposes. LL1 can be used to verify the correctness of the specification in terms of missing definitions, spurious rules or general clerical errors. Thus a syntactically correct and complete specification is assured. A specification can be checked for correctness of meaning by using LL1 to provide analyses of attributes and applying transformations as appropriate. During all transformations made by LL1 full audit listings are is produced to ensure the designer is aware of any manipulations and to allow the design to be tuned as necessary.

If an LL(1) specification is used then LL1 can derive the corresponding recursive descent recogniser. The use of the recursive descent technique to process systems defined by an LL(1) EBNF specification has many advantages. Perhaps the most important is that a formal specification which can be validated at the design stage is automatically transformed into the corresponding processing structure. This structure is, in a sense, a proved implementation since it is directly derived from the formal design. The implementation is also elegant and easily understood with many

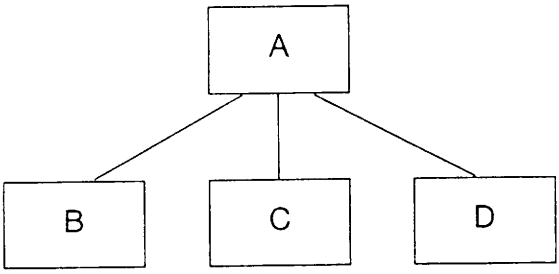
published algorithms for error handling[Fisc80,Grah75,Maun82,Turn74].

This method can be used as a prototyping tool to test the quality and effectiveness of the design. Using EBNF and LL1 will allow extensive testing to take place and make it relatively easy to modify the design and recycle through the verification and testing phases. It is hoped that the ease of use of these tools will also encourage the designer to investigate alternative designs.

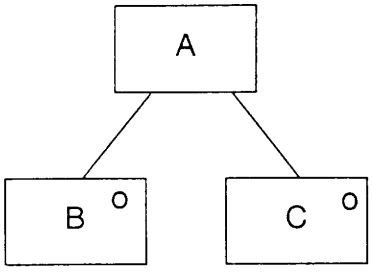
The objective for EBNF and LL1 in this application area is to give more formality, flexibility and efficiency in the design and implementation process.

5. Relevance of EBNF to Jackson Structure Diagrams

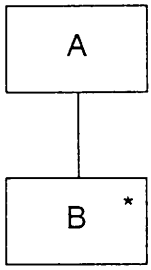
Jackson Structure Programming (JSP) is a popular system development methodology which is based on a graphical representation of data and process structure. It is noted that EBNF can be used to represent the various forms of structure diagram which can occur in JSP and preliminary work is being carried out on the relevance of this representation. The EBNF representations of the structures are as follows



<A> ::= <B> <C> <D> .



<A> ::= <B> | <C> .



<A> ::= (<B>)* .

If the structure diagram defines a data structure then the elementary JSP components are represented by a terminal which is the primary component of the data item (for example a character, digit, etc.).

If a process structure is being defined then the representation of the elementary components is more complicated and dependant on the standard used for labelling the components. A textual non-formal labelling standard may result in a component being represented by a non-terminal defined in the micro-syntax. A more formal labelling can give more power to the EBNF representation. If a unit is labelled "sum := a+b;" then an appropriate EBNF representation might be

<sum> ::= "sum := a + b;" .

The final program structure could then be derived from the EBNF representation through appropriate substitution of these elementary non-terminal definitions.

At the very least EBNF can be used as a formal "mathematical" representation of the graphical JSP diagrams. However the use of EBNF also opens up the prospect of applying the body of knowledge which exists about grammars to JSP diagrams, and of utilising automatic tools. LL1 might be used to verify that all JSP components have been defined and that there is no ambiguity. Also LL1 will produce a full cross-reference of component defintions and uses which can form the basis of a data dictionary which is known to be complete.

The relevance of some LL1 transformations to JSP diagrams is clear, for example the removal of useless productions will remove superfluous definitions from the structure. The meaning of a JSP representation being LL(1) is less clear, one avenue being explored arises from the phases in the JSP methodology

JSP data structure -> JSP process structure -> implementation

If a JSP data structure is represented by an EBNF specification then it can

transformed into LL(1) form. The processing of the data structure should then be less complex in that only one data component need be considered at any one time. The implementation from an LL(1) process specification might be automatic using the recursive descent technique. This offers all the advantages described in the previous section.

6. Summary

This paper has proposed EBNF as a formally defined specification notation which defines a grammar. This notation can be used to specify many different types of systems and allows the theory of grammars to be used to analyse and transform the specification. The EBNF standard is flexible, self-documenting and is one more tool in the toolkit of formal specification techniques.

LL1 is a software tool which can be used to manipulate and analyse specifications written in EBNF, it is especially designed for the production of LL(1) grammars. It provides an automatic mechanism foir validating specifications including the non-trivial task of LL(1) determination.

The use of EBNF and LL1 in language and dialogue design has been described and their relevance to JSP discussed. EBNF has wider applications than its traditional role in defining the syntax of programming languages and it is hoped that further work will be done in investigating its usefulness in other areas.

References

Aho72.
A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation and Compiling Volume 1 - Parsing, Prentice-Hall, 1972.

Back79.
R. C. Backhouse, Syntax of Programming Languages : Theory and Practice, Prentice-Hall International, 1979.

Back59.
J. Backus, "The Syntax & Semantics of the Proposed International Algorithmic Language of the Zurich ACM-GAMM Conference," Proc. Int. Conf. Information Processing, pp. 125-132, UNESCO, Paris, 1959.

Cole82.
A. J. Cole and R. Morrison, An Introduction to Programming with S-Algol, Cambridge University Press, 1982.

Davi81.
A. J. T. Davie and R. Morrison, Recursive Descent Compiling, Ellis-Horwood, 1981.

Fisc80.
C. N. Fischer, D. R. Milton, and S. B. Ouring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions," Acta Informatica, vol. 13, pp. 141-154, 1980.

Fost68.
J. M. Foster, "A Syntaxq Improving Program," Computer Journal, vol. 11, pp. 31-34, 1968.

Grah75.
S. L. Graham, "Practical Syntactic Error Recovery," CACM, vol. 18, no. 11, pp. 639-650, 1975.

Grif76.
M. Griffiths, "LL(1) Grammars and Analysers," in Compiler Construction - An Advanced Course (2nd. Ed.), ed. F. L. Bauer, J. Eickal, Lecture Notes in Computer Science, vol. 21, pp. 57-84, Springer Verlag, 1976.

Knut71.
D. E. Knuth, "Top-Down Syntax Analysis," Acta Informatica, vol. 1, pp. 79-110, 1971.

Maun82.
J. Mauney and C. N. Fischer, "A Forward Move Algorithm for LL and LR Parsers," Sigplan Notices, vol. 17, no. 6, pp. 79-87, 1982.

Miln86.
A. C. Milne, "LL1 - An LL(1) Grammar Analysis and Manipulation Tool," Dept. of Maths & Computer Studies Technical Bulletin 5, Dundee College of Technology, Dundee, Scotland, 1986.

Morr79.
R. Morrison, "S-Algol Reference Manual," Dept. of Computational Science Report CS/79/1, University of St. Andrews , St. Andrews, Scotland, 1979.

Naur63.
P. Naur and et al, "Revised Report on the Algorithmic Language Algol-60," CACM, vol. 6, no. 1, pp. 1-17, 1963.

Turn74.
D. Turner, "Error Diagnosis and Recovery in One-Pass Compilers," Information Processing Letters, vol. 6, no. 4, pp. 113-115, 1974.

Wars62.
S. Warshall, "A Theorem on Boolean Matrices," Journal of the ACM, vol. 19, no. 1, pp. 11-12, 1962.

```
?   Formal Definition of the Extended BNF Notation EBNF

<grammar> ::= <syntax-part> {<micro-syn-part>} .
<syntax-part> ::= [<rule>]* .          ? 1st rule defines the
                                       ? distinguished symbol
<micro-syn-part> ::= "%microsyntax%" [<rule>]* .

<rule> ::= <non-terminal> "::=" <production-list> "."  |
           <comment> .

<production-list> ::= <production> {"|" <production>}* .

<production> ::= [<clause>]* |
                 "<>" .                          ? null production

<clause> ::= <symbol> |
             <option-clause> <rep-symb> |        ? 0 or more times
             <choice-clause> <rep-symb> .        ? 1 or more times

<option-clause> ::= "{"<production-list>"}" .

<choice-clause> ::= "["<production-list>"]" .

<rep-symb> ::= "*" | <> .

<symbol> ::= <non-terminal> | <terminal> | <comment> .

<non-terminal> ::= "<"<char-string>">" .

<terminal> ::= <char-string> |              ? ends with meta-symbol,
                                            ? eol or space
               "'""<any-char-string>"'"".

<comment> ::= "?"<any-char-string>.         ? ends with eol

%microsyntax%

<char-string> ::= "any ASCII chars except meta-symbols" .

<any-char-string> ::= "any printable ASCII chars.".  ? use '" to include "
                                                     ? and '' to include '
```

Appendix 2 - Recursive Descent Recogniser Procedures

The following primitive procedures are used

mustbe.. - checks that a supplied terminal is the next input terminal token. If present the input stream is moved on, if not the error procedure is invoked.

is.. - indicates whether a specified terminal is next on the input stream.

For brevity the following abbreviations will be used

P	- a production
ds(P)	- tests for all members ofthe director set of P (using the procedure is..)
T	- a terminal
N	- a non-terminal name (not including "<" and ">")

Rules and Procedures

There is one procedure generated for each rule in the macro-syntax, the procedure name being the non-terminal name made into a valid identifier with ".." appended. For a rule with one production the generation is

```
<N> ::= P .                    =>    procedure N..
                                     begin  P  end
```

A production which consists of terminal and/or non-terminal symbols is handled as

```
N1 T N2                        =>    N1..
                                     mustbe..(T)
                                     N2..
```

Within a rule there may be alternative productions which are generated as

```
P1  |  P2                      =>    if ds(P1)
                                        then P1
                                        else P2

P1  |  P2 | ... |  Pn          =>    case true of
                                     ds(P1) : P1
                                     ds(P2) : P2
                                       ...
                                     default : Pn
```

EBNF clauses

```
{P}                    =>    if ds(P) do P
```

```
{P1 | P2 | ... }        =>      case true of
                                 ds(P1) : P1
                                 ds(P2) : P2
                                   ...
                                 default : {}

{P}*                    =>      while ds(P) do P

{P1 | P2 | ...}*        =>      { let end.loop.. := false
                                  while ~end.loop.. do
                                     case true of
                                     ds(P1) : P1
                                     ds(P2) : P2
                                       ...
                                     default : end.loop.. := true
                                }

[P]                     =>      same as for P

[P1 | P2]               =>      same as for P1 | P2

[P1 | P2 | ... | Pn]    =>      same as for P1 | P2 | ... | Pn

[P]*                    =>      repeat P while ds(P)

[P1 | ...]*             =>      { procedure group.. (cint time.. -> bool)
                                  begin
                                     ans.. := false
                                     case true of
                                     ds(P1) : P1
                                     ...
                                     default : { if time..=1
                                                    do syntax.error.
                                                 ans.. := true
                                               }
                                     ans..
                                  end
                                  let end.loop.. := group..(1)
                                  while ~end.loop.. do
                                     end.loop.. := group..(2)
                                }
```

Uncle - A Case Study in Constructing Tools for the PCTE*

Hans-Jürgen Kugler
Barry Lynch
Generics (Software) Limited
7, Leopardstown Office Park,
Foxrock Dublin 18
Ireland

Abstract

A set of common service interfaces has been proposed as basis for a portable common tool environment (PCTE). The PCTE favours a distributed workstation approach to software engineering environments and uses an object management system (OMS) based on an entity-relationship approach as fundamental information repository.

The PCTE defines functional interfaces allowing the creation and manipulation of complex data-models using the distributed OMS, thus facilitating the interpretation of development and management tools. The PCTE interfaces are currently based on Unix services and expressed in C.

UNCLE is a tool to allow the construction of command and response languages for the PCTE which are turned to the users' views of the underlying system and his or her skills. An object-oriented abstraction mechanism allows the construction of common tool interfaces.

INTRODUCTION

Portability and reusability are major issues in increasing productivity in system development. Portability requires the existence of standards in interfacing as well as a certain harmony in development paradigms. Industrial standards are being developed on a wide scale, especially in those areas which have to deal with interfacing to computers in the widest possible sense. Open System Interconnection, Graphical Kernel System, and a variety of programming languages, including Ada™ [1] as the latest example, are being standardised.

Uncle - a **U**ser's **N**ice **C**ommand **L**anguage **E**nvironment - is a realisation of the designs for user interfaces undertaken by working groups in the International Standards Organisation and the International Federation for Information Processing [2][3]. Uncle emphasises the portability of user interfaces and the need for adaptation to the user's view of a system. This

---

* Part of the work reported here is funded by the Commission of the European Community under the Multi-Annual Dataprocessing Programme, contract MAP 759.

™ Ada is a registered Trademark of the U.S. Government, AJPO.

implies that the interfaces of tools contained in the system are presented in a uniform way - and tailored to the user's understanding. To provide such functionality Uncle supports the concept of *public tool interfaces*.

The provision of a unified framework for tool interoperability was the concern driving the *PCTE* project in the European ESPRIT framework. PCTE is a basis for a **P**ortable **C**ommon **T**ool **E**nvironment [4], which is now protected as an industry standard.

PORTABLE COMMON TOOL ENVIRONMENT (PCTE)**

The PCTE project was launched in 1983 with the aim to create the technological base for the project partners, Bull, ICL, GEC, Nixdorf, Siemens and Olivetti, to construct modern software engineering environments. The SEEs would provide a set of tools and services to support the complete system development life cycle. These tools, however, are created by the SEE developer using services and function calls common to all PCTE systems. Thus the SEE is portable across all hardware supporting the PCTE.

The PCTE interfaces are a set of program-callable primitives which are machine-independent. They were published in document form early on in the project, so that the information would be disseminated throughout the rest of ESPRIT and indeed, beyond.

The preferred architecture for the PCTE is that of a Local Area Network (LAN) of powerful single-user workstations with a high resolution display and a pointing device. Account was also taken of the more conventional mainframe and mini computers which would be supported with or without LANs.

From the PCTE user's point of view, the structure of the network is completely transparent and no explicit communication is necessary between the user of one machine and machines elsewhere in the LAN.

The approach to portability taken by the design team led to the choice of UNIX being the first base system upon which PCTE would be built. The reasoning behind the decision was that more widespread availability would be achieved within a short time-frame because of the size of the Unix community. The increase in size of the X/Open [5] group, which includes all of the project partners, was also a factor.

Unix on its own is not a distributed system, so a distribution layer had to be built to provide a base for communicating nodes each running an independent Unix system. The logical link between the nodes is established by using a distributed **O**bject **M**anagement **S**ystem in place of the conventional Unix file system. The *OMS* provides the central mechanisms needed for tool interface, object and project activity definitions.

Currently the PCTE specification takes the form of a (large) set of C interface definitions, in the form recommended by X/Open. The long term objective of the project was to also produce a complete rewrite of the PCTE in

---

** Readers familiar with the PCTE may want to skip this section.

Ada on top of a minimal portability kernel. This would cater for the non-Unix operating systems. In the medium term an Ada interface definition was to be defined which would allow an Ada interface to the C definitions. This was seen as a necessary step, particularly if the PCTE was to be targeted to the US market.

The PCTE primitives cover three main areas:

1. Basic Mechanisms

   These deal with the manipulation of various entities found within an SEE as well as with program execution. These entities would be programs, documents, software tools and program data. Central to the PCTE is the Object Management System (OMS) where these entities are stored and accessed. This replaces the conventional concept of a file system and means that all objects in the environment are handled homogeneously.

2. User Interface

   The Workstation approach implies that a high resolution raster display is used, although non graphic terminals are also supported. Most operating systems now support multiple applications running in parallel and this suggests a multi-window approach. Thus, all the primitives for opening and manipulating windows as well as conventional I/O are provided.

3. Distribution

   Primitives are also provided to cater for the administration of the distribution mechanism. The distribution is, however, largely transparent to the individual user and linked to the data schemata provided by the OMS.

THE OBJECT MANAGEMENT SYSTEM

The OMS is the central store of information. It is distributed across the LAN. Most of the information is kept as local as possible to ensure quick access to data and to prevent unavailability of data during network breakdowns.

Information is kept in volumes which are mounted and dismounted as necessary. When mounted, a volume becomes visible to the whole network. The OMS is based on a typed Entity-Relationship data model, defining Objects and Relationships as being the basic components of the environment information base.

An *object* is characterised by

- a contents, in the traditional file sense,
- a set of attributes, which are individually named and accessed, representing primitive values,

- a set of relationships in which the object participates, where a relationship is a bidirectional link between a pair of objects (uni-directional links may also exist).

All objects, relationships (links) and attributes are typed entities to define their basic properties. The type definitions are contained in special predefined objects known as Schema Definition Sets (SDSs). SDSs can be a specific to a particular project or sub-project or to individual users and user groups.

A running process operates with a set of SDSs known as a Working Schema. This may be viewed as a set of constraints on the properties of collections of objects visible to a particular process. Different views of parts of the OMS may thus be possible for different users working with different Working Schematas.

The OMS primitives may be accessed directly using C. Schema manipulation primitives may further be used through a Data Definition Language (DDL). An interpreter is built on top of the OMS primitives.

Commercial implementations of the PCTE exist for Bull machines, and versions for other workstations, including Sun and MicroVAX are expected to be available soon. Several tool development projects within the European community target at the PCTE.

UNCLE

The Uncle project was originally set up to develop a portable and tailorable command and response language for the Portable Ada System designed by Olivetti [6]. This led to the to an initial Uncle interface specification [7].

Uncle was redefined to work with the OMS part of the PCTE and uses a prototype version developed by Olivetti in Pisa, Italy.

The PCTE philosophy makes a clear distinction between the underlying system as described above, and the software tools utilising the services of the system. Therefore a command language processor (CRLP) is a tool allowing user-system communication. This allows context and application specific command and response languages (CRLs) to be utilised. Uncle is a tool to create such CRLs in the context of an Ada Programming Support Environment.

Abstract Machines

Central to Uncle is the idea of a Universal Store, a logical concept which would ensure independence of and, at the same time, sharability of user resources and data by tightly controlling access to objects contained within the store while maintaining relationships between these objects and their registered holders and sharers. The objects contained in the store may range from simple data structures to tools and large databases. In the current implementation strategy, this store is seen as the PCTE OMS.

The Uncle objective of providing facilities which exactly match users' skills and needs is achieved by an abstraction mechanism which gives the user his or her own "Abstract" Machine. All abstract machines are derived from a single all-powerful **B**asic **A**bstract **M**achine, the *BAM*. The BAM is implemented on the services provided by the PCTE. Individual abstract machines, in turn, are implemented in terms of the BAM or other higher level abstract machines.

Different abstract machines may share objects. When the problem to be solved requires object sharing the user may become aware of the existence of other users and their abstract machines, even if the views of the objects may differ. In this case mechanisms to build strict control for manipulating shared information are necessary.

The Abstract Machine provides the the user with

- a set of abstract data types
- a set of objects, which are instances of these data types
- a CRL to manipulate the objects

Command and Response Language Types

The Uncle system has three levels of abstraction. At the lowest level is the Object Management System. The BAM types are implemented using the OMS primitives. The types provide an interface designed for consistency with Ada applications.

New CRL types and new views of existing types can be defined when designing new abstract machines. From the system developers point of view CRL types are objects, from which new, more specialised instances can be derived to populate a specific user interface. A CRL type object defines structure and values for instance objects, operations allowed in these, and permissions which are necessary to call these operations. Note that operations are treated as objects themselves - they belong to a CRL type which models executable objects.

Structuring the Object World

Rather than offering the primitive (but powerful) entity-relationship facilities to structure a user's object world, Uncle provides predefined directory types, which can be used to construct hierarchical systems in a fairly conventional manner. Similar to Unix directories these are groupings of objects identifiable by (simple) names. Several related object types are grouped into classes, and the class an object belongs to can also be used to distinguish objects.

Directories are pairs of simple names and object references, which will allow the user to retrieve objects from the underlying store. This level of indirection also allows directories to share objects - a concept easily represented using the OMS primitives.

To model the possibility of wider object world structuring, including short and wide-haul networks, the concept of a context element was introduced. A context element references several directories and can therefore be seen as a group of entry points into a directory structure. The universe of objects is now structured by having many context elements forming a tree. The user would usually be working with objects found through directories at a leaf context element of this context tree. Objects in other, higher context elements can be made accessible to the user if the user has a means of describing the path from his current context element along the context tree. By introducing a naming convention for context elements object in any part of the context tree can be identified.

This provides a natural model of the object world accessible to the user: objects local to an activity embedded within the context modelling a session, which is again contained in the local network node and so on.

Protection

All user can, in principle, share objects individually or on a group basis. This requires a powerful access control mechanism if owners of objects are to be guaranteed control of their objects. Another requirement is that the protection mechanism be adjusted to the semantic level of the user interface. This means that a simple read/write/delete tag does not offer the right level of granularity.

As stated previously, the definition of a CRL type includes the definition of the operations which can be performed on objects of this type. The first level of the protection mechanism relies on the typing of objects, which means that formal and actual parameter types in operation invocations must match. Operations do not only specify types the actual parameters must have, but also a set of permissions the caller must hold for this object.

In processing an invocation request the CRLP first performs name resolution. If the operation and parameter objects are visible to the given caller, then the actual permissions associated with the objects in the current context are evaluated to return a set of access rights. A simple key - lock technique is now employed to find out if the operation was permissible. The formal permissions for each object are also evaluated to produce sets of access rights, which must form subsets of the corresponding actual ones.

Only if all these conditions are fulfilled does the CRLP retrieve the objects from the *bonded store*, which locks the objects stored in the OMS from uncontrolled user access.

The process of evaluating permissions to produce access rights can be used to express a large variety of security strategies. Permissions may be conditional or unconditional. Unconditional permissions always return the same set of access rights. Conditional permissions can return context dependent sets of access rights. Assume, for example, that an optimising compiler requires a large amount of processor time, whereas a standard compiler could be more efficient. A permission could be used to control the load on the system:

```
if      Number_of_Users > 30
then    {Standard_Compile}
else    {Compile,Link}
```

CONCLUSIONS

In developing a prototype Uncle tool for the PCTE (OMS) it turned out that the power of a relational OMS considerably facilitated the construction of schemata representing the context and typing structure of the Uncle system. Generally one can build the tool faster and more flexibly using the PCTE rather than using Unix directly. One can conclude from this that PCTE is one step closer to the objective of making even complex tools interchangeable.

The largest drawbacks the encountered were limitations imposed by the prototype - performance as well as functional limitations.

There are, however, some more fundamental problems.These are clashes between design philosophies. The PCTE OMS implements a typing and subtyping structure with inheritance, but does not cater for abstraction. It is thus very usable for refining, but makes it more difficult to implement the Uncle concept of abstracting to produce CRL types. Also, the OMS does not treat types as objects, which means that link between objects and types can only be system, but not user defined.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the invaluable contribution of the German partners in this project, U. Kugler and C. Unger, and would like to thank the reviewers and project officers of the CEC for their helpful comments.

REFERENCES

[1] "Reference Manual for the Ada Programming Language", Ada Joint Programme Office, January 1983

[2] Newman, I. (Ed.), "Operating System Command and Response Language Specification", Working Draft, International Standards Organisation, July 1987

[3] Beech, D., Gram, C., Kugler, H.-J. , Stiegler, C., Unger, C., "The IFIP WG 2.7 Reference Model for Command and Response Languages", Springer Verlag, Berlin, September 1985

[4] Bourgignon, J.P, "PCTE: A Basis for a Portable Common Tool Environment", in: Roukens, J., Renuart, J.F. (Eds.), "ESPRIT '84: Status Report of Ongoing Work". Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1985.

[5] X/OPEN Group Members, "X/OPEN Portability Guide", Elsevier Science Publishers B.V. Amsterdam , 1985.

[6] Olivetti Grouppo Informatica Distribuita, "Portable Ada Programming System", Global Design Report, January 1982

[7] Hopper, K., Kugler, H.-J., Unger, C., "Uncle - Towards a User-Oriented Command and Response language for the PAPS", October 1986

A DISTRIBUTED DESIGN ENVIRONMENT FOR DISTRIBUTED REALTIME SYSTEMS

Christoph Senft
Institut fuer Technische Informatik
Technical University Vienna
Austria
senft@vmars.UUCP

ABSTRACT:

Over the past few years the need for systematic design of real-time systems has received considerable attention due to the increasing criticality and complexity of the applications. This paper presents a distributed project support environment for the design of fault tolerant distributed real-time systems. The innovative aspects of this environment relate to the handling of real-time from the requirements phase to the detailed implementation stage and the integration of the design tools with the evaluation tools. Data refinement, function refinement, evaluation and project management are supported by an open, but coherent tool set. The uniform man-machine interface is one the integrating factors of the whole environment.

The design environment is implemented under the UNIX† operating system on a state-of-the-art graphic workstation with pointing devices. The X-window system supported the creation of the window and icon driven user interface. All the information gained and used during the design process is stored in an entity-relationship model implemented by the relational data-base DB++.

KEYWORDS:

Software Engineering, Distributed Realtime Systems, Design Environments, Design Evaluation, Graphic User Interfaces.

†UNIX is a registered trademark of AT&T in the USA and other countries

1. INTRODUCTION

In the 1970's, the effort in software engineering focused primarily on the development of technical procedures and management methods that addressed specific aspects of the software life cycle. In the 1980's, this emphasis has shifted towards automated support for various tasks associated with software development and towards integrating tools and techniques into coherent software development environments. Examples of these environments are the Gandalf project /Hab86/, the Unix programming environment /Ker81/, the Saga project /Cam84/, the Smalltalk environment /Gol84/, the Ada environment /Ste81/, and the Toolpack project /Ost83/.

Current research is taking place in at least two areas. The first area is concerned with developing programming environments, that ease the programming process, i.e. powerful integration of editors, compilers, debuggers, etc. The second area focuses on the construction of system development environments, which cover management aspects and life-cycle support of a system development from the overall beginning. Only few of the developed environments are suitable for the design of real-time systems, e.g. Mascot /Mas86/. But there is none which focuses on hard-real-time systems and the consequences of heavy load and fault conditions.

This paper presents an *environment for the design of fault-tolerant distributed real-time systems*. The distributed technical approach is reflected in the design methodology, in data and project management support. These main aspects are integrated in one framework, which appears to the user as one system due to the consistent interface of the different tools.

Section 2 discusses principles of distributed real-time systems which guarantee a foreseen behavior even under extreme load and anticipated fault conditions. The objectives and goals of a support environment to create and manage such systems are stated in section 3. Every design environment has to be supported by at least one certain method. Section 4 presents a methodology for the design of distributed real-time systems, which follows the stated principles. The design environment, proposed in section 5, integrates the technical development and the derived project management approach. A final conclusion discusses, how the environment meets the required goals.

2. PRINCIPLES OF DISTRIBUTED REAL-TIME SYSTEMS

Distributed computer systems have gained wide acceptance in real-time applications. They hold the potential of improved reliability and increased functional flexibility over central architectures. Functional flexibility is a

necessary prerequisite for extensibility, which permits the development of a generic distributed computer system for a large real-time application area. Without redesigning major parts of the basic system, each particular application instance can then be implemented with minimal effort /Fra81/.

In a real-time system typically a control object (the environment) is connected to a controlling system (the computer) via sensor and actuator based interfaces. The control system accepts information from the environment via sensors, processes the data, and outputs the results to actuators. The output-data influence the control object such that the effects can be observed via the sensors closing the loop as shown in figure 1.

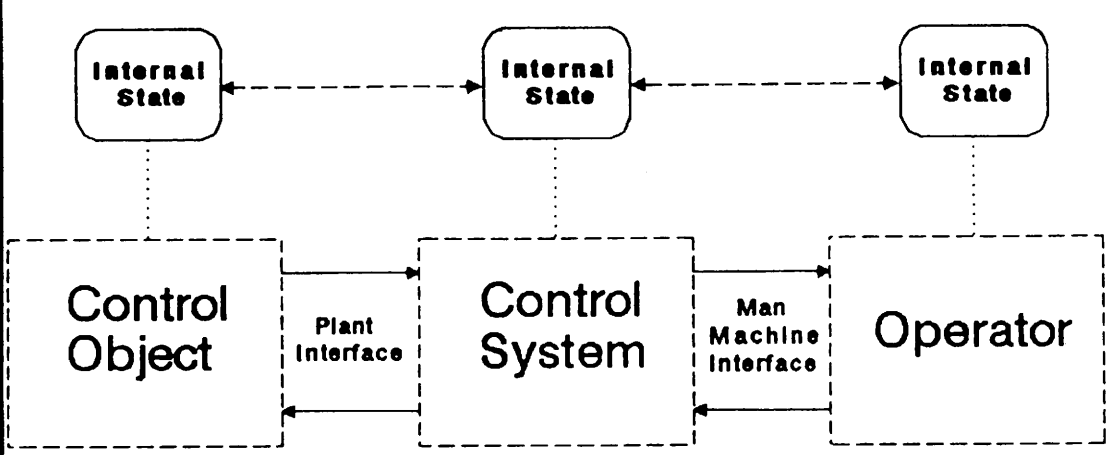


Figure 1: A typical realtime system

It is important to avoid inconsistencies between the internal states of the controlling system, the controlled object and the operator. As an example consider an "automated car" before a traffic light. The car vision system recognizes the value "the light is green". If the *validity of* this *information* expires, i.e. the light turns to red before the car moves on, the inconsistency between the controlled object (the traffic light is red) and the information in the car guiding system (the traffic light is green) can lead to an accident.

The control system must respond to a stimulus from the control object within an interval dictated by the environment, the so called *response time*. This response time must be guaranteed even under extreme load and anticipated fault conditions. An example of a system meeting these basic requirements is MARS /Ko85a/. In the following, several principles for the design of real-time systems as specified in /Ko85b/ are considered.

The operational structure of a real-time application can be described by a set of components which produce, consume, and process information and by the information flow between these components. This structure can be modeled by an information flow graph - the components form the nodes and the channels form the edges. Usually, this graph is not a tree, so the appropriate structure of a real-time application is a ***network***, not a hierarchy. In order to manage the structure of such a large network the concept of *clustering* is introduced. A cluster is a subset of the network with a high inner connectivity. Clusters composed of communicating components form the basic elements of our design methodology.

A *component* is a hardware/software unit of a given functionality and performance. This early binding of software functions to hardware units is necessary for the early analysis of the timing and reliability properties of a design. The concept of component implies the information hiding principle, i.e. the inner details of component implementation are hidden from the outside. Components are autonomous, they are responsible for plausibility checks on their input and output, and for intelligent behavior under fault conditions. We further assume that components have the selfchecking or failstop property, they either operate as intended or do not produce any results.

The information exchange between the distributed components is expressed by the *exchange of messages* only. A message is a named unit of information which is formed for the purpose of communication. In MARS two types of messages, event and state messages, are distinguished. Event messages are used in the exchange of information about events, which occurred at a given time. They are queued when received and consumed when read. State messages are used to exchange information about the state of the environment which has been observed at a given time and is assumed to hold for a certain time interval. A new version overwrites an old one. State messages are not consumed when read. They can be read an arbitrary number of times. In the case of periodic state messages the transmission period of the messages determines the maximum information flow, making the communication traffic predictable even under high load conditions.

To guarantee the correctness of a system under any load conditions, a system should be designed and validated for the peak load. As a consequence the *timing behavior* of the whole system will be predictable. This consideration of the worst case should not only concern the timing aspects, but also the *single fault case*. It has to be assumed that every physical unit fails after its characteristic mean time between failures. The effects of such failures and their consequences for reliability and availability of the whole system have to be analyzed during the design phase.

3. GOALS OF A DESIGN ENVIRONMENT

Methodology Support:

A large number of methods, techniques, and tools have been developed to address various aspects of system development and evolution activity. Initially, these approaches focused on the coding activity, but more recently efforts have been made to cover all phases of the life cycle, from the initial system concept to testing and system modifications. Due to this broader range of activities a methodology has to be supported by an environment of integrated tools.

Extensibility:

However, system technology is advancing so quickly, that any environment locked in the state of the art would rapidly become obsolete. Thus it is desirable that an environment even initially tailored to current needs, can be adapted to incorporate future methods and tools. This flexibility and adaptability should not only concern the ease of change of the environment to new technologies, but also facilitate the creation of new tools to expand and improve the tool-bench.

Scalability:

Analogous to the extensibility of the environment by new methods, it must be possible to design systems of different magnitudes in size and effort. Moreover, not only the development of systems of a fixed size should be supported, but also the creation of small systems which in time can grow to larger ones. In this way the design process is open ended and no restrictions on the architecture of the environment with respect to the increasing size of the designed application will occur.

Project Team Support:

Another prime objective is to provide support for project teams rather than just for individual programmers. In general, such teams are distributed over distinct geographical locations or organizational entities. The support system must therefore also be distributable.

Integration:

Integration emphasizes the difference between an environment and a loosely coupled set of tools. Usually, tools in an integrated environment share a common intermediate representation of the system and present a

consistent user interface. The purpose of integrating a design environment is to build a unique tool that does not force the designer to perform mental context switches, e.g. between different editor interfaces or various data-base languages.

Reusability and Repeatability:

Different studies /Boe84/ have shown that in the very long run the biggest productivity gains in system engineering will come from increasing use of already existing software. Moreover, the costs of development of the environment are so high that it has to be suitable for a wide variety of applications or projects.

Ease of Use:

In many projects, development tools are not accepted by the working personnel due to their complexity and difficulty of use. An environment, which integrates a set of tools, should simplify and not complicate, their usage. This can be achieved by a multiple window display, state-of-the-art pointing devices and effective graphic tools.

4. THE DESIGN METHODOLOGY

4.1. Basic Concepts

Software design and specification is a well established field in computer science research and application development. The transformation from ideas to deliverable software is guided by a software development process model, e.g. the traditional cascade-model /Boe81/ or advanced approaches like rapid-prototyping /Acm82/. These phased development models are supported by many different development techniques. /Was83/ contains the results of a questionnaire-based survey of 48 software development techniques in a condensed way. For more details according to a lot of techniques, see e.g. /Bir85/.

Almost all these techniques are developed to produce conventional software, i.e. software which is not related to real-time (and hard real-time) topics. Usually, a certain matter of criticality is charged to real-time problems. Faults in the software could cause enormous costs in material and potential loss of human lifes. Thus, a design methodology developed for the support of real-time software must gain a concrete understanding of the phenomenon time and the related fault tolerant aspects. Real-time systems

are strongly connected to the resources and power of the underlying hardware (e.g. the processor), the operating system, and the communicating network. Four basic concepts have to be underlined:

Network Approach:

A primary reason for the inadequate specification of real-time systems is determined by the use of hierarchy-function models. Anything worth to be said about something must be expressed in a maximum or fewer number of pieces /Ros77/. Any part, that is not sufficiently easy to grasp, must be broken further into pieces. This implies a hierarchical, top-down decomposition of the whole into easy-to-grasp chunks. We feel that ***the operational structure of a real-time application is a network*** and not a hierarchy. In the network representation a real-time control system is described by a set of components, the nodes, which produce, consume and process information exchanged via channels, i.e. the edges of the network. In order to manage the complexity two refinement-steps are used. The first step is the horizontal partitioning of a given system into disjunct clusters; the second the vertical decomposition of a cluster into components. The network approach supports the parallel design of clusters and components, and the connection of additional clusters or components due to requirement changes. Components are modules of information hiding /Par72/, and can therefore easily be expanded into a cluster without change of the rest of the system.

Specification of Real-Time:

In the non real-time world, we are only concerned with the value-domain of information, i.e. the correctness. In a real-time environment, any information has to be assessed in two domains, the value-domain and time-domain. Consider the car-example again: the correct, but late information "the light is green" is useless and dangerous. The ***specification of real-time*** begins as soon as possible in the system life-cycle, i.e. during the requirements phase.

Semiformal Design Approach:

Requirements engineering must include rigorous, but natural ways to describe models of realworld problems. In the ***semiformal system design approach*** we follow neither a fully formal nor an exclusively informal way. It is unnatural to start a problem description in a formal language, as the engineer is hindered in his liberal way to state all the information related to the problem. On the other hand it is not recommendable to use only unstructured, ambiguous and inconsistent narrative text libraries. A minimum on structuarization and formal arrangement is introduced, i.e. requirements consist of several attribute-slots which can be filled with narrative text. The

general bipartition in non-functional and functional requirements, e.g. /Yeh82/, is very function-oriented and does hardly support data-orientation in the requirements phase and thus in the whole life-cycle.

Function and Data Orientation:

System requirements are distinguished in general, transaction-oriented, and data-oriented requirements. They comprise several timing and failure domains, e.g. minimal and maximal time between stimuli of transactions or criticality and cost-factors in failure-effects. These system-requirements determine the establishment of the clusters, components, tasks, and messages during the design-phases. ***System-refinement*** is realized as well ***function-oriented*** (e.g. transactions -> sub-transactions -> tasks), as ***data-oriented*** (e.g. data-requirements -> cluster-data-field -> messages). The allocated requirements are refined, too, e.g. system-requirements to cluster-requirements and cluster-requirements to component-requirements. This repetition of similar elements is incorporated in the user-interface of the tool-set.

5. THE DISTRIBUTED DESIGN ENVIRONMENT

5.1. Outline

There are two distinct views of what constitutes a good design and programming support environment. On the one hand it should nurture creativity by providing freedom, flexibility, and extensibility and on the other, it should promote orderliness by providing control, discipline, and accountability. Some people might argue that the two views represent those of academia and industry, or developers and managers. A good design environment must support both aspects: the ***technical development*** of a system and its ***project management***, e.g. by the use of an appropriate ***data management***.

The technical objective is the design of an environment for the creation of distributed real-time systems within the framework of the design methodology proposed so far. The project management includes the management of the development in planning, control, and evaluation. The data management has to provide mechanisms for the capture, storage, and retrieval or inference of all potentially useful information relating to product and process.

5.2. The Environment Tools

We distinguish between *design creation tools* and *design evaluation tools*. Design creation tools support the system analyst in the creation of the distributed real-time application; evaluation tools can be used for the analysis of a given design and the verification to the proposed requirements. The analysis can be carried out in a formal and automated way. In contrast, design is a creative process. It incorporates the conversion from informal requirements to a formal specification and cannot be automated by nature. However, this convergence can be guided by similar application designs processed so far. Figure 2 gives an overview of the tool set.

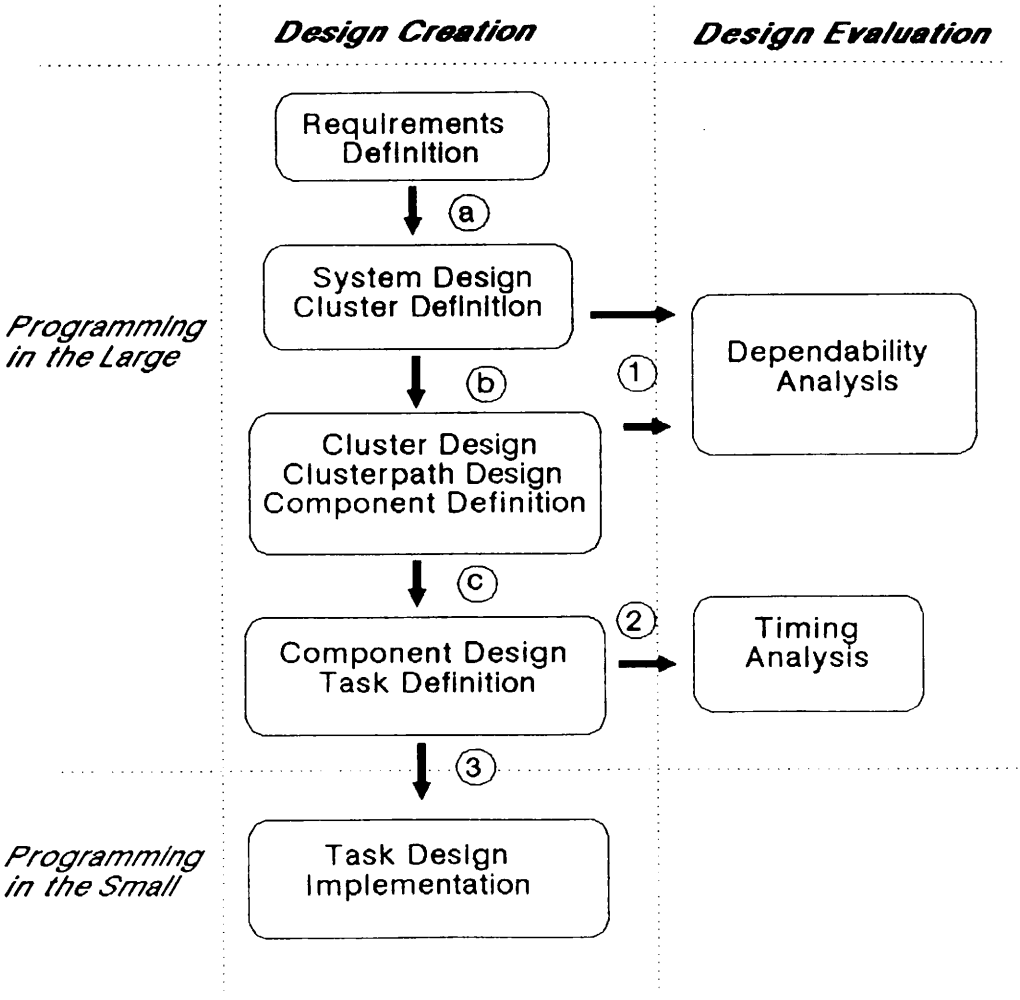


Figure 2: The toolsystem

5.2.1. Design Creation Tools:

Requirements engineering is a systematic process for the development of requirements through an iterative process of analyzing a problem and documenting the results as requirements. The ***requirements definition*** phase provides the engineer with a tool needed to gather the information, to reason about, and to understand the problem domain. As already stated, requirements are specified in a semiformal way. They are divided into functional transaction-oriented, data-oriented, and general requirements. Functional requirements are represented as transactions, according to the well known stimulus-response model. A transaction is characterized by stimulus, response, maximal transaction time, minimal interval time between stimuli, maximal interval time between stimuli, and a criticality description. Data-oriented requirements, the so called spots, include the physical data-producing elements (e.g. sensors) and a description of their behaviour in terms of input and output data-items.

The specified system requirements (transactions, data-spots, and non-functions) represent the system on a very abstract level. It is a key concept of a system design technique to reduce abstraction by decomposition. As already mentioned the final representation of a real-time application is a network. Since it is too complex to translate the system requirements in a single step to the components we take one additional step, the breakdown of the overall system to the cluster level. There are at least two clusters in every system, a controlling and a controlled cluster. The definition of the clusters and the cluster-interfaces is guided in the ***cluster definition tool*** by a "structured walk" through the system-requirements.

Every cluster can now be refined independently of the rest of the system. The design is taking place in numerous ways, in the refinement of the cluster requirements, e.g. cluster-items to messages and cluster-transactions to sub-transactions (component-transactions), and in the establishment of the components. These activities are supported by the ***cluster design and component definition tool***. Special cluster characteristics have to be also specified, e.g. the parameters of the general cluster-communication system, the so called cluster-bus. All the cluster-data-items related to one cluster build the cluster data-field. This data-field will be refined into a set of messages. A message is characterized by one sending and several receiving components, the send-time, and the validity-time of the message information.

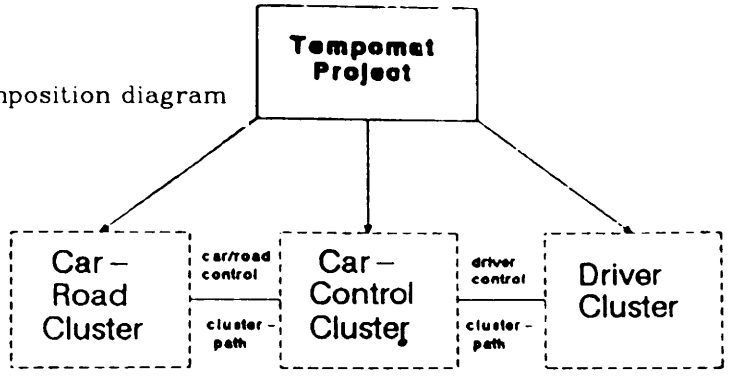
The basic elements of the whole system, the components, are located on the cluster bus and communicate via message exchange only. They are determined by the component requirements, e.g. the allocated subtransactions, the sent and received messages and the usage in the cluster, e.g. as an interface-component to another cluster. A component is specified in detail

during the *component design and task definition* phase, where the tasks and their characteristic time attributes for the implementation are modeled. Different kind of tasks are distinguished. In the MARS kernel it is guaranteed that the most critical tasks, the so called hard real-time tasks, will stay within their specified time limits /Dam87/.

User Interface:

The global user interface outside the tools will be explained later. Inside a tool design activities can be realized textually and graphically. A kind of template structure is chosen to specify the explicit values of an object in a textual way. The refinement of objects is supported by the handling and operation on graphical diagrams. We distinguish between decomposition and relationship diagrams. The next two examples are taken from /Sen87/. The decomposition of a "tempomat system" in three nearly independent clusters and their clusterpaths is shown in figure 3, the relationship between the "calc-throttle-setting" component and its imported and exported messages is given in figure 4.

Principally the screen within a tool is divided in four non overlapping windows. In the two right quarters the different design objects are represented as little icons, i.e. in the lower part the input objects of a former phase and in the upper part the actual new designed objects. In the lower left part all graphical presentation and processing is done, in the upper left part the textual template of an actual object can be edited. The information between the four parts is transferred with the help of a three-finger-mouse.



'igure 3: Example of a system-decomposition diagram

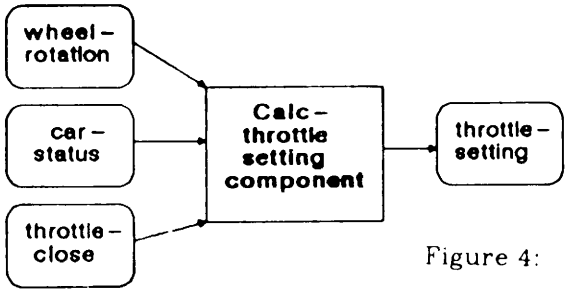


Figure 4: Example of a component-messages relationship diagram

5.2.2. Design Evaluation Tools:

It is the problem and challenge of every methodology to evaluate the early phases of system design in order to detect failures as soon as possible. It is well known in the software engineering community, e.g. /Boe81/, that the later errors are detected, the disproportional the cost-increase will be. In the developed approach the evaluation of a given design is possible in a very close loop from the viewpoint of timing and failure behaviour. Since these analysis are automated, design alterations can be evaluated efficiently and effect the system cost in a bearable way. Both checks are CPU-time consuming. The distributed approach makes the use of different workstations as single task machines reasonable.

It is the objective of the *timing analysis tool* to determine if the timing properties of the given design are consistent with the timing requirements stated in the requirements phase. This checking concentrates on a predictable timing behavior under heavy load conditions, i.e. it is a worst case analysis. The tool input contains the characteristics of the individual tasks defined in the component design, e.g. messages produced or consumed, basic cycle time, maximal execution time(s), and information about the cluster e.g. size and bus-properties. Complex algorithms try to find a solution, how the different tasks can be scheduled on the component-processors and how the communication system can handle the message traffic between this scheduled tasks.

According to /Mul87/ the dependability of a system, i.e. reliability, availability, safety, is investigated with respect to system cost. Three types of cost are considered, cost for the design of the application, cost for delivery and installation of a system, and cost for running the system, e.g. cost of system failures. The interactions between the design structure, failure effects, fault occurrence and fault/error handling, and repair and maintenance strategies are analyzed in the *dependability analysis*.

Different methods for the analysis and description of dependability models exist, e.g. block diagrams, fault trees, markov-chains, etc. During the last decade different tools have been developed to support these models on a mathematical and theoretical level, for a general comparison see /Mul86/. The proposed new approach connects these basic levels with a practical application layer bounded to a system design and integrates all the aspects in one tool. The applicability of this model can be seen as well from the design-evaluation view, as the design-guidance point.

5.3. The Distributed Data and Project-Management

Every design environment uses a certain form of information or data base to carry all the information according to the supported project. Every large software project maintains a project library containing a collection of project components which include plans, documents, forms, code, schedules, resources, etc. The organization of this project base and their differing use by the project staff has to be well considered.

According to the Stoneman report /Bux80/ many approaches can be followed to build a central database. All project members are to be supported in their work by one project master data base. Each individual user deals with some subset of the project base in a certain way. It must provide views depending on user roles in the project and capabilities depending on user expertise on the environment. Thus, the underlying structure must support abstraction and composition of objects depending on these roles.

The central database approach works well as long as small projects are developed and as the developers work on the same station. Usually projects based on a relational data base architecture decrease in performance from a certain size of data. It seems that a ***distributed data-approach*** in a distributed design environment for the creation of distributed real-time systems is more realistic. A fully distributed and loosely coupled approach of the design of the information base in many independent "small databases" is chosen due to the still unsolved problems of data integrity and data consistency in strongly coupled distributed data-bases.

All tasks within a project should be well defined, and tasks for every member of the project team should be clear at all times. The various tasks should be properly coordinated in such a way that the efforts of different team members can be combined to accomplish the overall project objectives. Efforts of individual project members should be protected so that their stored data are not undermined by the actions of others, intentionally or accidentally.

These requirements demand a management architecture which is based on a network structure. All participants are coupled in a loose net, where different workstations with different tools are considered as the nodes and a special communication system as the edges. It could be even possible that one tool requires a work station for its own, e.g. if complex markov-modeling is done for the evaluation of a fault tolerant design. It does not matter how many tools are distributed on different stations as long as they fulfill the basic requirements of a consistent user-interface and the interrelated data exchange. Every node in this net includes and maintains its own little database, closing the gap between the ***distributed management-approach*** and the

proposed distributed database-approach.

Todays most powerful database management systems (DBMS) are based directly or indirectly on the relational data model. In /Ber87/ several handicaps are presented why even relational DBMSs are inadequate for the data handling requirements of a design environment, including i) storing multiple versions of data, ii) storing large, variable-length objects, iii) storing of flexible data-types, e.g. with complex nested structures, iv) flexible storing of graphs and dependencies, e.g. flow graphs, syntax trees, integrity constraints.

An E/R approach on the logical level:

Interest in the entity-relationship concept /Che76/ stems from the need for a modeling tool, that surpass implementation oriented data models (the hierarchical model, the network model, or the relational model) in specifying designers needs and requirements. The concept of entity and relationship is natural to modeling a system on a logical level. A simple diagram technique visualizes entities, relationships, attributes, and value types. Therefor the E/R approach is useful to support the design phases by capturing data and their relationship from the requirements phase to task definition.

A hybrid database approach on the implementation level:

Although the relational database approach does not seem to be sufficient for representation of structured design objects and interdependencies, we found it the best model available. The relational DBMS DB++ /Agn86/ was chosen because of its availability on all current workstations and its implementation embedded in the Unix philosophy which facilitates development. For large amount of textual data in one attribute, e.g. the contracts specifications, Unix files are used. The possible support of "unlimited" textual data dominates the performance loss by opening an additional file.

The refinement from the entity relationship model to DB++ relations and Unix files is done by some simple rules. The mapping consists merely of translating each object into a relation and each relationship into a relation. The attributes of the original entities are the domains of the new relations. Some complex attributes became themselves relations. Unlimited textual data are stored in Unix files, the file name is the domain (attribute) value of the associated relation. Tables 1 presents 3 relations and their domains of the primary physical data-spot entity of the requirements phase. In table 2 the relation between cluster and clusterpath is shown with data of the example in figure 3.

Physical Spot Object	
domain	type
name	string
header	string
description	textfile
initial-cost	integer
running-cost	integer
failure distribution	string
repair distribution	string

Spot Failure Cost	
domain	type
spotname	string
probability	real
cost	integer

Spot Repair Cost	
domain	type
spotname	string
probability	real
cost	integer

Table 1: Mapping of a physical-spot entity to three relations

cluster	*clusterpath*
car/road cluster	car/road-control path
car-control cluster	car/road-control path
car-control cluster	driver-control path
driver cluster	driver-control path

Table 2: An "E/R relationship" relation with examplary data

5.4. The Contractual Approach

The network architecture reflects the management need to coordinate a multi-person effort on a common project. It is important for its success that each activity and sub-activity must be defined in terms of nature and characteristics of its inputs, outputs, and operational environment. This must be agreed on before related activities are initiated. If the work for a client has to be undertaken by a subordinate, the agreement must be recorded in some way, e.g. in terms of management in a contract. The subordinate may split up his activity in subactivities fixed in sub-contracts with other servers. Generally, it is possible to model a project and its activities in a structured set of contracts. This *contractual approach* of system development was first proposed by the Istar technology and environment, which can be generally used for software development and evolution /Leh86/.

In our approach, a contract is the only interface between different tools and their users. Therefore, it should include all necessary information for the subcontractor to maintain his own little database and to accomplish his tasks. The core of any contract is a precise and concise specification. A contract specification has three main components: i) a header, ii) a technical specification, and iii) a management specification including constraints on the process or product and their acceptance criteria. It must also enclose

deliverables such as the contract product and required reports.

Once a project has been started, a contract hierarchy emerges. The amount of contracts active will grow and shrink as tasks are initiated and completed. The hierarchy defines the decomposition of project activities at any time. As previously mentioned each contractor holds its own database, so only the collection of all such bases comprises the overall project information base. A snapshot of the dynamically changing contract-structure and their related data bases reflects the state of the project. A sequence of snapshots provides the total project history.

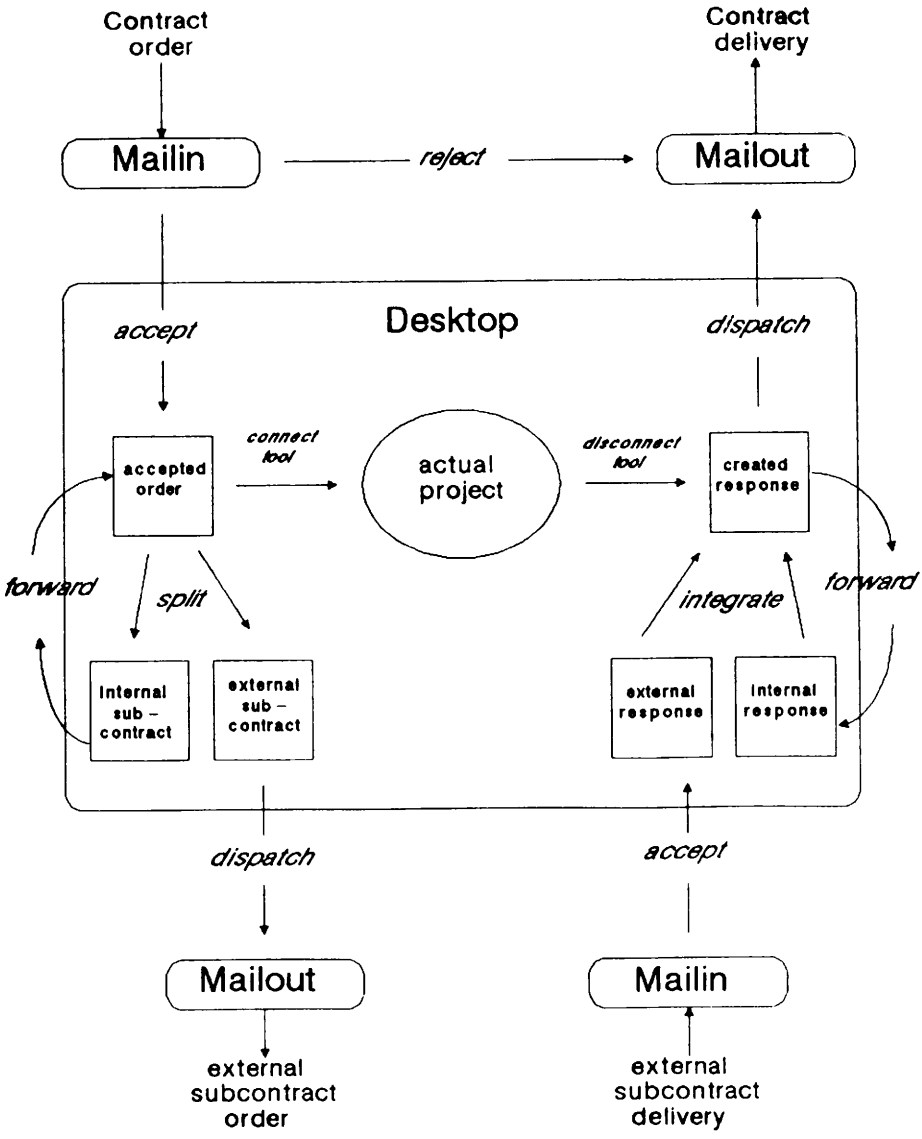


Figure 5: The contract passing scheme

The architecture of a loosely coupled tool-and-workstation network bears another important design goal, extensibility and scalability. The proposed construction supports the development of large systems by the open endedness of the related tool network. There are no constraints and limits in personnel and hardware involved. Moreover, the evolution and/or modification of tools are facilitated by the nearly independent nodes with contractually defined interfaces and the concept of information hiding.

Figure 5 demonstrates the main states of a contract and its operation in a tool. A new contract for the supplier reaches the "mailin" box. It is rejected, i.e. rejection explanation added and moved to the "mailout" box or accepted, moved to the suppliers working area (desktop). Now he can decide if he wants to complete the contract i) alone or ii) with the aid of internal and external subcontracts.

5.5. Desktop oriented design environment user interface

The contract passing scenario revealed the few important objects in the user interface outside the tools: mailin box, mailout box, contract folder, desktop for actual projects, design toolbox, office toolbox. At this point it seems to be reasonable to describe some aspects of the design environment user interface on the global tool level. On the nongraphic oriented terminals the handling of the objects and tools are embedded in the Unix programmers toolbench. In the graphic oriented implementation on the vaxstation (the core design tools are graphic oriented) these objects are represented by icons as shown in figure 6.

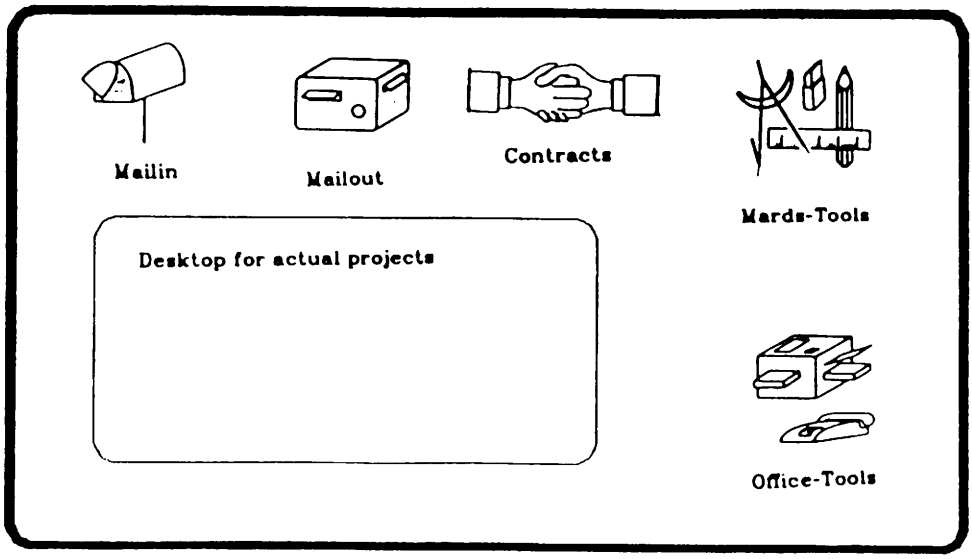


Figure 6: On the desktop level

Manipulating certain objects in command oriented user interfaces, e.g. the widespread Unix workbench interface, is based on a sequence of *<command> <options> <object>*, which does not fit the natural human way of thinking: *<object> <activity> <tool>*, e.g. orange-peel-knife. Disadvantages in the common approach are i) that a user must know the exact name of a command (including options) which provides his desired activity, ii) that the same activity needs different commands with different objects, e.g. the commands rm and rmdir for the activity delete-a-Unix-object, iii) that the user must input all names in an error prone character oriented way. The interface model, which is achieved on the MARDS desktop level, is based on a simple *<object> <tool>*, *<object> <activity>* approach, supported by the graphic and icon oriented X window software.

Desktop user interfaces have been pioneered at the Xerox Palo Alto Research Center during the seventies. The most common desktop activities have been made popular by Apple's Macintosh user interface, e.g. take and move an object is simply achieved by selection of the object with the cursor controlled by the movement of the "mouse" and "dragging" it with a pressed mouse-button to the new desired place. Opening a folder is realized by simply clicking on it. Besides these de-facto standardized activities, we implemented the application dependent actions in two ways. Activities, which are valid on most objects, e.g. look at, print, copy, are realized by tools in the office toolbox, where you find icons for glasses, printers, copy machine (<object> <tool>). For example if you need a high quality printout of a contract, open the office toolbox and drag the laserprinter-icon over the desired contract-paper or vice versa (you don't need to know any print command). Activities, which are valid only for few objects are not represented by graphical icons, they are presented as a little pop-up text menu attached to the selected object, e.g. if you click on a new arrived contract in the mailin box, you will get a menu, which contains among others the terms "accept" and "reject" (<object> <activity>).

6. CONCLUSIONS

This paper presents the importance of a design environment for the specification of real-time systems. A new approach is described including three main aspects: the technical development of a real-time system and its project and data management.

The technical part of the system is built on significant principles to handle a system under fault or heavy load conditions. Typically, a real-time system has to react to stimuli from the application environment within a critical, specified period of time. This response time has to be guaranteed even under

extreme load and anticipated fault conditions. If a real-time system is designed for the peak load, it will work under all load conditions. In order to support these requirements the clear network concept of clusters, components and messages - according to the MARS approach - is used. A new design methodology is introduced, that focuses on breaking the complexity of the overall system into subsystems under real-time constraints. In this way the system analyst is assisted to bridge the gap from an informal to a formal representation by a knowledge based decision support, which is based on a reusability of already designed and existing systems.

The sooner a design can be evaluated, the smaller is the risk of transferring faults into the implementation phase at significant additional cost. The evaluation tools in our approach examine and verify the timing and fault behavior immediately after a proposed design.

The special architecture of the design environment handles the whole project management. A distributed workstation approach was chosen to support the objectives of addressing different project members on different project tools on different design stations. Every tool maintains its own little data base and can use its own hardware if necessary. The management corporation between the tools and their users is established by contract and subcontracts, which include all information about technical specification, acceptance criteria and deliverables of the tasks to work on. While the project progresses, a dynamic contract hierarchy grows and shrinks as tasks are initiated and completed.

The environment implementation is developed under the Unix operating system on a state-of-the-art workstation for the graphic supported design tools and some other Unix machines for non graphic requiring tasks as e.g. design evaluation. The X window system supports the creation of the window and icon driven graphical user interface. The relational DBMS DB++ is used in connection with the Unix file system for design information storage and retrieval. As far a complete environment prototype is finished and evaluated on some smaller applications in an academic environment. Further work will concentrate i) on an improvement of the single tools, ii) on the installation of a "personal assistant" component for intelligent design assistance and guidance, and iii) on the development of practical industrial process control applications.

7. REFERENCES

/Agn86/ Agnew M., Ward R., *The DB++ Relational Database Management System*, Proc. of the European Unix User Conference, Florence, Italy, April 1986

/Acm82/ *Special Issue on Rapid Prototyping*, Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop, Columbia, Maryland, ACM Software Engineering Notes, December 1982

/Ber87/ Bernstein P., *Database System Support for Software Engineering*, Proc. of the 9th Intern. Conference on Software Engineering, April 1987, Monterey, California, pp. 166-178

/Bir85/ N.D. Birrell, M.A. Ould, *A Practical Handbook for Software Development*, Cambridge University Press, ISBN 0-521-25462-0, Cambridge 1985

/Boe81/ B W. Boehm, *Software Engineering Economics*, 1981 by Prentice Hall, Inc., Englewood Cliffs, N.J. 07632, ISBN 0-13-822122-7

/Boe84/ Boehm B., et.alt., *A Software Development Environment for Improving Productivity*, IEEE Computer, Vol. 17, No. 6, June 1984

/Bux80/ Buxton J., *Requirements for Ada Programming Support Environments: Stoneman*, US Department of Defense, OSD/R&E, Washington D.C., October 1980

/Cam84/ Campbell R.H., Kirslis P.A., *The Saga Project: A System for Software Development*, Proc. of the Software Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984

/Che76/ Chen P., *The Entity-Relationship Model: toward a unified View of Data* , ACM Transactions on Database Systems, Volume 1, Number 1, January 1976, pp. 9-36

/Dam87/ Damm A., *Kernel Aspects of the Realtime Operating System of MARS*, Technical Report 6/87, Institut für Technische Informatik, Technical University Vienna, February 1987

/Fra81/ Franta W., *Real-Time Distributed Computing Systems*, Advances in Computers, vol.20, Academic Press, 1981, pp.39-82

/Gol84/ Goldberg A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984

/Hab86/ Habermann A., Notkin D., *Gandalf: Software Development Environments*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 12, December 1986

/Ker81/ Kernighan B., Mashey J., *The Unix Programming Environment*, Computer, Vol. 14, No. 4, April 1981

/Ko85a/ Kopetz H., Merker W., *The Architecture of Mars*, Proceedings of the FTCS-15, Ann Arbor, Michigan, June 1985

/Ko85b/ Kopetz H., *Design Principles for Fault Tolerant Realtime Systems*, Mars Report No. 8/85, Technical University Vienna, June 1985

/Leh86/ Lehman M., *Approach to a Disciplined Development Process - The ISTAR Integrated Project Support Environment*, ACM Software Engineering Notes, Vol. 11, No. 4, Aug. 1986

/Mas86/ *Special Issue on Mascot 3*, Software Engineering Journal, Vol.1, No. 3, May 1986

/Mul86/ Mulazzani M., Trivedi K., *Dependability Prediction: Comparison of Tools and Techniques*, Proceedings of Safecomp86, Sarlat, France, 1986

/Mul87/ Mulazzani M., *Dependability Analysis and System Design* , Technical Report 8/87, Institut für Technische Informatik, February Technical University Vienna, 1987

/Ost83/ Osterweil L.J., *Toolpack: An experimental software development environment research project*, IEEE Transactions on Software Engineering, Vol. 9, No. 6, pp. 673-685, November 1983

/Par72/ Parnas D., *On the Criteria to be used on decomposing Systems into Modules*, CACM, Vol.15, December 1972, pp.1053-1058

/Ros77/ Ross D.T., *Structured Analysis (SA): A Language for Communicating Ideas*, IEEE Transactions on Software Engineering, Vol. 3, No. 1, pp.16-34, Jan. 1977

/Sen87/ Senft C., *Remodel - A REaltimesystem Methodology On Design and Early evaLuation*, accepted for the IFIP Conf. on Distributed Processing, October 1987, Amsterdam

/Ste81/ Stenning V., Frogatt T., Gilbert R., Thomas E., *The Ada Environment: A Perspective*, Computer, Vol. 14, No. 6, pp.26-36, June 1981

/Yeh82/ R.T. Yeh, *Requirements Analysis - A Management Perspective*, Proceedings Compsac 1982, November 1982, pp. 410-416

/Was83/ Wassermann T., Freeman P., *ADA Methodologies, Concepts and Requirements*,

NERECO : An environment for the development of distributed software

Giandomenico Spezzano, Domenico Talia
CRAI , Località S. Stefano, 87036 Rende (C.S.), Italy

Marco Vanneschi
Dipartimento di Informatica - Università di Pisa
Corso Italia, 40 - 56100 Pisa, Italy

ABSTRACT

NERECO (NEtwork REmote COmmunications), the system here described, is constituted by a set of tools for the development and the execution of distributed programs. From an architectural point of view the tools supporting such programs have been realized through the enrichment of many sequential languages (Pascal, C language, CHILL) with concurrent statements and the run-time support for such statements. The integration of the concurrent part into the sequential languages has been done in order to preserve the syntactic coherence.
The cooperation model utilized is derived from CSP, with some extentions to allow asymmetrical and asyncrounus communications; moreover, some statements for fault treatment have been added. The run-time support implements the syncronization and the communication among the processes; for its implementation the IPC of UNIX 4.2BSD, and particurarly the TCP/IP protocol has been utilized. The **NERECO** system has been implemented in C language on a collection of SUN workstations connected by Ethernet.

1. INTRODUCTION

Efficient and reliable distributed software design needs mechanisms and tools based on a clear semantics, characterized by high modularity, expressive power and robustness. Though the kernel and the lower levels of remote communications offer mechanisms for the control flow (e.g. dynamic memory management, swapping, buffer area for messages

his research has been supported by CSELT.

trasmitted by value or by reference) the greatest part of distributed policies require a set of mechanisms more powerful and at higher level to manage complex communications.

To make a linguistic level available as an abstraction level in which resources are defined like abstract data types encapsulated into one or more manager processes which implement them we need to define :

- resources data-structures,
- a set of operations and associated parameters which can be executed on the data structures.

This enables the distributed software designer not to take into account pre-existent network architecture, communication protocols and operating system, when he has to solve tipical problems of the distributed environment (concurrent activities management, synchronization, data and processes replication, fault tolerance).

A distributed program is a set of processes cooperating through message passing and located on one or more computers. The proposed environment provides a metodology for modular and robust structuring of distributed programs by:

- characterizing the processes in a functional way, by associating a type with each of them,
- using unidirectional typed channels,
- expressing communication forms either point to point (*rendez-vous*) or by diffusion (*broadcast* and *multicast*),
- controlling nondeterminism in communications,
- handling, in a simple and flexible way, error conditions, by detection, confinement and recovering.

The system described here is the first of a tool series which will be developed at CRAI, with the purpose of achieving this goal. It is a prototype of a distributed support for the development of concurrent distributed programs called NEtwork REmote COmmunications (NERECO).

2. MOTIVATIONS

In this section, the architecture and tools of NERECO will be described in general, especially in order to point out design criterions and choices. Several aspects, particularly static and dynamic tools, will be described in the next sections.

The principal goals of NERECO are:

a) making some mechanisms available, in a small number but sufficiently general and powerful to achieve concurrency

management, communication facilities and error recovery;

b) do not bind users to one specific set of statements and data types for sequential part;

c) making the distributed run-time support for concurrent constructs simple and efficient as much as possible, even in order to take into account new mechanisms and fault-tolerant requirements;

d) without being bound to a particular host system guaranting a good portability.

A fundamental problem is to choose the programming language to be offered to the users for developing distributed programs. The requirement **b** is obviously incompatible with the choice of one of the concurrent languages available at present or in development , such as Ada® [**Ada 83**], NIL [**Strom 83**] and CSP-based languages [**Hoare 78**] such as ECSP [**Baiardi 84**], CSP80 [**Jazayeri 80**], Occam [**Inmos 84**], Planet [**Crookes 84**], Joyce [**Brinch 87**], etc. Actually, the use of languages with powerful abstraction mechanisms on data and on control flow, like Ada, could be suitable, but it is opposite to the needs of many users, who want to use a programming style already experienced in centralized systems. Further, more complex languages contradict requirement **c**, because the complexity of the sequential part has remarkable impact on concurrent run-time support.

The choice of NERECO is to put a set of concurrent mechanisms, with well-formed sintax and semantics, into a sequential language, completing static development tools with those for concurrent part. This approach has been used successfully in other projects, such as Conic [**Magee 86**]. At the moment NERECO is based on Pascal language.

As regards point **a**, our choice has fallen on a set of mechanisms derived from ECSP language [**Baiardi 84b**], both because familiar for authors and above all for his characteristics of flexibility and generality, already fully experimented. Some mechanisms, such as process nesting, with respect to ECSP have not been considered, while others, such as broadcast communication, have been added.

The run-time support of concurrent constructs has been also realized by cooperating processes, as a virtual machine on an existing operating system (OS). In this case the OS is UNIX† 4.2BSD. The fundamental aspect is that the designer is able to transform typical mechanisms of concurrent languages into system calls easily by a good knowledge of concurrent programming metodologies. The virtual machine, in the

® Ada is a trademark U.S. Government-Ada J. P. Office.

† UNIX is a trademark AT&T Bell Laboratories.

NERECO development, has been initially described in a ECSP-like language and then "translated" in C language [**Kernighan 78**] plus UNIX system calls [**Ritchie 78**], with a limited design effort.

Another advantage of the choosed approach consists in a greater possibility to concieve reconfiguration and fault-tolerant mechanisms, in the support implementation . This aspect has been essential in order to allow the user to get rid of whatever problem derived by network physical configuration and by network reliability.

In outline, dynamic tools realize:

- the installation/initialization of program
- the distributed run-time support of concurrent part, constituted by processes which interpret concurrent constructs and by processes communicating on the network
- the concurrent constructs logging.

3. THE PROGRAMMING LANGUAGE

Like ECSP language, the processes cooperate through communication channels and by input/output commands; the channels are typed and are identified by the triple:

(sender process set, receiver process, message type)

they can be symmetrical or asymmetrical, generally they are asyncronous. Channels are not shared objects, but are private of the receiver process, in order to achieve better protection. Finally, channels can be dynamic: in this case the name of the partner process is a variable of *processname* type.

The cooperating model utilized has some differences with respect to the ECSP language, both as limitations and as extensions. More considerable limitations are in the lack of parallel command for the processes activation. In the NERECO system a concurrent program is constituted by a set of processes at the same level and all activated at the same moment.

More important extentions are:

* explicit declaration of message type, in order to have a complete static type checking and process interfaces checking
* addition of output asymmetric communication forms : *multicast* and *broadcast*, required by the lack of parallel command.

The informations which characterize a process in a distributed program are its name and its type. The process type is useful to identify a class of processes, such as: monitor, file_server, etc. This characterization is useful when, for example, a process needs to operate on a replicated resource available on the network. The process sends the request towards all the resource managers without mentioning the name of each process, but only

their type.

On the inside of each process there are the declarations of the process itself and its partners, as follows:

self < *process_id*> : < *process_type_id*> ;
partners < *process_id*> ,...,< *process_id*> : < *process_type_id*> ;

To allow dynamic channels management it is necessary to declare variables of *processname* type, variables whose values are process names and on which two constructs are defined: **connect**(*X,P1*) (to assign a value and the communication rights) and **detach**(*X*) (to assign the undefined value and to revoke whatever communication right); the declaration is as follows:

procvar < *procvar_id*> , ..., < *procvar_id*> ;

further it is possible to specify a range in which his values can vary.

Channels are "logic objects" realizing the communication among processes into the program. A communication channel, always unidirectional, is considered private object of the single receiver process. Channels can be static or dynamic; in the first case the name of the partner is represented by a constant, in the second case it is represented by a *processname* variable. It is necessary to specify that in the communication constructs, channels are not mentioned. In the constructs the names of partners quoted in the channel declaration are mentioned, differently from other languages (e.g. Occam), in which ports are used.

The channel message type is constituted by a pair (**co, T**), where **co** is the type constructor and **T** is the type offered by sequential language. In pure syncronization channels there is only the constructor. Static channels can be defined as follows:

- symmetric and syncronous
- symmetric and asyncronous
- asymmetric and syncronous.

Dynamic channels can be defined as follows:

- symmetric and syncronous.

Let us show, for example, the syntax of an asymmetric syncronous static channel:

chan from (< *process_id*> ,.....,< *process_id*>)
type < *costr_id*> (< *msg_type*>);

Notice that, having explicitly declared the message type, makes it possible to check automatically the process interfaces, increasing reliability and making easier the integration test. Processes define, by channels, visible points through which it is possible to make requests and to receive messages. Program security is considerably enhanced guaranting that a process can send or receive a message on a channel if and only if the message type is equal to the channel type.

Communications are realized by i/o commands: **send** and **receive**. The **send** construct can have symmetrical or asymmetrical form:

send *(< proc_id> , < costr> (< msg_var>));*

send (**all of** (< *proc_id_list*> , < *costr*> (< *msg_var*>));

in the first case, only a partner exists, in the second there is a set of partners, defined by a list (*send multicast*) or by a process type identifier (*send broadcast*). Nondeterministic constructs are similar to the ECSP ones, namely **repetitive** and **alternative** commands with input guards and priority.

The syntax of the **receive** statement is :

receive *(< proc_id> , < costr> (< msg_var>));*

receive (< *procvar*> :**any of** (< *proc_id_list*> ,< *costr*> (< *msg_var*>));

in the first form, there is only a sender, in the second there is a set of senders, but one of them delivers the message.

A process can terminate at any moment whether as a result of a construct failure, or by natural termination. For this second case the construct **terminate** is provided, which lets the process it executes terminate, informing all the partners. Notice that a process can exclusively execute his termination, and constructs are not provided to force other processes termination.

Fault treatment policies handle the communication failures because of:

- partners termination
- channel disconnection
- physical communication media faults.

Failures can be handled by means of the **onfail, onterm, onprot** clauses. They make possible to execute some recovery actions (*forward recovery*) when a failure occurs. In fig. 1 we used the **onprot** clause as a mechanism to continue the process *trans_1* execution when a communication fails because the partner has disconnected the dynamic

channel, refusing the access to a resource.

```
TRANS_1 ::
        . . . . . .

    send ( EXE_O2, ready_to_commit())
      onprot
        send ( EXE_O1, abort()) ;
        send ( EXE_O3, abort()) ;
        terminate ;
      endrec ;
        . . . . . .
```

Fig. 1. The onprot clause

4. STATIC TOOLS

The architecture of NERECO is composed by two parts. The first, identified as the *off-line component*, is constituted by a series of modules devoted to support the development of a distributed program. They are: the *preprocessor*, the *consistency checker*, and the *configurator*. The second, identified as the *on-line component*, is constituted by a set of modules devoted to support the program execution. They are: the *installator* and the *network server*. In this section we address the *off-line component*, in the next one the *on-line component*.

4.1. The preprocessor

The preprocessor carries on, essentially, the role of a compiler for the concurrent part of the language. It produces a program in sequential language starting from a concurrent one. The preprocessor output will be the input for the compiler of the sequential language. Notice that it is a "rational preprocessor" [**Aho 86**], namely it is not a simple macro translator, but a precompiler for a language enhanced with new data structures and new control flows.

The preprocessor operates on every single process composing the program. As we said above, the dependence of NERECO from a particular sequential language is confined on the preprocessor. From that it is easy to deduce that the choice of a different sequential language in which to embed the concurrent part involves to change only this module.

In the preprocessor design a lot of attention has been paid to preserve the syntactic and semantic coherence of the host sequential language. The preprocessor generates the sequential code after the execution of the syntactic and semantic analysis of the concurrent part. In the code generation the concurrent constructs are translated into function calls, which contain the code to communicate with the run-time support, using the Inter Process Communication (IPC) of Unix 4.2BSD [**Leffler 83**]. Further, it generates a table containing the necessary informations to execute the consistency controls among the processes of the program.

4.2. The consistency checker

The consistency checker executes the static analysis of consistency among the concurrent objects of the processes componing the program. In this phase the entity distributed program is created, assigning a name to the set of processes. The consistency checker analizes all the objects which have a global interaction on the program; they are: process names and channels.

The consistency checker analizes the tables generated by the preprocessor for every process, and works out the following operations:

1. consistency analysis among the declarations of processes;
2. production of diagnostic messages on the insubstantiality;
3. synthesis of the global informations, about all the concurrent objects of the program, in a global table.

4.3. The configurator

The configurator provides the physical configuration of the distributed program on the network hosts. It receives from the user the host name on which each process must be executed, hence it takes care to transmit the executable files on the corresponding hosts. Finally it creates a table of correspondence between processes and hosts, which is useful to the run-time support. It is not necessary to allocate the executable files on the network hosts, having a distributed file system available, like that of UNIX 4.2BSD rel. 2.0 [**Lyon 84**],

A program, once developed can be configured in all possible ways without changing the code of the processes. This is possible because the concurrent constructs are independent from the particular processes location. It is the distributed run-time support, which provides the

communication routing, on the basis of the configuration table. Now we are going to enhance the configuration facilities by putting in the run-time support a tool for load balancing configuration.

5. THE DISTRIBUTED RUN-TIME SUPPORT

The NERECO run-time support has been realized as a virtual machine, by a set of cooperating processes located on the hosts involved into the program. They are implemented as UNIX processes, communicating by means of lower level inter-process communication facilities. The distributed run-time support mainly implements the execution of concurrent constructs offered by the language. Besides this, it provides the initialization of the program and the logging of concurrent executed constructs.

The NERECO run-time support is constitued by two logic components (everyone of which is composed of a set of processes) :

1. **Installator**: it provides the installation of the distributed program;
2. **Network server**: it provides the run-time support of the concurrent part of the language.

For processes implementation and for processes communication we used the UNIX facilities, in particular *sockets*. They also offer rough mechanisms for error detection.

5.1. Sockets

In the previous releases of UNIX, interprocess communication was realized by means of *pipe*, which provides a unidirectional and symmetric channel. The use of *pipe* is restricted among processes which have a common ancestor, namely among processes which are forked by a parent. Further, *pipes* can't be used among processes located on remote hosts. In UNIX 4.2BSD new IPC facilities have been added, extending the *pipe* concepts. They are called *sockets*.

Sockets implement bidirectional communication channels among detached processes, without a common ancestor and even located on remote hosts. The utilized protocols are TCP/IP and UDP. To TCP/IP corresponds the reliable *stream socket*, to UDP corresponds the *datagram socket*. The *stream sockets* are not so efficient as *datagrams*, because they offer higher services. On the other, *datagrams* force the user to handle, by program, if any failure or message duplication occurs. We preferred to utilize *stream* type because of the high reliability required by our system. In the *datagram* hypothesis we should enhance the number of communications for avoiding errors and failures.

Sockets have resulted suitable for solving all the problems of local and remote communications in the NERECO run-time support. Further, they

have been useful to check error condition in the communications. By them it is possible that a process failure can be revealed by all the other partners located on the same host or on a remote one. Therefore *sockets* offer to the programmer the basic tools for the distributed handling of failures.

5.2. Architecture

The main phases of distributed run-time support are :

a) **Start execution**: the user asks for the program execution;

b) **Rendez-vous**: from the node on which the execution request has been done, some actions start to make an early set of channels towards the other nodes;

c) **Network set-up**: in this phase the logic network is completed, by building a complete connection among all the hosts;

d) **Local set-up**: on every node the data structures of the run-time support are set-up and the instances of user processes are created;

e) **Run-time support**: in this phase all requests of communication among the processes are served and moreover the support for all concurrent constructs is provided.

The logic component Installator implements the Start execution, Rendez-vous, and Network set-up phases. Its task is to receive the user requests and to settle a logic communication network. To provide these facilities, the Installator is composed by the following UNIX processes:

- **NERECO initializer** (NRC)
  The NRC is the user interface component. The user invokes the NCR execution from the shell environment, specifying the program name. NRC delivers to the local RCSP process the execution request.
- **Remote Connection Service Point** (RCSP)
  There is a RCSP process in every node, its address is constituted by a pair : (host address, Internet port). RCSP receives the request from NRC and communicates with other RCSP processes on the remote host involved in the program. Then it forks and executes the LIS Master process, whilst every remote RCSP execute the LIS Slave process. After that the RCSP breaks away and waits for other programs execution requests.
- **Local Initializer Server** (LIS)
  There are two kinds of LIS processes: on the node where the execution is requested is present the LIS Master, which will check the Network set-up by communications with the LIS Slaves installed on the other nodes. LIS Master and LIS Slaves establish a logic mesh network, implemented by means of *sockets* as

described in fig. 2. Each LIS Slave receives the name of the processes which will be executed on its node. When the Network set-up is terminated the LIS Master process informs the NRC process. Hence every LIS process is transformed to a network server process, by the *execl* system call.

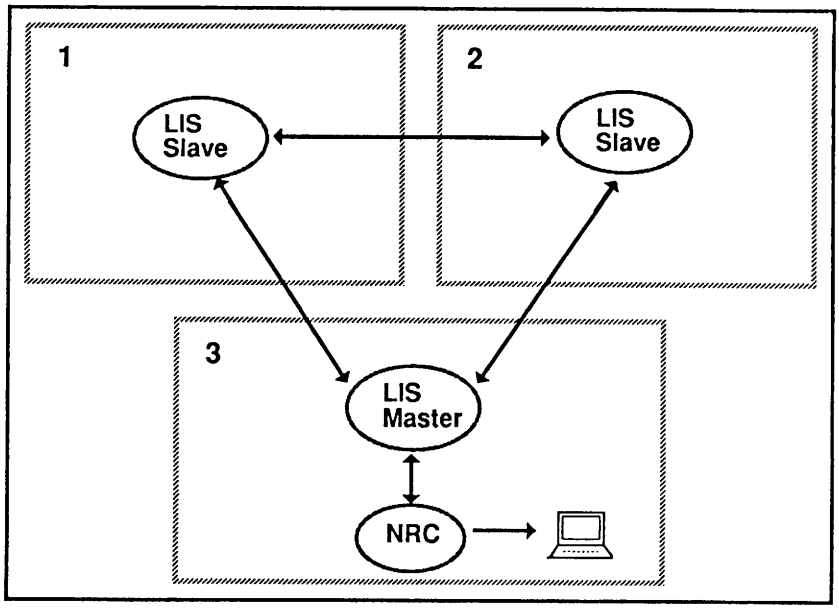


Fig. 2. LIS connections on three nodes.

The NRC and LIS processes are created dynamically for every distributed program whereas the RCSP process is created in one single copy in every node when the bootstrap occurs, and remains always active; in summary it is a UNIX daemon. Obviously, if the program is configured on only one node, any network procedure is not executed and the program set-up is done only in that node.

The network server is the logical component which provides the run-time support of the concurrent constructs. It implements the various forms of communication provided by the cooperating model, and provides to interpret the nondeterministic commands, and to manage dynamic channels. The network server is created by transforming the LIS process image, when the installation is finished with success. It is composed by the processes : NS, IN, OUT, and NETLOG. Later we will denote the user processes of which the program is composed as Process Component (PCs).

- **NS**
  The NS process, created on every node, loads its data structures putting in them informations about local PCs and remote PCs which are mentioned in the communication constructs of local PCs. These informations allow the NS process to control the local PCs and their remote partners status. The NS processes create local PCs by means of *fork* and *execl* system calls, hence they set out to serve the requests coming from local PCs or from remote hosts. In the first case the NS process will forward the communication requests towards the remote hosts by the OUT process (according to the fig. 3), or it will provide to support the communications among the local PCs. Further, NS processes cooperate each other to maintain consistent informations on the PCs status.
- **IN and OUT**
  The IN and OUT processes are created by LIS process during the installation phase. Every OUT process is connected to every IN process and viceversa. The existence of these processes makes it possible to enhance the computing bandwidth of the network server by executing in parallel extern communications and internal computing. In fact, they serve to send towards and receive from the network.
- **NETLOG**
  This process provides to maintain the log of every executed concurrent construct, in order to monitor the distributed application. It is only created on the node from which the execution has been requested, and receives informations from every node. Finally, the NETLOG stores the informations in a file.

Processes which compose the network server end their tasks when every PCs is terminated. Hence they provide to the network server termination.

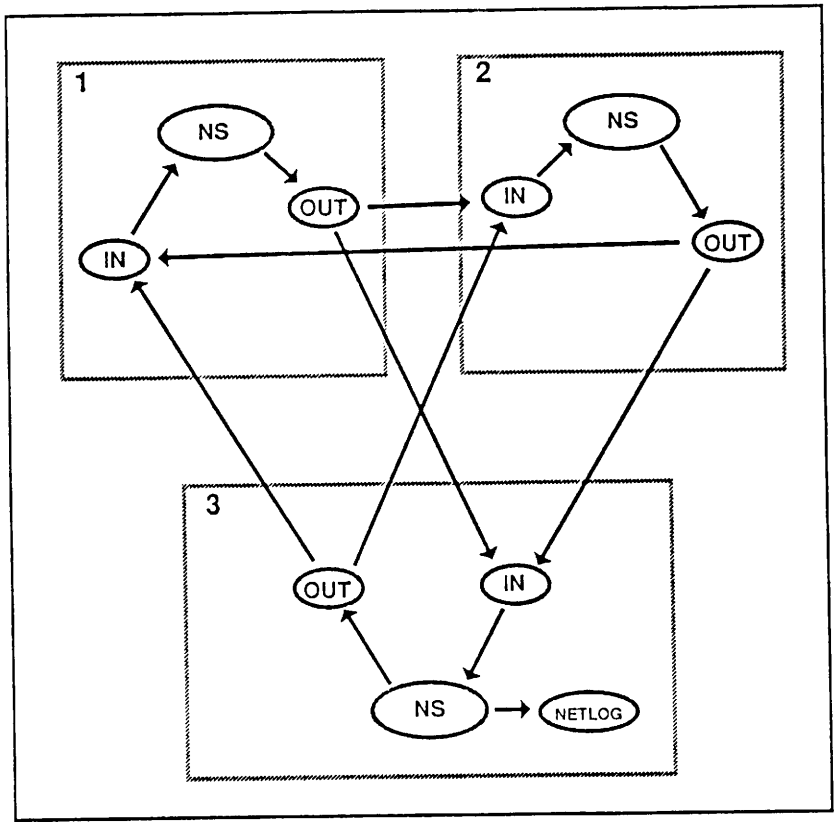


Fig. 3. Run-time support processes on three nodes.

6. CONCLUSIONS

This paper has described the language and the architecture of the NERECO system. At present, NERECO is used to develop distributed programs. The system demonstrated that the use of an high-level language is useful to develop distributed programs.

Now we are going to put in the system some tools for load balancing management, concurrent debugging, and syntax driven editing. These new tools will enrich the NERECO environment and will assist the user to develop distributed software.

References

[Ada 83]
Ada Joint Program Office, *Reference Manual for the Ada programming*

language, ANSI/MIL-STD 1815 A, 1983.

[Aho 86]

Aho A.V., Sethi R., Ullman J.D., *Compilers: Principles, Techniques and Tools*, Addison-Welsey, Reading, Mass., 1986.

[Baiardi 84]

Baiardi F., Ricci L., Vanneschi M., "Stating checking of interprocess communication in ECSP", *ACM Sigplan Notices*, 19, pp. 290-299, 1984.

[Baiardi 84b]

Baiardi F., Ricci L., Tomasi A., Vanneschi M., "Structuring processes for a cooperative approach to fault-tolerant distributed software", *4th IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1984.

[Brinch 73]

Brinch Hansen P., *Operating System Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Brinch 87]

Brinch Hansen P., "Joyce- A Programming Language for Distributed Systems", *Software - Practice and Experience*, vol. 17, pp. 29-50, Jan. 1987.

[Crookes 84]

Crookes D., Elder J.W.G., "An experiment in language design for distributed systems", *Software - Practice and Experience*, vol. 14, pp. 957-971, 1984.

[DeFerrari 85]

DeFerrari L., Spezzano G., Talia D., "NERECO : Architecture", *Internal Report* , n. 85/24, CRAI, Rende, June 1985.

[Hoare 78]

Hoare C.A.R., "Communicating Sequential Processes", *Commun. of the ACM*, vol. 21, n. 8, pp. 666-677, Aug. 1978.

[Inmos 84]

Inmos, *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Kernighan 78]

Kernighan B.W., Ritchie D.M., *The C Programming Language*, Englewood Cliffs, New Jersey, 1978.

[Jazayeri 80]

Jazayeri M. *et al.*, "CSP/80 : A language for communicating sequential processes", *IEEE Compcon Fall 1980*, pp. 736-740, 1980.

[Leffler 83]

Leffler S.J., Fabry R.S., Joy W.N., "A 4.2BSD interprocess communication primer", *UNIX Programmer's Manual Berkeley Software Distribution*, Virtual VAX-11 Version, vol 2C, Univ. of California, Berkeley, Aug. 1983.

[Lyon 84]
Lyon B., Sayer G. *et al.*, "Overview Of The Sun Network File System", *Sun's Network File System Documentation*, 1984.

[Magee 86]
Magee J., Kramer J., Sloman M., "The Conic support environment for distributed systems", *NATO Advanced St. Inst., Distributed Operating Systems: Theory and Practice, Izmir, Turkey*, Aug. 1986.

[Ritchie 78]
Ritchie D.M., Thompson K., "The UNIX time-sharing system", *Bell Syst. Tech. J.*, 57, 6, pp. 1905-1929, 1978.

[Strom 83]
Strom R.E., Yemini S., "NIL: An integrated language and system for distributed programming", *SIGPLAN Symp. Progr. Lang. Issues in Software Syst.*, pp. 73-82, Jun. 1983.

A KNOWLEDGE BASED CAD SYSTEM IN ARCHITECTURE ON UNIX

Authors' Names :
S. HANROT, P.QUINTRAND, J.ZOLLER(GAMSAU)
E..CHOURAQUI, P.DUGERDIL, P. FRANCOIS,M..RICARD (GRTC)

Authors's Addresses :
GAMSAU/EAM - 70,Route Léon Lachamp 13288 MARSEILLE
GRTC/CNRS - 31,Chemin Joseph Aiguier 13402 MARSEILLE

Speaker name : Michel RICARD - member of the AFUU

Mail address : GRTC/CNRS - 31,Chemin Joseph Aiguier 13402 MARSEILLE cedex 9

Electronic mail address : GRTC@FRMOP11.BITNET

Abstract :

The aim of this project is to realise an intelligent CAD system integrating part of the Architectural knowledge.It is mainly based on the assumption that this knowledge is partially included in the vocabulary of the architect and in the drawing techniques elaborated along the centuries.
In this system the symbolic representation of knowledge is based on an object-oriented language, OBJLOG, we purposely defined as a layer above PROLOG II, and which offers some interesting processes of inheritance.
To implement the geometric model we have chosen SMALLTALK-80, an object-oriented language provided with a rich graphic programming environment, which permits us to realise a quite sophisticated user interface(pop-up menus,mouse device controls..).The communication between the model of knowledge and the geometric one is carried out by UNIX in running Objlog and SmallTalk as two concurrent processes.

INTRODUCTION

A general survey on the CAD systems in architecture shows that they are mainly used in the final phases of the project in the production of the drawings or for the more sophisticated of them in the instrumentation of the project : they are used as tools of development and completion.

So, one of the major problems the systems designers have to face is the creation of a tool available in the design phase, allowing the modelling of uncompletely specified objects or ill-defined problems characterising the architectural design.

The classical programming methods do no fit very well this kind of problem; Conversely, it seems to us that an artificial intelligence approach could bring interesting elements in their resolution.

In that, the TECTON project is a contribution to the resolution of such a problem where choosing an artificial intelligence approach did not mean realising an expert system to solve the architectural problems but rather the creation of a tool integrating part of architectural knowledge concerning the description and the manipulation of objects during the phase of design.

Three classes of problems are investigated :

1-Architectural knowledge modelling.
2-Symbolic representation of architectural knowledge
3-Graphical representation of architectural knowledge.

ARCHITECTURAL KNOWLEDGE MODELLING

The universe of knowledge in architecture is defined as a regular universe which can be described,modeled and represented. It is described through the observation of the knowledges and the

know how, referring to a set of objects and methods used in in the architectural project. Through its descriptive vocabulary, elaborated along the centuries, the analysis of architectural knowledge reveals a whole set of facts and rules which could be carried out in the framework of an "artificial intelligence" approach so as to describe and manipulate Architectural objects while designing.
The investigated domain of Architectural knowledge is described by related objects provided with some properties and whose analysis is performed at different levels and point of views.
An architectural object is defined at four levels .These levels refer on one hand at the different states of progress of the project and on the other hand ,at the geometrical scales and categories of description the architects are used to handle.
At every level of definition the architectural object is described as a composition of material objects and immaterial ones (walls and openings at the level two ; envelopes and rooms at the level three).These objects are described by properties in respect with different point of views (Morphology , Building , Style , Functionality).
The relations associate objects at differents levels in accordance with the point of views(The relation "morphological composition" associates "envelopes" and "walls-openings").
Hence, the knowledge base aims to help the architect at first in the definition of the architectural object at the different levels and then, in its effort to find a consistency between the selected point of views .

SYMBOLIC REPRESENTATION OF THE ARCHITECTURAL KNOWLEDGE

Short description of the model

The knowledge representation is based on the object-oriented model through a language, OBJLOG,we are developing as a layer above and over Prolog.This model provides the architects with a conceptual and methodological framework to structure their knowledge.
The object is the fundamental element which itself splits in classes and class instances. The classes represent the architectural knowledges.
Classes and instances are described with slots and the slots with facets that give the characteristics and values of the slots. Some of those slots and facets are predefined in the language, the others depend of the application domain.
The predefined slots are :

KIND-OF : express the link between a sub-class and its super-classes. This slots contains pointers to a set of classes.

ISA : express the link between an instance and its class. This slot contain a single pointer to a class. The reason of introducing this different slot is to remove the ambiguity, common in object oriented languages, between the inclusion relationship given by "kind-of" and the element-of relationship that exist between an instance and its class.

PART-OF : express the relationship between an object that represent a part of one or more complex objects. It is located in classes and instances that represent part of classes and instances respectively. In the first case, its value contains pointers to classes. In the second case the pointers are to instances. This slot, particular to our model, allows us to define the selective inheritance process between objects : the set of slots that an object which represent a part of a complex object can inherit from the latter is a sub-set of the slots of the complex object. This subset is explicitly given as the value of a distinguished facet of the "part-of" slot.

Inheritance mechanisms :
Inheritance is the fundamental mechanism operating on this type of representation . It allows the description of an architectural object at different conceptual levels and to access,from one given class, to all the classes higher located in the hierarchy. We distinguish between :

Vertical inheritance
it is that of the KIND-OF and ISA slots. It is defined as the access by a sub-class to the whole set of slots of its super-classes. The inheritance process is then multiple (several paths). The process is the following : when a value is needed for a slot of an instance, its own slots are examined. If no value is present, its class and super-classes are examined breadth-first and the list of the values found is returned by the process.

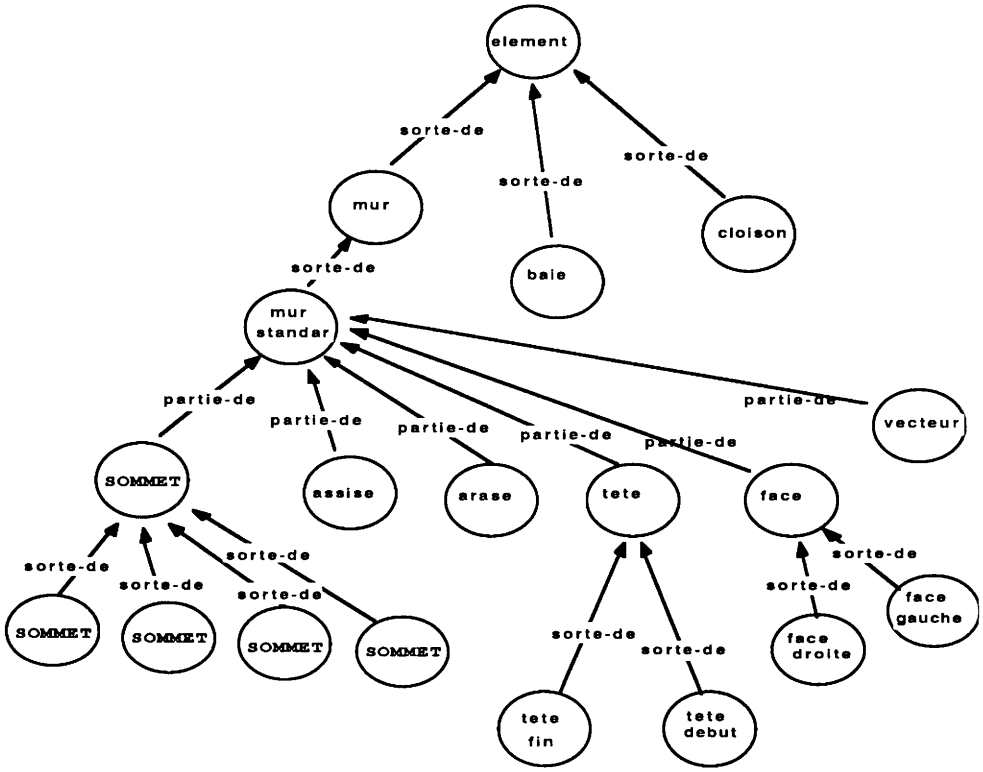
Selective inheritance
it is that of the "part-of" slot. As two classes sharing the part-of relationship represent two different concepts, their instances will be separate and will also contain the "part-of" slot. So the selective inheritance process will begin at instance level before

going through classes and then allows the inheritance of instanciated values. The list of inherited slots is given only at class level. The system will then look for this list during the inheritance process. This process is the following : when a value is needed for a slot of an instance and that it is not found in the instance, the presence of a "part-of" slot in the instance is search. If the "part-of" slot is found, the list of inherited slots is search at class level. If the slot of which a value is needed is given in this list, it is inherited from the instances pointed by the "part-of" slot. This inheritance process is then multiple.

Rule of composition for the inheritance processes.

As it is possible to simultaneously inherit of a value through both inheritance processes, the conflict resolution principle is the following :

- The selective inheritance process has an absolute priority over the vertical inheritance.
- If the selective inheritance can be initiated, then the vertical inheritance process will be disabled.
- If not, the vertical inheritance process is initiated.
- If no value is returned by the selective inheritance process, the vertical inheritance process is initiated from the last instance examined by the selective inheritance process.



graph of a knowledge base with inheritance.

GRAPHICAL REPRESENTATION OF KNOWLEDGE

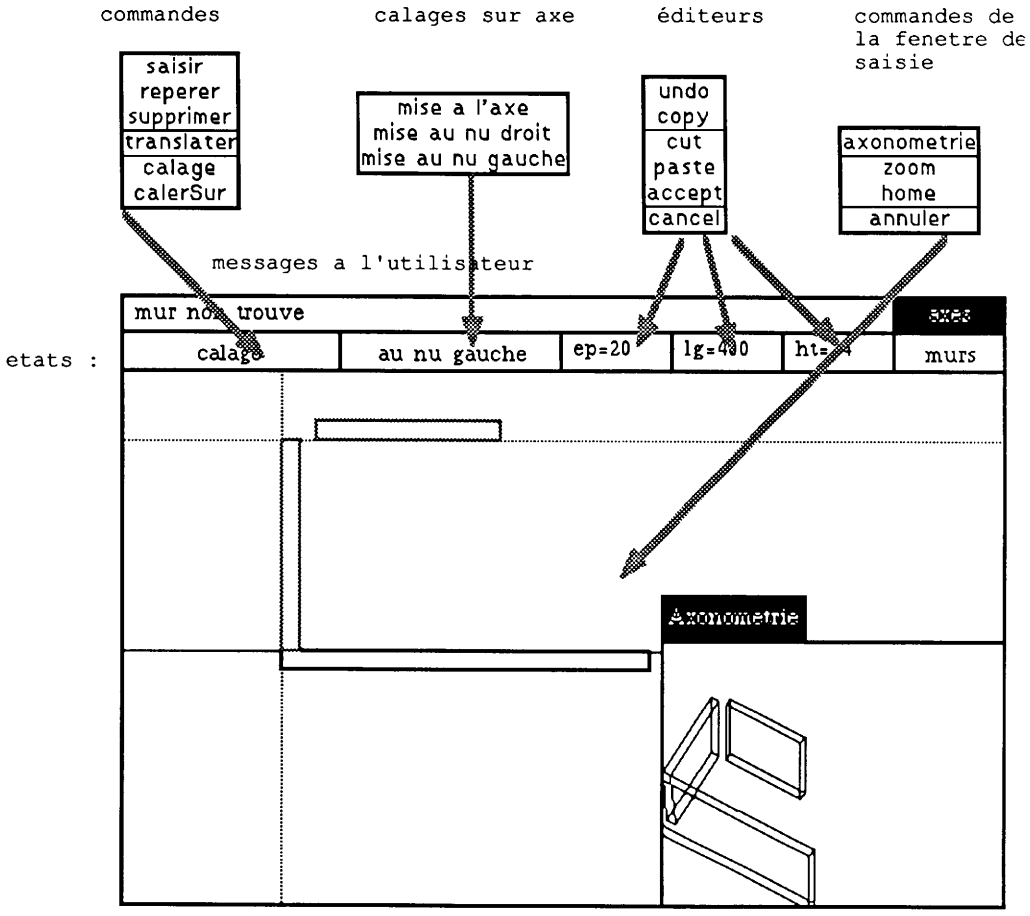
The geometric modelling

The aims of TECTON concerning graphic is to provide the architect with a designing tool allowing :
-The manipulation of geometrical shapes, their definition and the modification of their relations.
-The graphical representation of objects on display devices.
-A high level of interaction between the graphical representations and the conceptual model of the object in the description as well as in the modification of any kind of knowledge of its concern.
To achieve this goal ,we distinguish between the following different levels in the geometric modelling :

-A purely conceptual level in which the geometrical component of the objects is described by its properties and by its relations.
-A formal level of constraints expression, more specific than the previous one ,concerning the geometric nature of the objects as well as their composition.
-A level of 3D graphical representation of the model from which we can extract 2D subviews on a display screen in order to interactively manipulate the object.

The implementation of the first two levels is integrated in Objog itself written in Prolog II. But the lack of graphic predicates in this latter leads us to choose an other language to implement the third level.
We have chosen SmalltalK-80, an object oriented language provided with a very rich interactive environment not only about the tools (many hundreds pre-defined classes and methods) but also about the concepts(MVC triad,dependence,..).
Generally in CAD there are close links between the model, the views and the interactions.SMALLTALK distinguish the three parts and allows their connections so that an action on a certain view associated to the model is managed by the controller.
This modelling fits quite well with our problem
- The *model* contains the data of the problems(classes) and the corresponding operations(methods). In our case the architectural project in its globality.
-The *view* displays (generally on a bit-map screen) informations from the model.
-The *controller* allows the interactions with the project,by means of the keyboard or the mouse which permit, for instance, to point at a particular architectural element on the associated view, or to choose a command in a pop-up menu .

To trigger the updating of the views, the model use the notion of dependency *(changed,update)* .A given model may have any number of couple view-controller that necessary (ex: plan ,face view,side view,3D axonometry..).



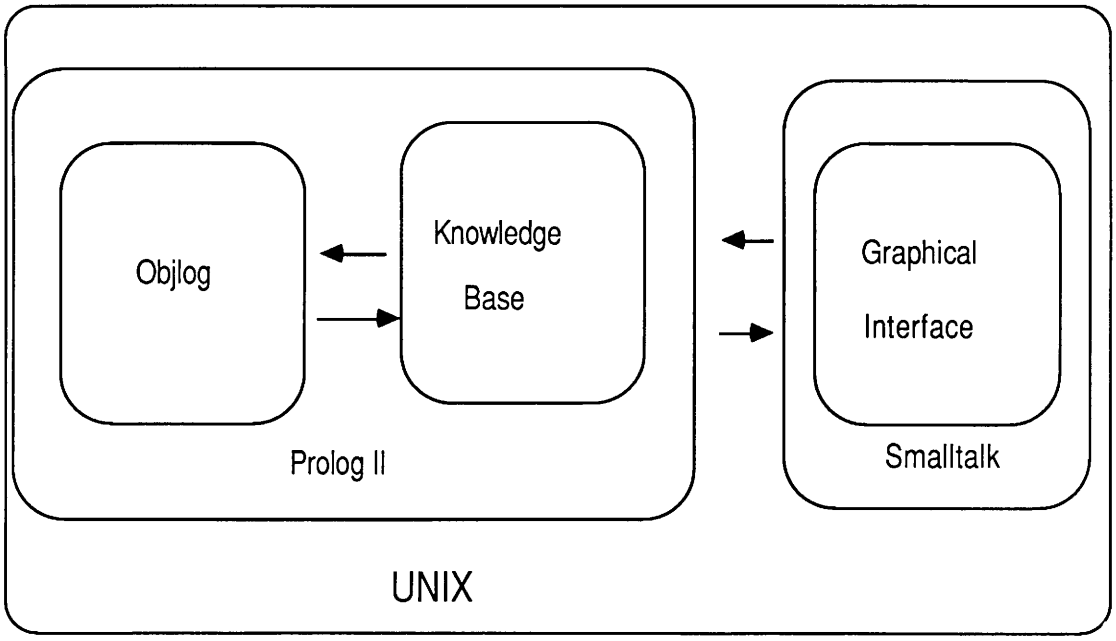
Example of views associated to a model.

The implementation of the System on UNIX.

It was one of our hypotheses that architectural modelling, knowledge representation and geometric modelling modules should be in close connection in the system. The communication between these components must be performed with a complete interactivity :

any graphic modification must be immediately

transmitted to the knowledge module which check that the integrity constraints are not violated; this latter send back the information necessary to modify the drawing of the dependent objects.
In terms of computer implementation, this requires that the knowledge representation module and the graphical module run concurrently .Smalltalk deals only with the graphical representation and the interaction with the user while Objlog manages the knowledge base and the dependencies between the Architectural object; so, to make them communicate we need a software layer above them : it is UNIX.



In Unix a process is the execution of a computer execution environment.Several processes may run concurrently . Data may be sent between two processes via an inter process channel called a **pipe** .
So, to run both the knowledge base management program (Objlog/Prolog process) and the interface user (Smalltalk-80 process) we have to send the following command to the system :

/usr/Prolog/prolog2 objlog.multi -w4 /usr/Smalltalk80

The first process to be activated is Objlog/Prolog ,then the 'w4' option means that the following process (Smalltalk80) must be created and run concurrently and 4 is the number of the co-process; these processes can communicate through two pipes : The pipe 3 from Objlog towards Smalltalk and the pipe 4 from Smalltalk to Objlog . Thus the execution of the previous command will allow us to achieve our goal :A CAD system with a friendly interface written in Smalltalk-80 and with a knowledge base management written in Objlog/Prolog run as a background process.

Communication from Prolog towards Smalltalk.

The Prolog process executes an infinite loop on the following sequence of statements :
1/ wait for the reception of a string from the concurrent process(the user interface).
2/turn this string into a Prolog goal.
3/execute this goal, in fact send a message (unary or with arguments) to a class instance.
4/ send the answer relative to the previous message to the concurrent process.
5/receive the end message.
6/go to 1.

Message processing.

In fact ,Prolog always receive a string of the type

sendsun(name of class,message;<argument or list of arguments>,error)

for instance :

sendsun(wallS,create-inst,<wallS1>,x)

which aims to create an instance named wallS1 of the class wallS
After execution of the message, *sendsun* send the result towards the concurrent process through the pipe 4 (after transforming it into a string) and wait for the reception of a string from the same process indicating that a correct ending for the sequence has occurred .

Along the type of message it receives, Objlog returns either the error type expressed as two digits ('00' for a normal ending free of errors, or an error code) followed by a list of names separated by slashes.

Communication from Smalltalk towards Prolog.

Smalltalk consider Objlog as an object belonging to the class Objlog we purposely defined.
This class manages the transfers between the two actors : it allows the sending of the kind of messages we considered above and the reception of the corresponding answers to these messages.

sendsun(name ofclass, message, <argument or list of arguments>, error code)

In the class Objlog we defined methods to send messages towards Prolog, to interpret the received string in respect with the message sent and to return the end of transmission message.
In Smalltalk we use the class UnixSystemCall containing the set of Unix commands which allows the reading and the writing of data through the pipes defined in the master process.
Hence in Smalltalk user interface, sending a message towards Objlog/Prolog will be performed in sending, in case of an error code is expected, the message **requeteWithError** with a string selecting the message to send in argument, the returned value being an error code.
example:

Objlog requeteWithError : 'sendsun(wallS,create-inst,<wallS7>,x)'.

In case of a detailed answered is expected, we use the method **RequeteWithAnswer** .

Actually we are achieving the implementation of a restricted architectural model on a SUN-3/160 running UNIX 4.2 BSD version 3.0; We use An interpreter Prolog II version 2 on Unix distributed by PrologIA and Smalltalk-8O version VI 2.1 distributed by ParcPlace Systems.

CONCLUSION

This approach is interesting for its taking into account the architectural knowledge in its full complexity. The computer implementation will provide with a general object-oriented system of knowledge representation. The integration of the triad knowledge base - Objlog - interface user on UNIX is fairly efficient and prove that this latter is a great environment for the communication inter program.

BIBLIOGRAPHIE

QUINTRAND P. et al. - La CAO en Architecture.Paris Hermes Publishing.1985.

CHOURAQUI E., DUGERDIL PH. - Application des langages orientés à la CAO de l'architecture. 3emes Journées d'étude sur les langages orientés objet,AFCET,IRCAM,Paris,8,9 et 10 Janvier 1986.

DUGERDIL PH. - Une méthodologie orientée objet pour la représentation des connaissances en CAO architecture. Mémoire de DEA en Informatique et Mathématiques, Faculté de Luminy, univ. d'Aix-Marseille II, 1985.

HANROT S. -Elements de modèlisation des connaissances architecturales. Memoire de DEA en informatique mention XIAO, Faculté de ST-Jerome , Univ. AIX-Marseille III 1986.

CHOURAQUI (E.), DUGERDIL (Ph.), FRANCOIS (Ph.), RICARD (M.) "Le projet TECTON : un système expert de CAO intégrant le savoir architectural. Journées Internationales CAO et Robotique en Architecture et BTP", Marseille, 25-27 juin 1986, ed. Hermes, Paris , 1986, pp. 73-84.

CHOURAQUI (E.), DUGERDIL (Ph.), FRANCOIS (Ph.), RICARD (M.), et All.. Le projet TECTON. Un système expert de C.A.O. Intégrant le savoir architectural. International symposium on Computer aided design in architecture and civil engineering, ARECDAO 87, Barcelona, Espagne, 1-3 avril 1987, 77-79.

CHOURAQUI (E.), DUGERDIL (Ph.), FRANCOIS (Ph.), RICARD (M.), HANROT (S.), QUINTRAND (P.), ZOLLER (J.), GIRAUD (C.). A C.A.D. system integrating architectural knowledge. Fourth International Symposium on Robotics and Artificial Intelligence in Building Construction, Haifa,

Israel, june 22-25, 1987.

BOBROW D.G., WINOGRAD T. - An overview of KRL, a knowledge representation language. Cognitive Science, Vol 1 N°1, 1977.

GOLDBERG A. , ROBSON D., Smalltalk-80 , the language and it implementation, Addison-Wesley,1983

PROLOG II - Manuel d'utilisation . PrologIA , 1986.

A user interface for geographers—what can UNIX*offer?

Roger Bivand

Nordland College, P. O. Box 6003, N-8016 Mörkved, Norway

1 Introduction

Geographers are playing an important role in the development of user interfaces to major data sources. Geography, regional and environmental studies have a scope defined by resolution from the individual phenomenon—a journey to work, the erosive effects of running water—to system-wide interaction up to the global scale. This implies the need to move smoothly from scale to scale, like zooming, and to be able to tackle empirical issues without necessarily being theory-bound. Certain problems have a cartographic solution; others result in more complex reporting forms.

As Openshaw has pointed out (1987), geographers have not developed portable software tools either for themselves, to assist in model development, or for end-users. Much of the basic work has however been done for manipulating the data sources, and for creating low-threshold markets—like the "Domesday project." What is missing is an environment at least promising a consistent approach to portability, one suited to software development.

School geography has made great advances with uncomplicated hardware including now fractal simulation of urban land use patterns. Beyond this, programs continue to build upon Fortran subroutine libraries which geographers have never managed to standardise, despite the potential markets for user interfaces integrating character, numerical, and graphical presentation media. Since leading computing geographers are now starting to use UNIX systems, the paper concludes that the portability and software development qualities that UNIX offers can assist geography as a subject over the whole educational range, and, more importantly, can enhance the perception of geographical approaches as features of user interfaces in more general terms.

2 The geographical domain

A typical description of geography is as a set of items intersecting with sets representing subjects such as ecology, meteorology, geology, economics, anthropology, history and so on. This picture has country-specific versions, with geography in the United States covering only the "human" side, dropping the physical aspect. Geography has no evident market, a feature which has led to the coopting of planning with a variety of adjectives—urban, rural, regional—as a promotion vehicle. Most of the commercial market for software associated with geography has indeed been for planning, for instance transport modelling.

Curiously, this professionalisation of geographical knowledge has not been very successful, with the pursuit of instrumental goals running in parallel with cuts in research and teaching at all levels. There are many examples of early commercial niches as it were backfiring, with transport models being just one. It turns out that forecasting models are very sensitive to the zone boundaries used for trip origins and destinations, something which perhaps should have been self-evident. The consequence is of course that the forecasts are conditional on the chosen zones, the "modifiable areal unit problem" (Openshaw, 1977). In addition, much work with statistical models has foundered on the impossibility of defining a "geographical sample." A typical instance—a study related used car prices in the 49 continental US states

*UNIX is a trademark of Bell Laboratories.

(and DC), to sales taxes and delivery charges on new cars, explaining around 25% of price variability. If one on the other hand looks at the average of used car prices in neighbouring states, explanation is 73%, with tax and delivery charges dropping out of the model (Bivand, 1984). Again, this "neighbourhood effect" should have been expected, reflecting Tobler's First Law of Geography: Everything is dependent on everything else, but near things are more dependent than more distant ones (1970). Finally, study areas or "regions" are not imperviously encapsulated, but are subject to often unspecifiable influences from beyond their boundaries.

In terms of system theory, geography has a domain determined by choices of entity and scale of resolution. One can set bounds on resolution, down beyond the current best in remote sensing, from the Spot satellite, up to lunar earthrise. The French geographer Brunhes has illustrated the link between entitation and resolution by placing himself in the basket of a slowly ascending balloon. From a first view of the hedges around the field, the fauna and flora, the perspective widens to the village, paths, roads, reaching out to the broader region, market towns, rivers; the increase in altitude continuously removes the higher-resolution detail which has gone out of scope.

The unresolved issue underlying geographers' discomfort is the discipline's lack of fit in modern times—which is not an acknowledgement that the times should dominate every field of knowledge. One of the pioneers of computing in geography, Torstein Hägerstrand, has devoted the last decade to considering this question. In essence, it is the conclusions that can be drawn from this train of thought which delimit the qualities required in a user interface of and for geographers.

In two recent utterances, Hägerstrand has explained very clearly the burden of his thought, instanced in the following anecdote:

> "Some years ago a young colleague of mine, Dr Thomas Lundén in Stockholm, told me that the beginners' class of his little daughter has been asked to classify different items. The teacher drew three pictures on the blackboard representing *a spruce tree*, *a bear*, and *a mouse* and suggested that the children should place a ring around the two things which belonged together. Thomas's daughter combined the spruce-tree with the bear. The teacher told her that this was a mistake. She should have joined the bear and the mouse, because they were both animals. When the girl came home she asked: 'But daddy, did I make it wrong? Don't bears live in woods?' As a good geographer Thomas, of course, answered: 'Yes, you are right. They do.' " (1984, page 374).

He continues to draw out the moral in the story:

> "The teacher who maintained that the bear and the mouse belonged together and that the other suggestion was wrong, has an overwhelming scientific tradition behind her. ... We have inherited a structure of knowledge which ties every discipline to its own class of phenomena. ... When the little girl drew her ring around the tree and the bear she demonstrated that at least for an unconfused mind there exist other criteria than similarity for grasping the complexity of the given world. Instead of taking items apart according to category, we could attend more to a given bounded pocket of the world." (1984, page 375).

The term which Hägerstrand employs to depict this pocket is *diorama*[1] which in the above anecdote would represent the wood with a bear about its own business. Parts of the system constitute the landscape, and others projects "submitted" by actors (1982, page 325). Atkin has described a similar conceptualisation as traffic on a backcloth (cf. Gould, 1981). The landscape/backcloth is subject to slow dynamic change, while for the time and spatial resolution chosen for viewing, the action/traffic has a fast dynamic (Bennett, Haining and Wilson, 1985). Returning to Hägerstrand, we can begin to see that the diorama is a powerful modelling tool:

> "A real-world diorama as I understand it can never be fully accessible or describable in an empirical manner. The fullness must instead be taken as the ceiling of a way of viewing the world, the geographic

[1]from the Concise Oxford English Dictionary: ... small representation of scene with three-dimensional figures, viewed through window etc.; small-scale model or film set.

way in its most genuine sense. Nevertheless, it should help us to observe relations which would escape us otherwise. It should help us to ask questions which we would not ask without it. And it should help us to estimate the importance of what we have to leave out of consideration, because we know at least the locus of what is left out" (1982, page 326).

After an examination of some facets of geographical computing in the following section, I shall return to the idea of diorama generation and manipulation as a source for user interfaces.

3 The current situation

3.1 Communication forms

Geographical computing differs little from that of other disciplines, with less emphasis on mapping or remote sensing than one might have expected. It is now only a few geography departments that succeed in topping university usage statistics, despite the handling of large data masses, for example from remote sensing or population censuses. Since the aim of data processing is to arrive at results worthy of passing on to others, it is necessary to pick out the groups with whom geographers wish to communicate. Subsequently one should examine whether this may be supported using relevant software, and whether software or hardware development is called for. The prime current communication function, at least as stressed in recent reviews, is that of publications, articles and conference papers, but takes place chiefly within special interest groups. The content of well over half such items, to judge for example from the titles in the programme of the 1987 European Congress of the Regional Science Association, is based on empirical data. Beyond this, specialist interests address wider audiences in teaching, and in the reporting of research to contracting authorities and the public at large.

The medium most commonly employed is the printed page: text, tables, and figures, including of course maps. This has always been supplemented by live or recorded forms of presentation, especially of photographs, but more recently also overhead transparencies. It is however only in the present period of rapid development of computing hardware that these media have been able to be integrated in communication with selected audiences. The "Domesday Project" illustrates many of the benefits which may be gained by providing Geographical Information Systems (GIS) with suitable interfaces, despite the technical problems encountered and largely solved in its marketed form; the contribution of geographers is described by Openshaw, Rhind, and Goddard (1986). It is this project in particular which possesses a likeness to the diorama concept, permitting access to pockets of data by stepping down a hierarchy of grid sizes.

3.2 Interface features

The present decade has seen a major change in the use of data processing by geographers. This has taken place with the diffusion of interactive terminals on multi-user systems and of microcomputers, also used to emulate such terminals. Transfer of data and programs was previously largely limited to magnetic tape and punched cards, and then as numerical data and program source code, most often in Fortran IV, but also occasionally in Algol. Output units were line printers, and for the fortunate plotters, and were not suited to the production of documents. The priorities revealed in much software development were those of one-off products meeting specific demands, having in common with most of those years' programming few guarantees of achieving the intended result, at least not as efficiently as possible.

As already mentioned, most academic geographers now have direct access to processing power and data storage that have grown by several orders of magnitude within the present decade. Indeed, many of those with whom geographers communicate have enhanced their computing base to an even greater extent; this remark certainly applies to students and the educational sector as a whole, and the household sector. The areas which appear to have hung behind are those involved in the generalisation of one-off programs to libraries of subroutines for use on differing hardware running

under a variety of operating systems. Further, it has only been the educational market which has seen the distribution of geographical software in binary form in any extent, supplemented by the entertainment market to a certain extent.

The issues which emerge are related to the priorities perceived within our discipline, on the one hand for software development, and on the other for production-class systems. They encompass programming language qualities, programming style, portability and compatibility, interface characteristics and device drivers.

Chiefly because of the speed of diffusion of computing machinery, it turns out that the transfer of information even between terminals and hardcopy devices is often a non-trivial problem - something those of us who need characters beyond the 7-bit ASCII set would acknowledge. The transfer of figures is still more bound to the system specification, unless one utilises a programmable filter, such as a screen or plotter device driver, which has been ported to a variety of systems (cf. PostScript). Again, such drivers entrain non-trivial decisions regarding the image being processed, critically the interaction between pixel resolution, raster size, and figure dimensions, solved very creditably in the "Domesday Project" (Openshaw, Wymer and Charlton, 1986). The transfer of information between systems, despite advertised compatibility, is also fraught with difficulty, an example being the proliferation of different diskette formats, and of occasionally inadequate upward compatibility following system updates.

One may object that portability and compatibility are of little significance in activity typically conducted by individuals or small groups and directed to others chiefly in spoken or document form. This does however imply that skills built up by these individuals are carried with them, and not necessarily transferred to their environments, meaning that everyone has to start from the beginning. Such a situation would be rejected in cartography, and has also been in quantitative geography in its modelling and statistical form. As we have built up our experience of statistics, we have undoubtedly become more aware of the pitfalls lying in the path of those who choose to believe the verbatim results they obtain from statistics programs, of whatever provenance. It must be accepted that if commercial software publishers update their products to remove errors, even given the scale of resource use involved in development, then one-off programs must be very vulnerable to the risk of error. This feature is reflected in Nelder's introduction to the collection of statistical algorithms edited by Griffiths and Hill (1985, p. 11), remarks which could, and perhaps should have been made by a geographer:

> "We began tentatively, and we made inevitable mistakes, some of which are tactfully alluded to in this book; what I suspect none of us realized at the beginning were the difficulties that beset the production of a good algorithm. It is hard to get the code right, to ensure that the algorithm is efficient and well-structured, that it adheres to the language standard, that all kinds of potential misuse are trapped, and that the description is clear and accurate. ... To the extent that increasing numbers of people are now learning statistics from running programs rather than from reading books, the executable algorithm has become part of a new and powerful form of literature, with its own style, and perhaps eventually its own masterpieces."

Summing up, geographers have not been sufficiently concerned with the programming tools that they have used, and have not succeeded in economising on development costs by sharing relevant software through subroutine or function libraries. While geographers have worked hard on geographical information systems of various kinds, especially for handling large data sources, the retreived information is relatively highly compartmentalised, and seldom suited for systematic interrogation or simulation. Indeed, in the light of advances in the handling of image data, including maps, outside geography, one might feel justified in concluding that we are neglecting our calling.

4 Criteria for user interfaces

4.1 User identity

The idea of a diorama presupposes an enactment, a staging of one of a range of possible models of a pocket of reality, comprising input from many differing sources. Such a theatrical/movie analogy suggests that a major user identity is that of the receptive group or audience, in front of whom the "drama" is played out. This is also the situation in

museums with displays of this kind, in particular the new wave of exhibition facilities attempting to give the visitor a whole picture of life in a different time/space pocket. The burden of responsibility for maintaining the interest of the audience here lies with the producer, who should be able to tailor productions to suit particular target groups. The speed of the interface is here not critical, since the diorama is presented in a non-interactive form.

In more organised teaching, the interface would be manipulated by the teacher, both in order to present some feature considered of importance for the learning process, and in response to current class activity. This implies that the time taken to introduce changes becomes more vital to the success of the presentation. Finally, in full interactive use, the user should be able to retrieve information in various formats, and to relate this information in a systematic way to relevant models. The diorama used for simulation can then take on some of the qualities of a spreadsheet for a system of interacting elements at the chosen level of resolution.

While the above discussion has been couched in terms of the enactment of events in dioramas of a geographically relevant scale, there is no very evident reason why the experience of other related fields cannot be utilised, nor why others should not be able to use such interfaces in order to present or examine complex pockets of modelled reality. Much is to be learned from software supporting the design of integrated circuits, and from CAD in general, involving as it does the interaction of hierarchical data bases, knowledge rules, and graphical presentation.

4.2 Passive use

For production of a movie or slide-show type of diorama, certain facilities are required. These include the display of the landscape of the pocket of reality to be represented at an acceptable level of resolution, with the ability to reveal the relations assumed to be fundamental for its form in terms of the model. This could be termed showing the reverse of the backcloth. Traffic on the backcloth, or projects in the landscape are to be depicted, perhaps by icons, together with their reasoning—what the actor is doing and which factors are involved in path choice. Interactions between the traffic and the backcloth should also be accessible, although the more often they are revealed in the surface structure, the more the "story line" will be broken. Changes in both traffic and backcloth over time are features requiring further refinement. It is likely that the use of changes in resolution, altering the time and space envelope in scope will ease the depiction of the structuring features, the reverse of the backcloth and so on. Since it is these structures, the frames on which the pictures are hung, that incorporate generalised knowledge, their representation will often take the form of text, tables, diagrams and formulae.

4.3 Active use

Active use of a diorama can occur at two main levels, interaction in the choice of retrieved information and its temporal and spatial scope, and interaction in altering the models and structures used to generate the exhibits. In the second case the interventions will only seldom have a deterministic character, so that the simulation of a range of outcomes may be necessary.

A thematic area may be introduced here to illustrate some of the required features, perhaps the intersection between human settlement and environmental risk. The management of risk from flooding is a key feature of most cultures, either through engineering or social solutions. Other, more modern, threats from pollution are less well socialised, and require more effort in terms of the calculated description of the "real" level of horror. While forest death is associated with acid rain, with consequent effects on settlement, employment and recreation, the systematic relationships are far from clear, and could be explored in the form of a diorama which knew about competing hypotheses relating to factors like felling rates and soil structure deterioration. Again, a diorama approach to the accidental emission of radioactive materials, could combine knowledge of weather and settlement patterns with agricultural production related to season. Access to both the backcloth, and the traffic, and their reverses—the structural frames constraining them—would permit a perhaps less sensational but more reliable presentation of the actual risks.

5 UNIX and user interfaces

5.1 Software development

As mentioned above, geographers need user interfaces for two linked tasks, for software development and for production-quality systems. These needs are associated with the insignificance of computing geographers as a market: nobody is going to produce the tools we might find useful for us commercially. On the other hand, if our production-quality systems are good, then it may be possible to broaden our market from education into entertainment—understood as including enhanced public access to information and the models/dramatisations accompanying this information.

The various national communities have gone their separate ways, with solutions like the use of BBC microcomputers as a standard for software portability. This impedes the transfer of concepts like the "Domesday project" out of its national cradle; not totally, plans for the use of the same disk formats with other data and video material are extant in other countries too. Such bases are however a major hinder for software development, which would better be accomplished under an operating system that provided more uniform handles to low-level functions.

The advantages of UNIX as a software development environment certainly do not require elaboration here, but a number of features which strike a geographer may be worth mentioning. First is its availability, and the range of utilities and libraries included in standard releases. Second is the hierarchical file system, and the relative predictability of the location of the various system files within it. Third is the quality of the command language support given by the system, from the shells to special languages like *eqn* or *tbl*, or in a different context *awk*. Perhaps most important however is the "software tools" approach lying behind the utilities, their modularity and mutual compatibility. This can best be illustrated by the review by McIlroy of Knuth's literate programming solution to a problem set by Bentley in "Programming Pearls" last year. McIlroy shows conclusively that a script linking small but reliable tools can provide a prototype, or provisional answer without the need to deploy virtuoso programming skill (Bentley, 1986). Bentley has returned to this in his recent plea for discussion on profiling: a software tools script can highlight the parts of a problem which would benefit from special attention (1987).

Additional features which recommend UNIX as a software development environment for a user interface embodying geographical knowledge are the prominence of at least appeals to cleanliness and readability in programming within the UNIX community. Two approaches are of interest here, firstly literate programming. This has been implemented for C and *troff* by Thimbleby (1986), and as with Knuth's *WEB*, promotes the exposition of the reasoning behind coding choices in a discursive way. On the other hand, even the most elaborate comments cannot ensure that the code corresponds to the programmer's intentions.

The second approach of interest is through languages giving support to object-oriented programming and simulation. There are good reasons to suppose that dioramas may benefit from a profiling of programming work towards the definition of prominent landscape and traffic components. This might suggest that Smalltalk was a place to start, but the same arguments would point to C++, given the paucity of our knowledge of relevant classes.

5.2 Production-quality systems

Since there is a direct trade-off between image quality and processing power/screen characteristics, it is reasonable to expect pre-recorded dioramas to exceed interactive ones in visual impact. A certain sketchiness in pictorial representation may however be worth retaining in order to avoid burdening the user with detail superfluous at the level of resolution currently chosen. It is necessary to maintain awareness of the way in which we process information received visually (cf. Kosslyn, 1985), and of the ease with which humans read "pattern" into truly random distributions (Openshaw, 1987).

A model which has some attraction is to assign at least one parallel processor to the backcloth, and one to each traffic component, thus allowing them to assume their autonomy, communicating in message form when required. In practice however, an adequately powerful processing engine would be able to cover the same functions. The key features which a user interface modelled on a diorama would have to meet would include windows, for the diorama

itself, perhaps at different scales, and for access to the structuring rules. The interventions in these structures are best written in a command language, with alternative shells for users of differing persuasions and aptitudes.

Although UNIX offers a superior development environment for such a system, it is at present difficult to see how far portability would be prejudiced by graphics terminal heterogeneity. Given that *tty* variety is handled well in UNIX and supported in libraries like *curses*, and that groups like X/OPEN say a lot about common applications environments, this is possibly a problem in retreat. On the other hand, 7-bit ASCII is several dozen orders of magnitude easier to handle, as the move to 8 bits is showing.

Does a production-quality diorama need to run on a multi-user system? The "Domesday Machine" stands alone, with its laser disc store holding enough data to occupy an attentive user for seven years. The store holds up to ten copies of some maps at different levels of resolution, to save aggregating data on inadequate processors. An argument for multi-user systems is sharing of large data stores, performed in the cited case by their replication. If one wanted to do something else with the data than simply display from store, for instance change the structuring rules, then the advantages of shared storage might return. The advantages of multiple linked users would increase if accessible processing power was such that the large quantities of data associated with the diorama could be handled within acceptable time limits.

In summary, the role of UNIX in running rather than developing a geographical user interface based on dioramas would hinge on the overheads introduced by not bypassing the operating system in graphic display as on stand-alone microcomputers. Since a geographical user interface cannot expect to be resourceful enough to command networked workstations on grounds of cost, at least in the forseeable future, it is not easy to draw any very optimistic conclusions.

6 Conclusions

Recent thinking in geography has pointed to the importance of modelling system interaction in the form of dioramas composed of a backcloth or landscape upon which traffic or actions/projects are superimposed. The systems of interest vary in spatial and temporal scale, but include phenomena often compartmentalised by the segmental disciplines. Representation of such wholes in textual form presents problems of substantial scale for geographers, as for historians and others who wish to set their "stories" in contexts. Even in textual form, the "story" is no unprocessed depiction of reality, but relies on an elaborate structure of data and rules. The posited advantage of the diorama for examining complex ensembles "inhabiting" a place at a time—what Hägerstrand has called *thereness* — is the inherent character of the system interactions presented. However, this also requires the formulation of the subset of all possible interactions judged significant in generating the outcome.

The software tools that UNIX incorporates cover most of the requirements present in the modelling module of the diorama interface. This suggests that prototyping of the structural frames may be advanced by adopting a common development environment. The experience of users of the UNIX system in the production of animations certainly holds promise for the realisation of dioramas like slide-shows or dramatisations. Problems however arise in relation to questions of cost and speed in handling the graphics windows required to give an interactive diorama life.

References

[1] Bennett R J, Haining R P, Wilson A G, 1985 "Spatial structure, spatial interaction, and their integration: a review of alternative models" *Environment and Planning* A 17 625-645

[2] Bentley J L, 1986 "A literate program" *Communications ACM* 29 471-483

[3] Bentley J L, 1987 "Profilers" *Communications ACM* 30 587-592

[4] Bivand R S, 1984 "Regression modelling with spatial dependence: an application of some class selection and estimation methods" *Geographical Analysis* 16 25-37

[5] Gould P, 1981 "Letting the data speak for themselves" *Annals of the Association of American Geographers* 71 166-176

[6] Griffiths P, Hill I D, (eds) 1985 *Applied statistical algorithms* (Chichester, Ellis Horwood)

[7] Hägerstrand T, 1982 "Diorama, path and project" *Tijdschrift voor Economische en Sociale Geografie* 73 323-339

[8] Hägerstrand T, 1984 "Presence and absence: a look at conceptual choices and bodily necessities" *Regional Studies* 18 373-380

[9] Kosslyn S M, 1985 "Graphics and human information processing" *Journal of the American Statistical Association* 80 499-512

[10] Openshaw S, 1977 "A geographical solution to scale and aggregation problems in region-building, partitioning and spatial modelling" *Transactions Institute of British Geographers* NS 2 459-472

[11] Openshaw S, 1987 "An automated geographical analysis system - guest editorial" *Environment and Planning* A 19 431-436

[12] Openshaw S, Rhind D, Goddard J, 1986 "Geography, geographers and the BBC Domesday project" *Area* 18 9-13

[13] Openshaw S, Wymer C, Charlton M, 1986 "A geographical information and mapping system for the BBC Domesday optical discs" *Transactions Institute of British Geographers* NS 11 296-304

[14] Thimbleby H, 1986 "Experiences of 'Literate programming' using cweb (a variant of Knuth's WEB)" *Computer Journal* 29 201-211

[15] Tobler W R, 1970 "A computer movie simulating urban growth in the Detroit Region" *Economic Geography* 46 234-240

The HUB: A Lightweight Object Substrate

Michael D. O'Dell

Maxim Technologies, Inc.
Vienna, Virginia USA

1. Introduction

"Lightweight Processes" are currently *très chic* in the leading-edge UNIX community, and "Object Oriented Programming Systems" (OOPS) are gaining in popularity. Both of these ideas may be fundamentally sound, but seem to be suffering somewhat at the hands of hypesters promoting them as the cure for everything from *designus inconceivus* to tertiary code-bloat. Combining these two hot topics is sure-fire way to get funding (and a paper accepted).

The Hub is a simple little operating system which is rather different from most of its process-based brethren seen running about. Like with most systems, these differences characterize both its strengths and weaknesses. The Hub has activities somewhat like processes, called tasks (sorry, but there are only so many words for these things), but it doesn't have the overhead of traditional context switching. Hub tasks communicate primarily with messages, but they don't incur all the message parsing and multiplexing overheads of other systems. The Hub could use a bit of linguistic support which it doesn't get from most programming languages, but the C preprocessor and a modicum of programming discipline largely fill the gap. The computation model of the Hub is somewhat non-traditional to folks raised on the beatitudes of pure processes. And finally, because the basic design doesn't deal with protection domains, the Hub is a *Scout's Honor* programming environment like most little operating systems, and it will run perfectly well within a UNIX process for debugging and development, or as an implementation of user-level "light-weight processes."

It also turns out that the Hub uses some of the central OOPS ideas, admittedly in crude forms, but it is, none the less, an interesting platform for supporting, shall we say, *object-inspired* programming. Hence the title: mix lightweight processes with object-oriented glue, and you get a substrate which can support objecty kinds of things, assuming they don't weigh too much.

2. What's All the Hubbub, Bub?

The Hub was created by Gary Grossman, then with the Center for Advanced Computation at the University of Illinois. The design was first implemented and described by Masamoto [MAS] for his Master's thesis under Grossman's direction. The current implementation is inspired by the system described in Masamoto's thesis, but revised considerably for portability, generality, and more concern for memory management issues.

There are three major elements of the Hub world: the Hub Queue, which gives the system it's name, Hub instructions which are placed in the Hub Queue, and Hub tasks which execute the instructions (see Figure 1). The sequence of operation is really quite

simple: it mirrors the traditional fetch-execute cycle of a computer CPU.

The Hub dispatch loop (the Dispatcher) removes the next instruction from the Hub Queue and inspects it to determine the recipient task and which specific task entry-point should be invoked to execute it. The Dispatcher then makes an indirect subroutine call via a task-specific methods vector supplying the decoded instruction fields as arguments. The task method then runs to completion, adding any necessary new instructions to the Hub Queue. When task method returns, the Dispatcher goes back to the top of the loop to fetch the next instruction waiting in the Hub Queue.

2.1. Instructions Pre-Fetched While You Wait

The Hub Queue functions very much like the instruction prefetch buffer used in modern CPU designs. Instructions which form the next fragment of the instruction sequence but are not yet executing wait their turn in the prefetch buffer, and likewise in the Hub Queue. Where do the instructions come from? In CPUs, the relentlessly advancing program counter marches new instructions into the prefetch buffer. In the Hub system, the stream of new instructions arises from the execution of other instructions! One of the most important side-effects of interpreting a Hub instruction is adding a new instruction or two to the Hub Queue, for without a steady supply of new instructions, *nothing* happens in the Hub world.

Tasks communicate by placing instruction for each other in the Hub Queue. The only way a task ever executes is for an instruction destined for it to bubble to the top of the Hub Queue and be dispatched. For example, instead of the classical *sleep()/wakeup()* process model, an interrupt routine in a device driver notifies its client "top half" of service completion by placing an instruction destined for it in the Hub Queue. With only very minor exceptions, state information is never shared directly between autonomous execution domains (this includes both other tasks and interrupt code). Instead, instructions carrying the necessary information are queued for execution by the recipient. This results in very infrequent processor interrupt lockout, but most importantly, the entire system is manifestly observable, if not truly synchronous. All activity flows through the Hub Queue and the Hub Dispatcher, so one need monitor only that one simple point to produce a very complete picture of exactly what is happening in the system. With a little planning, it is even possible to record and replay instruction sequences to assist in analyzing behavior.

2.2. Detailed Instructions

Hub instructions specify an opcode and several operands: source and destination task identifiers, source and destination port identifiers, and a general-purpose operand which is usually a pointer to a message buffer (see Figure 2). (There can be several general-purpose operands, but rarely is more than one used.) The opcode is equivalent to the method selector of languages like Smalltalk; it identifies which task entry-point (method) is to be executed to interpret the instruction. This saves multiplexing and demultiplexing in several ways. First, we avoid wasting the time spent assembling and then parsing explicit message headers which simply indicate the requested function. Second, decoding the instruction is very straightforward and very fast, and it only need be optimized in one place. Third, task entry-points tend to do only one thing so the path length is minimized and the code simplified.

The source and destination task identifiers in Hub instructions are used to create the *sender* and *self* references used for executing task entry-point methods. A task identifier is a handle to the task's Task State Vector (TSV) as it is called, and tasks are known by their TSV handle. A handle to the new TSV is returned to the parent task when a new task is created.

The source and destination port identifiers are small integers which are available for further multiplexing and demultiplexing within a method. In general, they are simply integers and can take on any such value. Their name, however, arises from an array of port structures in every task's TSV, and port values usually index this array. A port is simply a header for a doubly-linked queue which is included in the TSV port array. These are included in the basic TSV overhead because TSVs often need queues, and having a queue designation encoded in the instruction often prevents reinvention of mechanism and improves observability.

Arbitrary message data can be carried in an instruction by incorporating a reference to it in one of the general-purposed operands. This tends to avoid data copies and provides for quite general messaging. The messages carried by Hub instructions are usually said to flow from the source to the destination port of the participating TSVs.

2.3. Tasks - the Functional Units of the Hub

Tasks are the unit of execution in the Hub system. A task is represented by its activation record, which is called a Task State Vector, or TSV. It is common for "task" and "TSV" to be used somewhat interchangeably, although it isn't strictly correct. Associated with each TSV is a collection of functions called the Task Program (TP) which the task executes in response to instructions. Each such function is called a Task Program Entrypoint, or TPE. Mapping between the instruction opcode and the implementing TPE function is done with an array of function pointers stored in a generic section of the TSV. This binding allows task programs to be shared between tasks even if the code is not strictly reentrant.

The Task State Vector contains two sections: a generic common prefix and a data area specific to the Task Program being executed by the task. The TSV prefix contains the port vector, vectors to the TPEs, and a few other miscellaneous fields. The TP-specific data area is of variable size and is specified when then TSV is created. The *entire* state of the task is represented by data stored in this section of the TSV.

To summarize in object-speak, the Task Program is the methods collection for the object implemented by the task, and since the task program is shared, multiple object instances can be implemented which all share the same methods. Instance variables used by the implementing methods of an object are stored in the object-specific portion of the TSV. Class variables can be implemented (crudely, to say the least) by global variables, or static variables within functions contained in shared TPE code.

The Hub Queue and the Hub Dispatcher, Hub instructions, and Tasks are the basic inhabitants of the Hub world. The way these parts interact to do computations creates a programming environment which is both familiar and strange at the same time.

3. Not Entirely Unlike Processes

This section will examine the programming style which arises from the Hub's unique structure. As can be discerned from the description so far, systems based on the Hub are

composed of a collection of tasks which communicate with each other via instructions, and these tasks implement higher-level abstractions, either abstract data objects or compute objects like protocol machines. In point of fact, the Hub was born to do communications processing. This is a programming area classically described on the blackboard by clouds of processes blithely exchanging messages willy-nilly across beautifully abstract interfaces. The implementation, on the other hand, is usually quite different because of real-world performance requirements. One is easy to understand, the other runs fast enough to be useful. The Hub is an attempt to assuage the disparity between the pictures and the code.

Unmentioned until now is the central notion that all tasks in the Hub world obey essentially the same set of legal Hub Queue instruction opcodes. This makes sense when one reviews the structure of most communications software implementations, particularly those based on the traditional picture with processes. Tasks primarily exchange data with one task on one side, and do essentially the same thing with another task on the other side. The exceptions to this are device drivers, which talk to hardware on one side, but are tasks when viewed from the other, and multiplexing tasks which communicate with potentially many other tasks.

Each Hub Task Program is expected to implement the following basic instructions most appropriately for the function provided by the specific TP.

- **Initialize** - an instruction sent to a newborn task so it may initialize its instance variables before it receives any other instructions. The final act of this TPE is to call a Hub function which acknowledges the initialization. Sending instructions to a task before it has acknowledged the initialization is considered a serious error.
- **Die** - the task should clean up whatever it was doing and commit suicide by calling the appropriate Hub function. Any shutdown synchronization between the requestor and the dying task is purely their business.
- **Timer** - this instruction is posted to a task to notify it that a timer event has expired. The arguments in the Timer instruction are specified when the timer event is scheduled with a Hub primitive function.
- **Data** - this instruction is the basic data transfer mechanism. The argument usually points to a buffer containing the data to be transferred to the destination TSV and port. The usual protocol is that ownership of the buffer is also transferred and becomes the responsibility of the recipient. Note that this is essentially a *write()* function to a task expecting to receive data.
- **Datarequest** - this instruction is used to request a source to send data to the originator. This is essentially a *read()* request from a task desiring data. This instruction may or may not be used in all cases depending upon the flow control protocol between tasks, and the level of asynchrony between the tasks. Packets arriving from a network would typically be posted to an IP protocol machine with a Data instruction, while a task like an FTP server might use a Datarequest instruction when requesting data from its underlying TCP task.
- **Control** - a task-specific control function (like *ioctl()*)
- **Poll** - an instruction for implementing polite busy waiting. Typically, a task waiting for a bit to change in an interface would do the polling by posting a Poll instruction directed to itself. When the Poll TPE gets entered, it checks the appropriate bits and decides whether to post another Poll instruction, or whatever

instruction is appropriate to continue what it was doing. This provides the simplicity of busy waiting without hogging the machine.

- **Debug** - a task-specific function which manipulates debugging state. Hub systems use some internal protocol for determining how the debug state is interpreted, but there is a common mechanism for manipulating it. This encourages useful debugging machinery be included in every TP.
- **Private** - there are two instructions included which are valid instructions, but whose interpretation is local to each TP. This number is easily changed for whatever is needed.
- **Default** - this is not an instruction, per se, but is a TPE which is entered like an instruction whenever an opcode is not one of the above. (The name arises from the C case statement.)

Each Task Program Entrypoint is called from the Hub Dispatcher with two arguments: a pointer to the private area of the appropriate TSV (a *self* context), and a pointer to the instruction which caused this TPE to be executed. This has the unfortunate side effect of requiring local instance variables be addressed with something like

self –> localvariable

and is one area where a little language support would be useful. In Pascal, the *with* clause would do the trick; in C, a few preprocessor **#define**s reduce the pain.

The instruction pointer is supplied so the port values and sending TSVid can be ascertained, as well as to pick up any message buffer pointers in the general-purpose operands.

As can be seen, the Hub is non-blocking. There is no context switching code necessary, and almost no machine-dependent assembler is required, save for that necessary to glue interrupts into the require device driver functions, and for the two functions which set and return the machine's interruptibility state (equivalent to *spl()* and *splx()* in the UNIX kernel). This implies the Hub needs only one stack segment for execution. While a separate stack for interrupt routines an advantageous of some modern processor architectures, there is no requirement for any such support, and it can probably be readily exploited.

4. The Structure of a Hub System

Hub systems tend to be be constructed from "task teams" with one task managing the activity of several worker bees. Manager tasks tend to communicate with each other to create any needed worker tasks and to establish the plumbing between them, leaving the workers to do the actual work. This is much like the *daemon daemon* of 4.3BSD. In an X.25 system currently being built with the Hub, a Level 2 interface manager task watches link devices for signs of life. When the link starts up, the Level 2 interface manager contacts the Level 3 protocol manager with a request for Level 3 service. The Level 3 manager responds by creating a Level 3 protocol machine task and works with the Level 2 manager to arrange the rendezvous between the Level 2 protocol machine and the Level 3 machine. After the initial introductions, the two protocol machines interact with each other and only interact with the managers when some terminating condition

arises.

One important issue in any operating system is flow control. Flow control in this context means the procedures and protocols which constrain the amount of data the system must buffer at any one time. Several recent systems provide for synchronous interactions between processes, thereby rendering the flow control issue essentially moot. However, the disadvantage of this simplification is that the maximum possible concurrency seems to be somewhat limited.

Flow control within a Hub system is visible at two levels. Topmost is the issue of how many Hub instructions are currently awaiting execution. A runaway task which, for example, queues two instructions for each it receives could quickly exhaust the Hub Queue if it proceeded unchecked. This problem is handled by approaching the design with a resource "conservation law" in mind. In other systems, violations of the conservation laws result in bugs like memory leaks, or occasionally freeing an already free resource. The nature of the Hub encourages explicit establishment of such laws as part of the design process (this is both a strength and a weakness).

The other flow control issue is one of how message flow is mediated between tasks. The Hub system provides a modicum of queuing via the instructions in the Hub Queue. Again, a central notion of Hub execution is that a task must always do *something* with a message when it arrives. This is where the port queues come into play. Often, a task can't really do anything with a message because it algorithmically can't proceed; a closed TCP transmit window is a good example. The port queues can be used to sit on the data until the task can dispatch it.

In such a scenario where there is potentially a large impedance mismatch between a stream producer and a stream consumer, an explicit flow control strategy must be used. For example, after a producer sends a buffer in with a Data instruction, it must await a Datarequest instruction from the consumer returning the previously-sent buffer to be refilled. By simply changing the number of allowable outstanding Data instructions to be greater than one, we can easily implement sliding-window style multiple buffering between tasks for bulk throughput applications needing maximal concurrency. In other situations like an Ethernet driver sending packets to an IP protocol module, the interface may not be flow-controlled at all, relying on a simple policy of dropping excess packets upon an overflow condition. The important point is that the level of sophistication needed by any particular interface can be crafted from the available Hub facilities, thereby neither overbuilding for some uses or underbuilding for others.

In concluding the discussion of the Hub environment as seen from inside, it should be said that the Hub was intended to be a flexible framework for implementing what is needed to do the specific job at hand. It has a modest set of facilities beyond those described like timer management and buffer and storage allocation which, when taken with the Hub facilities described above, form more of an operating systems toolkit, rather than an ornate edifice replete with strictures. This is, of course, a double-edged sword.

5. Closing Notes

There are two other areas which deserve comment, one because it was advertised, and another because it is interesting to consider in light of the current architectural trends.

5.1. Some Thoughts on Implementing Objects

I hope by this point the notion of using the Hub to implement objecty systems, possibly hidden behind some syntactic overcoat, is not completely absurd. There are more than a few problems if you want real Smalltalk, but with a modicum of reserve, the Hub can go a reasonable job of supporting object-style programming done in a more traditional programming language. Note that this opinion arises from the belief that the most important feature of object-oriented programming is one of encapsulation, with limited, static inheritance far short of the Smalltalk extreme probably being quite adequate to realize the advantages of OOPS for most tasks. Others will certainly differ, probably strongly. That's what makes horse races.

The other topic has to do with the notion of context switching in general and the impact of evolution in machine architectures on the basic complexity of this mechanism which is so central to traditional process-rich systems.

5.2. Context Switching and RISC Machines

The traditional process context switch involves saving the "visible" processor state (register values, condition flags, interrupt level, floating point modes, memory management state, etc.), and reloading the processor with a new copy of this same information. In its full glory, a great many bits move around, and many, many machine cycles can go by if the state is very complex.

RISC architectures are often characterized by large register stacks on the processor chip which must be loaded and unloaded on context switches. These stacks are generally much larger than those of their CISC friends and it is interesting to ponder the performance impact of heavy context switching upon RISC performance. One useful description is that a process-based system tends to be "broad," meaning it spans many different state domains (many relatively shallow stacks), while the state of a RISC machine tends to be "deep," meaning it excels at nested subroutine calls within one state domain (stack). Since the Hub systems needs only one stack, and spends all its time doing subroutine calls, it may be particularly suited to RISC architectures. This is an interesting area to pursue experimentally with actual measurements!

6. You Have Tea and No Tea

In conclusion, the Hub is both traditional and radical at the same time. It is both process-like and thread-like. It contains some objecty notions, and some explicit programmer responsibility for managing the environment. Maybe the way to see the landscape is as a continuum between process-richness at one end, and traditional monolithic realtime multi-threaded systems at the other, with the Hub as an operating point somewhere in between. Traditionally, a *process* is a defined, fabricated object which, like Algol, everyone understands but can't quite nail down. With the advent of Objects, which seem to be somewhat like processes in their persistence and activity, but which somehow don't really satisfy the intuitive definition of *process*, maybe *process-ness* has become an analog value which can be possessed in greater or lesser degrees instead of being an absolute attribute. This is certainly in the spirit of the Hub, and possibly places the Hub at the confluence of Object-oriented programming concepts and operating systems.

7. Bibliography

[MAS] Masamoto, K., "Implementation of HUB Processor," Master's Thesis, University of Illinois at Champagne-Urbana, 1976

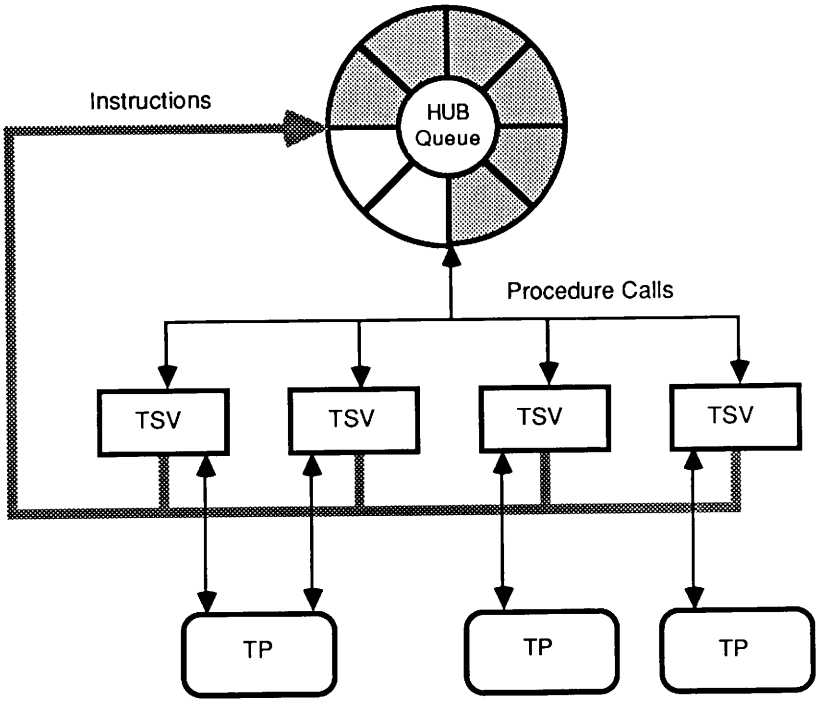


Figure 1
The HUB System

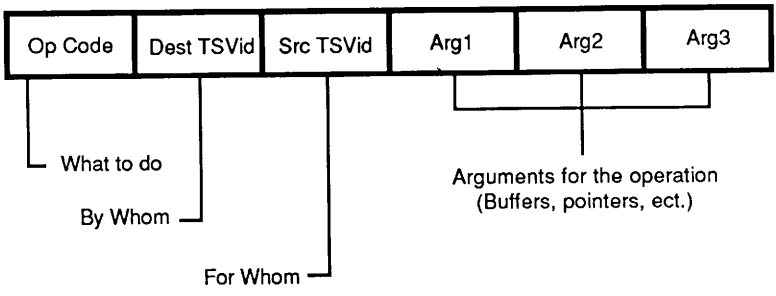


Figure 2
A HUB Instruction

More MIDI Software for UNIX

Michael Hawley

MIT Media Lab[†]
Cambridge, MA, 02139
mike@media-lab.mit.edu
August 8, 1987

1. Introduction

At the Media Lab we have configured a 25Mhz Sun-3[‡] to control 64 channels of MIDI synthesizers in real time. We also intend to control the ultimate player piano – a computerized Bösendorfer Imperial concert grand, a glossy, impeccable 9'6" instrument which records performance data using optoelectronic sensors and plays it back using a stack of solenoids. The imperial Bösendorfer is generally considered to be the Rolls-Royce of pianos – the "standalone" (i.e., non-computerized) version costs in the neighborhood of us$60,000 – so this is quite a lavish setup.

Persuading such machinery to render synthetically accompanied piano concerti or ensemble music is a seductive job and will consume much more time in hardware and software work. Of course, software that can take full advantage of such wonderful music hardware is a long way off, and it will continue to be a pleasure to evolve it. But to do this one also needs insight into what factors make a performance entertaining, and the fresh influx of high-quality performance data has sparked some ideas for music analysis software.

For example, the command

```
record | kee
```

records performance data from a musical keyboard and returns its key (e.g, "a minor"). The trick to this is not deep AI or even expert musical hackery (like looking at cadential pitches, or melodic interval analysis) but rather, the simplest conceivable statistic: a note-counting program counts all the pitches in a performance and looks at the histogram to pick the key.

Rhythmic analysis is more interesting, and closer to human perceptual issues. Consider what it means to listen to some piece of music, like a fugue – tracking the separate voices, recognizing each voice entrance, etc. Two crucial parts of the process are *finding the beats* (e.g., being able to count to three in a waltz) and *auditory streaming* (hearing the parallel voices as distinct streams). These are easy for people to do – as easy as tapping to the beat, or following the clarinet and soprano lines at the same time – but no one seems to have computed good solutions yet. At the MIDI data level, streaming turns out to be fairly easy to do (algorithmically) most of the time: simply graph the MIDI data in a 2-d pitch-time space and "connect the dots" by clustering the onsets of notes that are closest together. The graphical piano-roll views of data cluster into visual blobs almost as readily as they do in the audio sense. By a similar kind of clustering in

† Current address: NeXT, Inc, 3475 Deer Creek Rd, Palo Alto, CA 94304

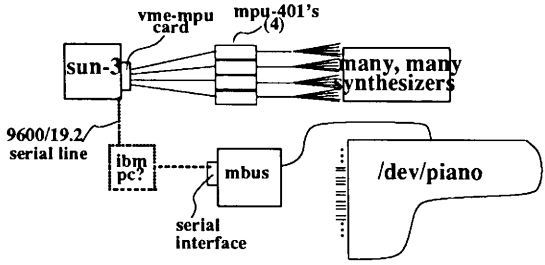
‡ Thanks to Sun Microsystems, Inc, for this donation.

the 1-d time domain, we can chunk strings of notes which are usually perceived as phrase groups by human listeners. For interesting musical recognition problems, like looking for tunes in a database of musical themes, recognizing pieces or styles, or recognizing repetitions of a melody within a piece, we think that the intersection of a few simple statistics like these will give us sufficient precision to find good matches.

In what follows we will briefly describe the machine configuration, and discuss the analytical tools we have developed so far for studying the data.

2. Hardware and Software

Our hardware setup is a somewhat enhanced version of that described in [1,2]:



hardware configuration.

The Sun contains a VME/MPU interface card which allows it to control four real-time MIDI processors (Roland MPU-401's). These appear as `/dev/mpu[0-3]` in software, and the utilities described in [1,2] will all run in this environment. Because they are addressed as separate devices (i.e., four file descriptors) synchronization of all four is likely to be a mild problem. Each MPU has two MIDI plugs for output and one for input. There are ways to make the devices clock to a "house sync," but we haven't yet explored them. Since each MPU can theoretically transmit 16 channels of MIDI output (a channel is represented as a 4-bit mask) while reading 1 channel of MIDI input (as from a performer at a keyboard) 64+4 i/o channels should be possible. We haven't yet assembled the performance software or synthesizer arsenal to take full advantage of this; an awful lot can be done with 16 instruments.

As i/o devices go, communicating with the Bösendorfer is a little like negotiating with a finicky prima-donna. At this writing, it is not yet possible to `open("/dev/piano",2)`, because there is no hardware connection to the Sun. The piano is unfortunately not a MIDI device – it has a somewhat higher data rate, and greater precision (e.g., 16 bits of velocity data instead of 8). The multibus cardcage houses some Z-80 cards which drive an arcane CPM operating system, including some strange floppy disks and a tape-recorder i/o port, but no modern data port of any kind. This of course would be a criminal offense in certain parts of California. Kimball (which owns Bösendorfer) has recently built an IBM-PC-based serial interface to the instrument which provides data at 19.2 kbaud (a barely adequate data rate, though sufficient to deal with most human playing, except perhaps Art Tatum on a good day.) We expect to have this hardware early in September, so some form of piano i/o should be possible soon. The problem of house synch is looming for the piano, too; eventually, all real-time controllers must clock to a common synch.

The graphics software is currently written using the library described in [3], which is adequate and expeditious for our needs.

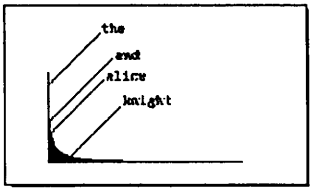
3. Simple Statistics for Music

Relatively little general statistical work has been done on performance-level musical data, mostly because that data has been so scarce until fairly recently. To begin with, we will use a few simple statistics to try and locate the most salient features from the data. First we'll look at a method for finding *key* words in text based on relative frequency, and show how, as simple as that idea may be, it does a good job of hilighting the most important topical words in a document.

3.1. Counting English

Many people have studied the word-frequency distributions of natural languages (George Zipf and Claude Shannon are prominent among them). This is usually done with an eye towards understanding the predictability and redundancy of the language. Zipf noticed that, in English and many other statistical populations, word frequencies are distributed in a `1/n`–like curve, with the commonest words (e.g., "the", "and", "of", ...) taking up the fat end of the curve:

```
% cat MobyDick/* | hist | histo
```



a typical word histogram.

Here, the `hist` program reads text and produces a sorted word count:

```
the 16037
of 8239
is 6325
 ...
```

and `histo` takes a stream of *label/value* pairs and plots them in a window.

To identify the most important terms in a document a good approach might be to filter out the function words and all other vocabulary words which appear with roughly average English frequency. The command `pword` does this: it computes an *index of peculiarity* for every word in a document, and prints out the list in descending order. The index is

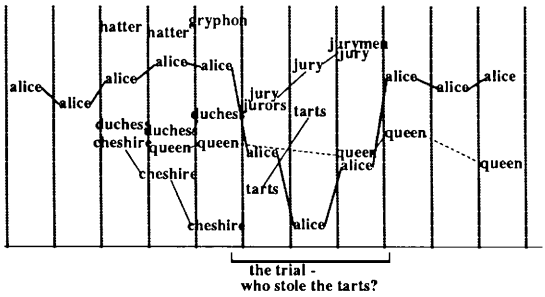
$$f_i^2 / f_g$$

where f_i is the frequency of the word *within* the document, and f_g is the *global* frequency of the word in some larger domain (common english). This prioritizes those words which are used relatively often in a document, and it turns out to be quite a good "important word" finder considering its simplicity. The peculiar word listings for *Alice In Wonderland* and for UNIX `man` pages do seem to capture the most important features:

```
% cat AliceInWonderland/* | pword
190.51 alice
 39.86 queen
 27.18 glove
```

```
 14.37 tart
 12.36 hatter
 11.90 gryphon
 11.49 humpty-dumpty
 11.20 tone
 11.13 turtle
 11.01 daresay lory tove
 10.06 rabbit
  8.75 dormouse
  8.08 wabe whisker brillig knight
  6.81 mock
  6.27 king
  5.99 duchess
  5.90 tweedledum
  5.82 remarked
  5.72 caterpillar
  5.61 teacup sneeze hookah mimsy juror telescope borogove outgrabe gimble...
  4.96 kitten
  4.40 tweedledee oyster
  3.59 mabel
    ...
% deroff /usr/man/man1/cat.1 | pword
745.30 concatenate
477.70 cat
 27.77 display
 14.01 character
 12.90 blank
 12.39 non-printing
  6.69 file
  6.64 standard
  6.26 output
  2.97 input
  1.07 number      (e.g., "line numbering")
```

`Pword` can optionally slide a sampling *window* of some size over the document. This shows how the priority of important terms changes over time, producing a chart that mirrors the storyline:



story lines in *Alice In Wonderland*

This graph, with a window size of 30Kb moving at 15Kb increments, shows how Alice figures prominently throughout the book, with a dip in the chapter in which she is put on trial for stealing the tarts (and not

allowed to speak). Other topics can be seen as they come and go throughout the book.

3.2. nc, kee, nv

As with English text, we can use very simple statistics to find some of the important features in music performances. For instance, to determine the musical *key* of a piece, we'll simply look at the histogram of pitches. The note-count program, nc, produces a count of the pitches in a MIDI file:

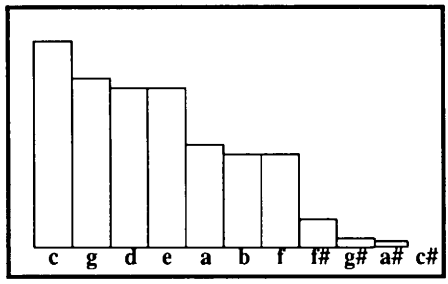
```
% nc bach/2part-inventions/i.01
 C2  36      1    311
 D2  38      1     53
 G2  43      4    261
 A2  45      3    202
 B2  47      4    276
 C3  48      7    605
 D3  50     13    585
 E3  52     16    724
 F3  53      6    266
F#3  54      8    477
 G3  55     18    901
G#3  56      2     77
 A3  57     22    965
A#3  58      3    138
 B3  59     20   1101
       ...
```

The listing shows, for each key that is struck, its symbolic name, its index on the MIDI keyboard, the number of times it was played, and the total duration (in *ticks*) the note was held. Folding all the pitches into one octave and sorting by duration,

```
% nc -m bach/2part-inventions/i.01 | sort +3nr
 C   0     68   4029
 G   7     69   3555
 D   2     72   3449
 E   4     69   3453
 A   9     65   2685
 B  11     58   2649
 F   5     48   2614
F#   6     17    853
G#   8      7    299
A#  10      7    282
C#   1      4    251
```

we see that, sure enough, Bach's first two-part invention is predominantly in C major (CG...E), and the next most reasonable choice would have been the dominant, G major (...GD...B). The histogram shows this:

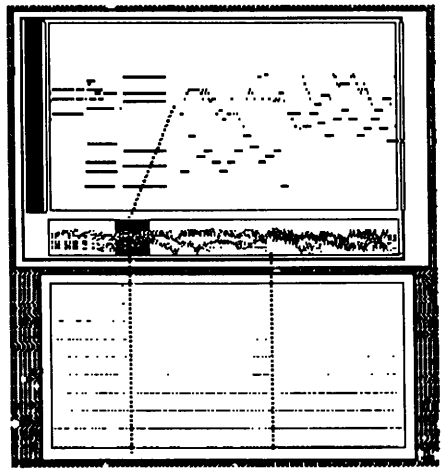
```
% nc -m bach/2part-inventions/i.01 | sort +3nr | awk ... | histo
```



pitch distribution in the C major two-part invention

For simple tonal music, pitch histograms are generally more crisply descending than this. Atonal pieces (e.g., much of Schoenberg) tend to produce flat distributions and some pieces, like Joplin rags, begin in one key and end in another (e.g., *Solace* begins in `C` major but ends in `F`, producing a bi-modal distribution in which the first choice is `F` – the ''correct'' key – and the second choice `C`). Because knowing the key of a piece (or whether or not it *has* a key) is important, we replaced the `nc` pipeline by a single program, `kee`. Of course, just as we slid `pword` over *Alice In Wonderland* to chart plot developments, we can slide the `kee` program over an entire MIDI file to locate the major tonal regions.

Another useful statistic is knowing the number of active voices, which gives a good indicator changes in texture. The program `nv` does this. Here is a view of the first movement of Bach's c minor partita, showing the piano roll aligned with a trace produced by `nv`:

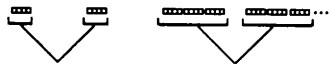


voice texture of Bach c minor partita, aligned with piano roll

In the scrollbar of the piano-roll view we can see that the piece has three main sections: an opening 6-voice chordal section, a florid middle section with melody and walking bass, and a final fugal section. These major structural divisions correlate nicely with blips in the `nv` trace.

3.3. Clustering and Streaming

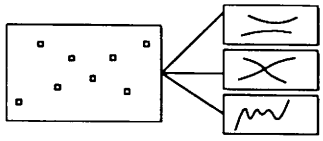
By *rhythmic clustering*, we mean collecting events that are proximate in time into a single group. At the simplest note level, this means finding attacks that are close together and making a group out of them. Consider the opening of Beethoven's 5th, reduced to rhythm:



rhythmic groupings in opening of Beethoven's 5th

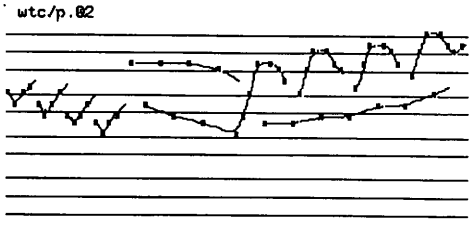
A rhythmic clustering algorithm will find four main groups here, which turn out to be the main utterances in the opening phrase.

Streaming is similar to rhythmic clustering but occurs in 2-d pitch/time space. A sequence of events may decompose into several streams depending on their relative spacing (e.g., how fast they are played). So a zig-zagging melody may be heard as a single wobbly stream at a slow tempo, but as two separate streams at a faster tempo:



several possible streamings

This can be a difficult problem in general, not unlike line recognition in vision. For instance, correctly following voices that cross may require knowing something significant about the context – like lots of important melodies – in order to stay on the right track. To help study rhythmic clustering and voice streaming we've written a graphical program which displays MIDI data and attempts to segment it into sensible streams. We have a database of many melodies and fragments, including all the melodies from the *Well-Tempered Clavier*. The following shows how the program might stream a performance of the c minor prelude from the second book:



c minor prelude, *wtc/II*, showing melodic streams

A crucial measurement is the *distance* between two note onsets: the proportion of temporal and pitch units determines how the notes will cluster. Also it is worth noting that two notes an octave apart may actually be "closer" than two notes a major 7th apart, because the octave is harmonically closer, and because octave leaps occur much more frequently than leaps of a major 7th. Thus it is probably desirable to weight

pitch distance by the frequency of interval occurence (and this can of course depend on whatever has recently been heard). We have not explored this yet.

4. Conclusions

We have presented a variety of simple analytical programs for finding some salient features in music. Most of this work was done in a few weeks during which our hardware and software coalesced, and we are looking forward to incorporating Bösendorfer data as well. The simple UNIX tool-building approach is a good way to focus on particular aspects of the problem, and it appears that a program which uses techniques like these (rhythmic clustering, melodic streaming, key sensing, voice texture, etc) should be able to recognize melodies, compositions, and perhaps compositional styles given enough training. However, one missing link is the lack of any felicitous way to implement a set of simple parallel parsers which inform each other. For instance, group-boundary suggestions which derive from key and voicing information should strengthen each other and make it easier to know "where to look" for important events.

Our analytical work is just beginning, and likewise, the hardware is just coming up to speed. We look forward to many entertaining interactive applications in the near future.

5. Acknowledgements

Sun Microsystems generously provided a full Sun-3 system, making much of this work possible. Daniel Steinberg and Don Jackson (both at Sun) adapted older multibus MIDI boards and drivers [1] to the VME/multi-MPU world, and kindly forwarded their knowledge to us. Dave Cumming (at MIT) built our VME board, cleaning up the design somewhat. Advisors Andy Lippman and Marvin Minsky at MIT haven't complained about the noise yet.

6. References

[1] Michael Hawley, **MIDI Music Software for UNIX**, *Usenix Summer Conference Proceedings* (1986)

[2] Peter Langston, **201-644-2332, or Eedie and Eddie on the Wire**, *Usenix Summer Conference Proceedings* (1986)

[3] Michael Hawley, Samuel Leffler, **Windows for UNIX at Lucasfilm**, *Usenix Summer Conference Proceedings* (1985)

Automatic Exploitation of Concurrency in C: Is It Really So Hard?

Lori S. Grob

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

August, 1987

ABSTRACT

The vast majority of work being done today on the automatic exploitation of concurrency, for both multiprocessors and vector machines, is not being done for C. Yet there are many important applications, written in C, which would benefit immensely from the speedup produced by such techniques. Many more applications would be written in C were there optimizing, vectorizing and parallelizing compilers for the language. Scientific and numerical computing is largely done in Fortran and so the bulk of the advances in both optimization and automatic exploitation of concurrency are done on Fortran compilers of various flavors. So a vicious cycle has developed. Even a company as *avant garde* as Thinking Machines has found it necessary to start a Fortran group in order to broaden their user base. Yet much work that is currently done in Fortran would be done in C were there efficient compilers. The issues involved in writing an optimizing C compiler are largely the same as those involved in writing a compiler or preprocessor that will automatically exploit concurrency.

In this paper we consider whether it is possible to have compilers that will automatically exploit concurrency in C. We discuss the relationship between automatic exploitation of concurrency for the purposes of vectorizing and multiprocessing. We review the basic techniques and transformations used and examine the necessary conditions to perform these transformations, with examples in C. Several elements of the C language and programming style, such as pointers and recursion, make it difficult to do the necessary data flow analysis. There are two possible approaches to this problem: to bypass code or blocks of code that contain "difficult" features and be unable to apply optimizations to these fragments, or to suitably restrict the language. We examine the choices made by the few available vectorizing and parallelizing C compilers and consider what the future may hold in the light of current research.

1. Introduction

The goal of automatic parallelization, whether for multiprocessors or for vector machines, is to take a program with serial semantics and have the compiler or preprocessor produce a parallel program.

The are many advantages to this. The programmer avoids having to deal with the difficulties of synchronization and other issues that arise in the writing and debugging of explicitly parallel programs (Grob and Lipkis [86]). There is a guarantee that the semantics of the program will be preserved and that the resulting program will be correct. The arguments that are used for stating the need for and advantages of optimizing compilers are applicable here as well. The compiler will be able to find parallelism that the programmer does not see. Programmers who set out to solve a problem shouldn't have to be experts in the kinds of data flow techniques used to find non-obvious potential parallelism, or in the transformations necessary to exploit parallelism. A parallelizing compiler will be able to work on library routines and other functions that were not written by the programmer. In a large software project with many programmers with different styles of programming and different levels of expertise the parallelizing or vectorizing compiler will supply a consistent level of parallelism and provide some guarantee of correctness. However, these arguments do not eliminate the usefulness or the desirability of explicit parallel programming. It is

often useful for a programmer to be able to manipulate the parallelism and control the asynchrony him or herself.

Because the potential benefits of automatic parallelization are considerable there is a great deal of interest in the topic. However, most work on the compilers that discover and exploit concurrency is being done on FORTRAN and not on C.

There are features of the C programming language that make it difficult to do the analysis necessary to optimize, much less exploit concurrency. These features are unconstrained pointers and other forms of aliases.

The first few attempts to write parallelizing or vectorizing compilers for C either restrained the language by limiting the use of pointers or didn't attempt to parallelize statements or blocks of code that contained pointers.

In this paper, we will attempt to explain what it is necessary to know about a program in order to parallelize it safely and why the C language poses problems. We will also give a few approaches to the problem that appear potentially promising. In order to present these issues clearly, we will first present an overview of the necessary background material.

2. Architectural Models

We are concerned with automatic exploitation of parallelism for three basic architectural models: vector processors, multiprocessors and very long instruction word (VLIW) machines. The techniques and constraints involved in finding parallelism for these architectures are similar although the architectures themselves are quite different.

2.1. Vector Processors

Conceptually, the idea behind a vector processor is quite simple. An operation is performed with two arrays as the operands. Instead of a loop iterating through all the elements of the arrays, all the elements of the arrays are processed in parallel. For many kinds of programs, which spend the bulk of their execution time on vector operations, this can be a significant speedup.

Architecturally, what happens in most modern vector processors is somewhat different. The elements of the arrays that are the operands of the vector operation are passed into a pipeline of processors. Each processor performs a part of the primitive operation on whatever data it is given. The data then moves on to the next processor which performs its part of the operation. At the same time the previous processors are performing on other data. It takes some amount of time for the first operands to move through the pipeline. There is a result on every tick thereafter.

2.2. Multiprocessors

The multiprocessors referred to in this paper are MIMD[1] machines. MIMD machines consist of many processors working together on a single job. Each processor may operate autonomously, not in lock step with the other processors. The processors may or may not share memory. They may communicate over a bus or a network

2.3. VLIW Machines

VLIW machines have a word size large enough to hold multiple machine instructions. Instructions loaded into the word at the same time are executed in parallel. The difficulty is in deciding which instructions may be executed in parallel and in restructuring the program to execute as many instructions as possible in parallel without changing the results of the program.

2.4. Hybrids

Although these are the three architectures that we are going to discuss, the categories are not absolute. There are vector machines that contain more than one set of vector processors and do vector processing in parallel and there are multiprocessors and VLIW machines that have vector units.

---

[1] In the taxonomy of Flynn [66], MIMD is the category of Multiple Instruction stream, Multiple Data stream computers (i.e. asynchronous multiprocessors); whereas in SIMD designs there is just a Single Instruction stream for the Multiple Data streams.

3. Dependences

Dependence analysis is important in many areas of optimization and critical in automatic parallelization, including deciding which statements may be loaded into a single VLIW instruction and executed together. A dependence exists between one statement and another or between a statement and itself when the same memory location or elements of the same array are accessed in those statements. This may enforce an ordering or serialization of the statements in which the dependence exists.

Not all dependences prohibit parallelization or optimization. In some cases, the code must be transformed and the dependence removed before any parallelization or optimization can occur. Sometimes all the dependent code must be moved or treated as a unit. In other cases, the dependence will preclude any parallelization or other potential alteration in the sequence of execution.

The dependences which occur between statements have been put into three classifications: flow dependence, anti-dependence, and output dependence (for more background on dependence see Wolfe[82] and Burke and Cytron[86]).

A flow dependence means data will "flow" forward from one statement to another. An example of a flow dependence is:

```
for (i=0; i<=100; i++) {
        x = w + z;
        a[i] = x;
          .
          .
          .
}
```

The second statement is dependent upon data from the first. The store must be completed before the load.

An anti-dependence is the reverse of a flow dependence. The load precedes the store. In order to get a correct value stored the order of the statements must be preserved.

```
for (i=0; i <= 100; i++) {
        a[i] = x + foo(g);
        x = w + z;
          .
          .
          .
}
```

There is no data dependence between the statements. But there is a storage dependence between them. The fact that load of *x* must complete before the store into *x* could be a problem if this loop were run in parallel or if these instructions were executed together.

Recent research suggests a solution to the problem of anti-dependences or storage related dependences called *variable renaming* (Cytron and Ferrante[87]). The use and reuse of variables is sometimes a matter of happenstance or storage conservation. Once it is determined that there is no computational dependence between two statements, only a storage dependence, then the dependence can be eliminated of by renaming the variables. Of course, all future references to the variable must use the last variable name. So the transformed code would look like this:

```
for (i=0; i<= 100; i++) {
        a[i] = x1;
        x2 = w + z;
```

```
        .
        .
        .
}
```

An output dependence is caused by several stores to the same location. Again, in order to get the correct result and not alter the observed behavior of the program the stores must be completed in the order that they are written.

```
for (i=0; i<= 100; i++) {
        x = a[i];
         .
         .
        x = w + z;
         .
         .
}
```

As with anti-dependences there is no data dependence here only a storage dependence. This too can be resolved with variable renaming. When removing output dependences the last assignment to a variable must have the original variable name or all later occurrences must be renamed. This is so that if the variable is printed immediately after the loop the observed behavior will be the same.

Flow dependences are more of a problem. The semantics of the program may require that data be communicated from one statement to a subsequent statement. Sometimes forward substitution can be used to remove the dependence while retaining the flow of data.

```
for (i = 0; i <= 100; i++) {
        a = x + 5;          (1)
        b = a - 10;         (2)
        c = a + b;    (3)
        x = c + y;    (4)
}
```

There are flow dependences between (1) and (2) because the value of *a* is carried forward, likewise between (2) and (3) with respect to both *a* and *b* and between (3) and (4) with respect to *c*.

After forward substitution this looks like this.

```
for ( i = 0; i <= 100; i++) {
        a = x + 5;
        b = x - 5;
        c = 2x;
        d = 2x + y;
}
```

4. Optimization

Optimization is both the precursor of and the necessary first step in automatic parallelization. Automatic exploitation of concurrency can be thought of as a special kind of optimization. In both cases the goal is to make the code faster. The analysis and code transformations needed to do optimization are a subset of those needed to discover opportunities to create parallelism.

In some sense, a compiler will generate code for the program that has been written while an optimizer should generate code for the program that would have been written if a very good programmer had a very good knowledge of the architecture. Of course this is a vast over-simplification. Optimizing compilers work on the principle that programmers write without any real cognizance of the architecture that their programs are running on. The optimizer takes the code that has been written to solve some problem and restructures it in order to make it run faster. A great deal of this restructuring is architecture independent. The restructuring that is not machine independent includes things such as register allocation and substitution of a faster instruction sequence for a slower one. In this section we will discuss some of the machine independent transformations.

One of the problems of optimization is deciding how much the code can be changed while still being semantically the same program. It is clear that the transformations done to optimize code change the state of the program in some sense. What defines a program? Certainly the output should be the same as before the optimization. Also, the optimization should not cause an exception to be raised in the optimized program that would not have been raised in the original program. But what about statement execution? If a statement isn't executed the same number of times in the optimized program as it is in the original program, is it still the same program? How difficult is it to determine that changing a variable name or deleting or moving a statement will not change the result the of the computation?

The idea behind many optimizations is that some statements do not have to be executed in the order that they are written. The ordering is often random or at least imposed only by the necessity of putting them in some some linear sequence. The relative order of every statement is rarely necessitated by the problem.

A similar line of reasoning pertains to variable names. Many times a single variable is used and reused for computationally unrelated purposes. It is only being used for storage. If this is determined to be the case then later instances of that variable can be renamed or moved.

Two of the most common optimizations are *constant propagation* (or *constant folding*) and *common subexpression elimination*. These techniques are first applied to small sections of code and then to whole procedures and functions.

4.1. Dataflow Analysis

Data flow analysis is done by summerizing information about a program in equations and then using those equations in the analysis. The flow of data is followed; information such as which variables are *live* at any point and which definitions of variables reach any given point is kept track of. There are equations that are recomputed along all the possible control pathes in a program. In some cases, the information is propagated forward along the program control graph. In other cases, the information is propagated backwards from the program exit to the beginning. An example of information that is propagated backwards is *live variables*. A variable is live if there is a later use of its value. An example of information propagated forward is *reaching definitions*. Reaching definitions are the set of possible definitions that reach a variable at a given point in the program. The problem of dataflow analysis becomes complicated when it is necessary to do it on a program with many modules. It is necessary to know which variables are changed or defined in a module.

4.2. Levels of Optimization

Local optimizations are optimizations applied at statement or flbasic block level. A basic block is a sequence of consecutive statements that has only one entrance and one exit. Global optimization is actually a misnomer. It is not global optimization that makes use of the overall program call graph, that is interprocedural optimization. Global optimization is applied at the procedure and function level. In order to show the kinds of problems and the types of solutions involved, we will give 2 examples of local optimizations.

4.2.1. Constant Propagation

Constant propagation is an important problem in optimization. The purpose is to find values in the program that are constant and propagate them through the code. When expressions are found whose operands are constant they can be evaluated at compile time and then these results propagated forward through the program as well. This last is sometimes also called constant folding.

4.2.1.1. Motivation

Constant propagation and folding can lead to faster execution because some of the evaluation will be done by the compiler. In some cases, this can be critical in deciding if code can be vectorized or run in parallel. Also, constant propagation can be an aid in register allocation. If there are several variables that evaluate to the same value they can be assigned the same register. If an expression in a conditional branch turns out to be constant then the branch not taken can be eliminated as *dead code*.

4.2.1.2. How many constants can they find?

All the work that we discuss below takes a conservative approach to finding constants. These methods may not find all constants but everything that is found will be constant over all possible executions of the program. The early work on constant propagation (Kildall [73]) found all *simple constants*. A simple constant is a value that is never changed along its path through the program. Furthermore, no assumptions are made about which direction a path will take. This means that if there is a branch that leads to a change in the value of a constant, even if that branch may never be taken, this will *kill* the constant.

More recent work (Wegman and Zadeck [85]) has made it possible to relax these constraints. It is possible to decide that a term is constant even though it appears as if its value may change as the result of an assignment under a conditional branch. It is also possible to propagate a value forward even if the variable has different constant values at different points. The branches that are never taken are ignored and parts of the program that are never executed are *dead code eliminated*. All the constants that can be found by evaluating all the conditionals that have all constant operands are found.

4.2.2. Code Motion

The goal of code motion is to make the code faster by moving code from places where it is executed many times to places where it is executed once. The most obvious place to apply this is to a loop. The more deeply nested a statement is, the more times it may execute.

Safety, a recurring issue in code optimizations, is especially an issue in code motion. Safety refers to the fact that the observed behavior of the program must not be altered. This includes avoiding any exceptions that would not be raised in the original program.

4.2.2.1. Safety

Safety really deals with deciding if moving a piece of code could affect the observed behavior of the code on some possible execution of the program. If the statement is a loop-invariant why should moving it out of the loop change the observed behavior of the program? The statement might be only conditionally executed. So moving it out of the loop and out from under its conditional guard may cause it to be executed when it wouldn't be executed if it remained in the loop. There might be a dependence of some kind between the statement and another statement, where only one of the statements is an invariant.

4.2.2.1.1. Dependence and Code Motion

Dependence has been discussed in a previous section. The relationship between dependence and code motion is slightly different with each type of dependence. Looking at the examples of dependences in the previous section:

In the case of the flow dependence: the first statement can be moved out of the loop saving the recomputation of the expression.

In the case of the anti-dependence: variable renaming removes the dependence. The invariant statement can now be moved out of the loop without modifying the program result.

The output dependence can be handled in exactly the same way as the anti-dependence. After the variables are renamed, the dependence disappears.

4.2.2.2. Strict versus Non-strict Code Motion

Code motion is strict if the affected statement is executed no more frequently after it is moved then before (Cytron, Lowery and Zadeck[86]).

But what if a statement is moved out of a while loop? Then the statement is moved from a place where it may never be executed to a place where it is necessarily executed once. This may violate the rule that the observed behavior of the program should not be altered.

If the surrounding control structure is loop-invariant, then the solution is to copy the control structure as well. This will insure that the statement will never get executed unless it would have been executed in the original loop.

```
while ( n + m + p > y ) {
        x = f;
        m++;
        if (n > z) {
                a[n] = z;
                y = p;
                 .
                 .
                n++;
        }
        .
        .
}
```

when transformed and after code motion becomes

```
c1 = n + m + p > y;
c2 = n > z;
if (c1) {
         x = f;
          .
       .
        if (c2) {
                y = p;
                 .
                 .
        }
}
while (n + m + p > y) {
        m++
        if (n > z) {
                a[n] = z;
                .
                .
                .
                n++;
```

}

Thus the observed behavior of the program is preserved. Nothing is executed more often then it would be in the original program. The necessary condition for strict code motion is that the test guarding the code must be movable or the code cannot be moved.

Non-strict code motion will move a statement from a place where it may never be executed to a place where it will be executed at least once in the hope that the expected behavior of the program will cause this to be a profitable move. Both kinds of code motion must be equally conservative in the sense of preserving the expected behavior of the program. In fact, all optimization algorithms must err on the side of missing optimizations rather than altering the behavior of a program.

The non-strict procedure looks for code or expressions that are common along all paths from a branch. If it finds such expressions, then they can be moved above the branch.

4.3. Interprocedural Optimizations

The same optimizations and transformations that are applied locally may be applied interprocedurally, subject to certain conditions. If the program consists of one single source file, then it is easy to apply the techniques above and others like them. However, if the program consists of several modules then there are various options. One option is to perform only local optimizations and not try to do any interprocedural optimization. This is a popular alternative because local optimization is easy and profitable and interprocedural analysis is difficult. In order to do interprocedural optimization it is necessary to take into account aliasing. Also, if any module is changed the entire program must be reoptimized. The kinds of optimizations that should be performed globally are the same ones that are performed locally; such as global constant propagation, dead code elimination, constant subexpression elimination and code motion. In order to do that it is necessary to perform interprocedural data flow analysis.

4.3.1. Interprocedural Dataflow Analysis

Aliases pose a problem in interprocedural analysis. They arise, even in languages without pointers, from parameters passed by reference. Aliases also arise from several structures mapped onto the same area of memory, such as unions in C and equivalences in FORTRAN. The most conservative forms of interprocedural analysis deal with the presense of aliases by making worst case assumptions. The worst case assumption is that any function call may modify any variable. So after the function call it is assumed that all the variables that are reachable by that function have been modified. This will affect all optimizations as well as any possibility of vectorization or parallelization. But, is it necessary to be that conservative?

There are groups that have applied modern dataflow techniques to this problem and been able to handle the problems of aliasing (Burke [84]). The information about a module is summerized and propagated back to its call point where it is matched with the parameters to the module. This information is then used in the parent module and propagated back to its call point. Thus, it is possible to to determine the actual aliasing.

4.3.1.1. Separate Compilation

In order to do good interprocedural analysis all code for all the modules must be present. In the presense of separate compilation this is not always feasible. Certainly it would be better not to have to reoptimize the entire program every time one module is changed. If the program calls library functions or links in some module where it does not have access to the code then it cannot do the reoptimization. For the most part, it is a problem that remains unsolved today. There are several partial solutions and attempts at solutions.

The MIPS solution is to link the intermediate code (Himmelstein, Chow and Enderby [87]). When a module is altered the necessary parts of the code are all there to reoptimize. So this lets you reoptimize, after changing a module, but forces you to look at the entire program with every change.

There is research being done that work says that when a single module of program is changed it isn't necessary to recompile the entire thing (Cooper, Kennedy, and Torczon [86]). Given a database of information containing compilation dependences, this information is compared with changes in the programs interprocedural information and a list of procedures requiring recompilation is produced. A change in one module might not have to necessitate the reoptimization of the entire program. This can be tremendously complicated, of course, by the presense of global variables and pointers.

5. Automatic Parallelization

5.1. Where does the parallelism come from?

Parallelism can be found at many different levels in a program. Expression level parallelism refers to the evaluation of several different parts of the expression at the same time. For instance, in an expression containing addition operations and multiplication operations the various operands could be evaluated at the same time, subject to the rules of precedence. This is the type of parallelism found in VLIW machines. Further, in VLIW machines, there is expression level parallelism for many expressions happening at the same time. There are also statements that can be run in parallel. There is loop level parallelism, where the iterates are run in parallel. Ideally, it is desirable to find all the parallelism in a program, no matter at what level it exists. It may be questioned whether it is worth executing something in parallel if it is only a single expression. The answer to this is based on the architecture and the implementation of those features that create and control parallelism.

Numerical programs and other scientific applications are often very regularly structured. The computation is often done in a series of loops. The data structures are arrays or matrices This type of program lends itself very well to automatic parallelization and vectorization as generally practiced.

Non-numeric applications may have a less regular structure. But if a program is written with the idea that it is going to be automatically parallelized, that is, bearing in mind the types of transformations done and the constructs that can be parallelized, these applications can get significant speedup (Lee, Kruskal and Kuck [85]). When non-numeric applications with irregular *shapes* are parallelized the speedup obtained is likely to be less significant. Because VLIW machines exploit fine grain or expression level parallelism they are less susceptible to this problem.

Most vectorizing compilers and parallelizing compilers for multiprocessing look for parallelism at the loop level. The vectorizing compilers attempt to parallelize the innermost loop in a nested construct. Each statement in the innermost loop will be vectorized separately. The compilers for multiprocessing attempt to parallelize the outermost loop in order to minimize synchronization and to achieve optimal parallelism. The goal of here is to execute loops in parallel with all the iterates being done at once. It is loop level parallelism that we are going to discuss.

5.2. When Can A Loop Be Executed In Parallel?

In order to understand the complexities of parallelism of this sort, one must picture all the iterations of a loop executing in parallel and at different speeds. The danger is that there may be a dependence existing between several statements in the same or different iterations. The existence of a dependence may make it necessary to have stores and fetches of various elements completed in a prescribed order. Vectorizing or multiprocessing may allow that ordering to be violated, causing the program result to be incorrect. So it is necessary to discover the dependences that exist and decide if they are of a type that will prevent the vectorization or parallelization of the loop. The data flow analysis that is necessary to find dependences for optimization is a subset of the analysis that is needed to detect parallelism.

5.2.1. Analysis of Array Subscripts

In a previous section dependences involving scalars were discussed. But in many programs the majority of the computing is done by manipulating arrays and elements of arrays. When there are loops, possibly nested, and arrays with one or more subscript variables, it is not always obvious when two or more statements will be accessing the same locations on the same or different iterations. The most conservative approaches for discovering dependences do not attempt to analyze the subscript of the arrays to see if a dependence exists. Instead, a store to any element of an array is considered to have changed the entire array. This may unnecessarily prevent parallelization or

vectorization since it will appear that there is a dependence where none exists. This is erring on the side of conservatism. It is much safer to see dependences where none actually exist and therefore not to parallelize a section of code, then to execute it concurrently and violate a dependence.

More sophisticated dependence analysis is based upon forming dependence equations based upon linear subscript expressions (Wolfe [82]). The corresponding expressions are set equal to each other and if there is a solution to the equation then there may be a dependence.

5.2.1.1. Context

It is possible to get a more precise answer to the question of the existence of a dependence if the *context* of the loop is considered. Context means that only integer solutions within the bounds of the loop are considered as solutions to the dependence equations. So if it can be proved that a dependence exists, but not within the bounds of the loop, then the loop can be parallelized.

```
for (i = 0; i <= 49; i++) {
        a[i] = b[i];
        d[i] = a[100 - i];
}
```

The stores into *a* in this loop are into elements 0,1,2,...,49 and the fetches from *a* are from elements 100,99,98,...,51. When i equals 50 there is a dependence between these two statements but since the bounds of the loop go from 0 to 49 no dependence exists within the context of this loop. Even if the loop bounds are symbolic they may well be available by compile time. Clever analysis and constant propagation may maximize the chances of parallelizing such a loop.

5.2.1.2. Plausiblity

There might be many solutions to the dependence equations. But only some of them will fall within the context of the dependences. The context is also a function of the direction that the dependence runs in. If there is a dependence that exists only if one loop is running forward while the other loop runs backward and both are actually running in the normal fashion from one to n then that dependence is *implausible* and doesn't have to be considered. Information about the chronological relationship between the subscripts of the two elements that are being tested is summerized in a *direction vector* (Wolfe [82]), (Burke and Cytron [86]). There is one entry in the direction vector for each subscript of the array. This vector can be interpreted according to the direction that the loop is running. This can help decide if a dependence that exists is plausible.

5.2.1.3. A Hierarchy of Tests

There are several different tests for independence. They can be thought of as forming a hierarchy that ranges from the most general that looks for any integer solutions to the dependence equations, to tests that look for only integer solutions within the loop bounds, to tests that look for integer solutions within loop bounds and consider the direction vectors. In order to minimize the cost of the analysis the cheapest test is done first. If this test proves independence it is not necessary to do any further analysis. If this test fails to prove independence then a more expensive test is applied. If there is still a dependence then finally the most exact test is applied. By using the tests in a hierarchy there is a possibility of holding down the amount of analysis necessary. This is because all the tests are conservative and will err on the side of not proving independence. Once independence is proved no further testing is necessary.

5.2.1.4. Flow Dependences

The dependences that prevent parallelization or vectorization are flow dependences. A flow dependence is one that occurs when a store of a variable precedes a fetch. It can also exist between iterations as when there is is a store

into an element in one iteration that is fetched in the next or in any succeeding iteration. This can be in a single statement or between two statements and can be thought of as a recurrence.

5.2.1.5. Vectorization

Vectorization works by running one statement at a time in parallel over all its iterates. It cannot take place if a cycle of dependences exists in the statement or statements being vectorized.

If there exists a flow dependence between two statements in a loop, and the statements are to be vectorized then one statement must be completely executed before the other is begun or the answer will be incorrect. Given the following code fragment:

```
for (i = 0; i <= 100; i++) {
        a[i] = b[i] + c[i];
        d[i] = a[i] + e[i];
}
```

If this was executed serially then in every iterate the store of *a[i]* would be completed before the access of *a[i]*. But if both of these statements were vectorized, then there might be at least one iterate where the load of a[i] in the second statement would be executed before the store of the same a[i], in the first statement. The solution to this problem is to vectorize the first statement and when it completes to do the second statement also in parallel.

Early vectorizers would parallelize all or none of a loop. More modern compilers try to do as much as they can.

```
for (i=0; i <= 100; i++)
        a[i] = a[i-1] + c[i] + d[i];
```

A clever vectorizer would separate the last piece of the statment and vectorize the addition of *c* to *d*.

5.2.1.5.1. Conditionals

Conditionals that limit which elements of an array are used as operands can be dealt with by taking the test and making a conditional array out of it.

```
for (i=0; i <= 100; i++)
        if (a[i] < n )
                d[i] = a[i] + b[i];
```

Can be transformed as follows:

```
Vectorize: c[i] = a[i] < n;
Vectorize: d[i] = a[i] + b[i] where c[i];
```

Then the vector operation is performed on all the elements where the conditional array contains a one. If the conditional array is very sparse it may not be worth it to do the vectorization. This can be applied to nested conditionals as well. The more deeply nested the conditionals are, the more important it is to estimate the sparsity of the

conditional array. Since each layer of nesting probably eliminates some number of iterates that can be vectorized, it may be that it can become unprofitable to perform the vectorization.

Conditionals that cause an exit from the original loop cannot be dealt with this way.

5.2.1.6. Multiprocessing

Multiprocessing of a loop cannot take place if there is a flow dependence between iterations. If there is a store in one iteration that is used in a subsequent iteration the two iterations cannot be run in parallel. There is every possibility that they will complete out of order and the result will be incorrect. If there is a flow dependence but it is only in the same iteration then that loop can be multiprocessed. Sometimes the dependences can be worked around by synchronizing the references. This provides an added cost and removes some of the benefit, sometimes all of the benefit of parallelization. If the dependence is across iterations of the outer loop or if there are many synchronization points in the outer loop then it may be most profitable to parallelize an inner loop. The desire to obtain the most parallelism by parallelizing the outermost loop must be balanced against the cost of synchronizations in that loop.

5.2.1.6.1. Do Across

If the loops have a flow dependence across iterations that makes it impossible to run them completely in parallel it may still be possible to get some of the benefit of parallelism. There has been a construct suggested[2] called the *do across* construct (Cytron [86]). It is a parallel loop with a delay. This allows a dependence in the form of a reference to a previous iterate be satisfied.

When the first iterate is executed, the second is not started until the store in the first iterate completes. This is repeated for all iterates. Thus if the store is is in the beginning of the loop, it is conceivable that all of the loops will be running at the same time in this overlapping fashion.

The profitability of this is directly linked to the length of the delay. A delay of zero is equivalent to a fully parallel loop.

5.2.1.7. Transformations for Concurrency

Code can be transformed in order to make it vectorizable or parallelizable or in order to increase the benefit of concurrency. Like the optimizations discussed in a previous section, these transformations must not cause the program to give an incorrect answer or raise an exception where none was raised before the transformation.

5.2.1.7.1. Getting Rid of Dependences

The first group of transformations are architecture independent and used to put code into a form which can be either parallelized or vectorized. One goal is to get rid of all output and anti-dependences and as many flow dependences as possible.

The first transformation is scalar renaming which removes anti and output dependences by using more storage. This was discussed in the previous section on dependences. Scalar expansion is a similar transformation. Here a scalar that is used in a loop is *promoted* to an array. This can eliminate a noncomputational dependence that could prohibit multiprocessing or vectorization. The price of this expansion is a large amount of storage which must be reclaimed at the earliest possible time.

5.2.1.7.2. Other Transformations

The number of implied and explicit gotos can be reduced by reproducing the program from the control flow graph. This has the effect of straightening the code, thus exposing more of it for possible parallelization. Also, loops can be normalized by making their lower and upper bounds zero or one through n and adjusting the array subscripts.

Since vectorizing code means vectorizing inner loops, if there is a dependence in the inner loop but not the outer loop it would be desirable to be able to exchange the inner and outer loops. The preconditions for loop exchanging are that the exchange must not turn an anti-dependence into a flow dependence or an flow dependence

into a anti-dependence.

Another transformation used for vectorization is to convert if statements to loops, where possible, and then vectorize them.

There are many other code transformations used to increase the amount of parallelism possible. These are some of the most important ones.

6. The C Programming Language

6.1. Difficult Language Issues

In order to be execute an instruction in parallel with another instruction, it is necessary to prove that no dependence exists between the them. This is true whether the instructions are going to be run on a vector processor, a multiprocessor or a VLIW machine. The reasons that make it necessary to prove independence and some of the techniques used to do so have been discussed in previous sections. There are features in the C language which make it difficult to prove independence. These are recursion, unions, parameters, global variables and unconstrained pointers. Of these features, only unconstrained pointers are unique to C.

In particular, unions, parameter passing by reference, global variables and recursion all occur in FORTRAN. The problems posed by these language features have been dealt with by various automatically parallelizing or vectorizing FORTRAN compilers in different ways. Since the purpose of this paper is to discuss the unique difficulties of automatic parallelization in C, we will not elaborate on the different solutions that have been applied to these problems.

6.1.1. Pointers

How does the presense of pointers make it more difficult to discover areas of potential concurrency? Pointers make it possible to refer to the same memory location through many different paths. This makes it very difficult to detect dependences.

When the use of pointers is constrained by the language, it is easier to devise some scheme for working around them. In C, where the use of pointers is essentially unconstrained, where the address of any variable can be taken, where a pointer to a type can be made to point to any other type just by casting, the field of possible aliases is enormous. The analysis becomes very difficult, If not impossible to do effectively.

6.2. Is There an Overall Solution?

The problem of dependence analysis in the presence of pointers is so hard that in the general case there is no solution. It is not possible to know to what location a pointer is pointing. Thus it is impossible to know, at compile time, when a variable is being fetched or stored and when it is not. Therefore, it is not be possible to compute the necessary dependence information.

This doesn't mean that nothing can be done to parallelize code that contains pointers. It is not necessary to be able to solve the general case. There are solutions for specific cases that can provide enough parallelism to be satisfactory, if not optimal. To put it another way, if it possible to run a loop concurrently because it has been possible to do the dependence analysis for that specific case, then the parallelism is still worthwhile even if it is impossible to solve general problem of dependence analysis in the presense of pointers.

6.3. Proposed Solutions

One approach that is safe and conservative is to treat all pointer variables and variables that have their address taken as if they are aliased together. Then whatever dependence analysis is done takes that alias into consideration. This is very safe and may yield some parallelism even in the presense of pointers.

---

[2]Now in use by Alliant.

This can be improved by allowing the programmer to pass information to the compiler by means of directives. The compiler could be told to vectorize or parallelize some code even if it looked like a dependence existed. This would cause a wrong answer if a dependence really did exist. It also relies on the programmer to understand enough about the process of automatic parallelization to know if a dependence exists.

We believe that it may be possible to do better. Several general approaches appear promising. We are not going to spell out algorithms; instead we are going to suggest some ways of thinking about the problem and some techniques that might lead to more information.

While it may not be possible to know in advance where a pointer is pointing; it may be possible, in a few relatively static cases, to know where it is not pointing. It is not necessary to know at what location a pointer variable is pointing if it can be proved that it is not aliased to anything that would prohibit the parallelization or vectorization of the loop. We would like to be able to partition the set of possible aliases.

First consider global (meaning one function at a time) analysis. Starting at the definition of a pointer variable and going forward along the control flow graph, every time a pointer variable is set to point to any specific named variable the aliased pointer variables could be put into an equivalence class. A member is removed from the equivalence class if it is set to point to an address that is specifically in another equivalence class. If a member of an equivalence class is found to be an alias for a member of another equivalence class then the two classes are unioned together. All pointer variables and variables that have had their address taken and that are not found to belong in a specific equivalence class are put together in an equivalence class of their own. Since it cannot be determined what the set of possible aliases for these variables are, it must be assumed that they can point anywhere. The analysis must be done assuming that these variables are included as potential aliases in every equivalence class. If the pointer is a parameter to a function the alias information is carried forward interprocedurally in the same way that aliased sets of reference parameters are handled in FORTRAN (M. Burke [84]). In this manner, while it may not be possible to know where something is pointing, it may be possible to know that two pointers cannot be aliased together. If the pointers are in two disjoint classes, they cannot be aliases for each other. This should increase somewhat the available parallelism.

In the case of a function that has only local pointers this should provide enough information to do the dependence analysis. In the presense of dynamic allocation and pointers returned by user functions, this is certainly not sufficient.

Pointers used to index into arrays are another area where we believe it may be possible to obtain sufficient information to perform the dependence analysis and get some parallelism.

The pointer and its increment can be thought of as an operation on an induction variable. An induction variable is a variable that is incremented or decremented a constant amount, usually on each iteration of a loop. The address arithmetic done to increment a pointer is often of this nature.

In this case, where a pointer is being used to iterate through an array, then by treating the pointer as an induction variable is possible to convert the pointer reference into an array subscript operation and perform dependence analysis as mentioned previously. If there is more then one pointer addressing s single array this is equivalent to having more than one subscript.

The pointer must be restricted from going past the end of the array.

6.4. Conclusion

There are real difficulties involved in writing optimizing, parallelizing or vectorizing C compilers. It appears to be possible to to work around these difficulties without restricting the language or totally giving up the advantages of parallelism. The cost of this is a great deal of analysis. It remains to be seen whether the benefits from the added parallelism will outweigh the costs of this analysis.

As dataflow techniques improve, it will become possible to make less pessimistic assumptions about data relationships without sacrificing program correctness. These improved techniques will be applied first to FORTRAN, which offers a much more static environment for experimentation and which has a much higher demand for parallelism. Inevitably these improved techniques will be applied to C and an amount of parallelism and optimization which was deemed miraculous a short time ago will appear to be unacceptable in the light of what can be demanded.

We do not believe that a general solution to the problems of pointers will be found. But we believe that it is possible to obtain a useful amount of parallelism from C even under present conditions. We also believe that the approaches that we have discussed here and other approaches that will arise from later dataflow work and be applied to this problem promise a greater amount of obtainable concurrency without restricting the language.

7. Acknowledgements

This work was inspired by and stemmed from conversations with Ron Cytron of IBM Research, who encouraged me to pursue this and whose own work contributed to the foundation upon which this rests. Stefan Freudenberger of Multiflow Computer, Inc., participated in later discussions. Jim Lipkis of the NYU Ultracomputer Project contributed to all these discussions and provided encouragement and insight. My heartfelt thanks to these three people.

References

F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing", *Proc. International Conference on Supercomputing*, 1987.

M. Burke, "An Interval Analysis Approach Toward Interprocedural Dataflow", *IBM Research Report RC11794*, 1984.

M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization", *Proc. the SIGPLAN '86 Symposium on Compiler Construction*, 21(7):162-175, July 1986.

K. Cooper, K. Kennedy, and Linda Torczon, "Interprocedural Optimization: Eliminating Unnecessary Recompilation", *Proc. the SIGPLAN '86 Symposium on Compiler Construction*, 21(7):58-67, July 1986.

R. Cytron "Doacross: Beyond Vectorization for Multiprocessors", *Proc. International Conference on Parallel Processing*, August 1986.

R. Cytron and J. Ferrante, "What's in a Name?: or The Value of Renaming for Parallelism Detection and Storage Allocation", *Proc.*International August 1987.

R. Cytron, A. Lowery, and K. Zadeck, "Code Motion of Control Structures in High-Level Languages", *Conf Rec. ACM Symposium on the Principles of Compiler Construction*, 1986.

M. J. Flynn, "Very High Speed Computing Systems", *IEEE Trans.*, **C-54**, pp. 1901-1909, 1966.

L. Grob and J. Lipkis, "Approaches to Parallel programming on Multiprocessors", *Proc. of the EUUG Autumn 1986*, September 1986.

M. Himmelstein, F. Chow, and K. Enderby, "Cross-Module Optimizations: Its Implementation and Benefits", *Conf. Proc. Summer 1987 Usenix*, June 1987.

G. Kildall, "A Unified Approach to Global Program Optimization", *Conf. Rec. First ACM Symposium on Principles of Programming Languages*, pp. 194-206, October 1973.

G. Lee, C. Kruskal, and D. Kuck, "An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors", *IEEE Trans. Comp.*, October 1985.

M. Wegman and K. Zadeck, "Constant Propagation with Conditional Branches", *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages*, pp. 291-299, January 1985.

M. Wolfe, "Optimizing Supercompilers for Supercomputers", *PhD thesis, University of Illinois at Urbana-Champaign*, 1982.

An Intelligent, Window Based Interface to UNIX

Stuart Borthwick, John R. Nicol, Gordon S. Blair

Department of Computing
University of Lancaster
Bailrigg
Lancaster, LA1 4YR
U.K.

email: mcvax!dcl-cs!stuart

ABSTRACT

Many operating systems have been developed with little thought given to the user interface. Most interfaces tend to be awkward to use and rely on simple command line interpreters. Although many experienced users have little trouble with the UNIX† interface, the popularity of the system demands that a more user oriented interface will be required in the future. One aim of this interface should be to alleviate the need for naive users to learn and remember a large number of command names and cryptic command lines.

The UNIX interface can be improved by employing the facilities offered by bit-mapped graphical workstations (menus, windows, icons, etc.). To use these facilities most effectively, the interface requires intelligence. This can be provided by associating semantic information with UNIX objects. This paper describes a UNIX interface which uses a mode of interaction known as *direct manipulation.* Direct manipulation assumes an environment of objects and associated semantic information (i.e. attributes). It encompasses the ability to select objects with a pointing device, display and issue commands associated with the objects and change attributes of the objects.

This kind of interface helps a user in several ways. Displaying UNIX objects (files, directories, etc.) as icons provides a more natural interface. In addition, the association of semantic information with objects makes it easier to provide a more intelligent user interface. For example, with typing information the system can deduce exactly which commands are permissible in a given context and does not rely on commands themselves to detect errors. Consistent and helpful error messages can also be generated. Finally, the use of direct manipulation, combined with semantic information, permits a much reduced command set to be displayed in a given user context. The authors see this as an effective way of assisting the user.

† UNIX is a trademark of Bell Laboratories.

1. Introduction

UNIX is unarguably one of the most successful operating systems in the history of Computer Science. The system has been adopted universally by both industry and academia. Indeed, the ever increasing acceptance of UNIX has resulted in the system being regarded by many as a de facto standard. However, the rapid growth in UNIX popularity has led to some serious repercussions. In particular, a number of important new requirements has created a pressing need for several aspects of the system to evolve further.

This paper focuses on one such aspect of the system's evolution path, its *user interface:* the growing number of non-expert UNIX users and the need to exploit the sophisticated graphics facilities typical of modern workstations have highlighted the inadequacy of the traditional UNIX user interface, *the shell.* The command line interface presented by the shell is difficult for naive users to learn, is inconsistent across products and is a burden to remember [12]. The shell cannot make use of the powerful facilities provided by workstations and its interface is in stark contrast to that presented by a window manager or a tool designed to run in a window system.

More recently, systems have been developed which use a mode of interaction called *direct manipulation* [5]. This is described as the interaction of a user with an information environment consisting of dynamically changing windows. The user has a simple pointing capability through which he/she can select objects in the environment, issue applicable commands and, possibly change attributes of the objects. The graphics display not only provides a means of presenting sophisticated forms of output, but can be used as a mechanism to reduce the complexity of the system itself.

The display can be greatly improved through the use of an iconic interface [9]. By presenting user commands in the form of icons, it is possible to exploit further the capabilities of the graphics display. The user selects combinations of objects and facilities as a substitute for commands and parameters. Some attempts have been made to provide this type of interface to UNIX (e.g. UNICON [8]), but these have been restricted by lack of information pertaining to objects and the relationships between objects in the underlying UNIX system.

This paper presents a user interface to the UNIX system which has been developed on a Sun workstation. The system uses the direct manipulation type of interaction introduced above. To overcome some of the restrictions imposed by UNIX, the interface has been provided with a degree of intelligence (the primary source of which is through the association of semantic information with UNIX objects). The use of direct manipulation combined with semantic information permits a much reduced command set to be displayed in a given user context and allows, for example, consistent and helpful error messages to be generated. The system can deduce exactly which commands are permissible in a given context and does not rely on commands themselves to detect errors. The result is a much more user oriented system which makes effective use of existing UNIX tools as well as workstation facilities.

The paper is organised as follows. Section 2 discusses the underlying mechanisms used to provide the interface with intelligence. Section 3 presents an overview of the window system and style of interface. The support of version and configuration control is described in section 4. Section 5 then gives a more detailed description of the

window interface, and a brief outline of the current implementation status is given in section 6. Section 7 summarises the advantages of the interface and, finally, section 8 discusses the implications of the work in the wider context of operating system design.

2. Providing UNIX with Intelligence

A central feature of the UNIX operating system is its hierarchical filestore. Two primitive object types are defined in the filestore: *files* and *directories*. Files can be further classified as being either *regular* or *special*† files. Regular files act as containers for anything from source and object code to data and text.

In our opinion, to improve the user interface, the traditional view of the UNIX filestore must be generalised. For example, it should be possible to define new object types (and the operations applicable to them), relationships between objects, and to associate additional semantic information with objects.

The authors have generalised the UNIX filestore by further classifying regular files into a new set of system *object types*. In our experimental system, the set of object types is based on the set of types recognised by the UNIX *file* command (although this set can be augmented further by arbitrary user-defined types). Once the type of each object in the UNIX filestore has been established, it is associated with the object as one of its attributes. Additional attributes can be defined in a data structure called *a frame,* with a different frame existing for each object type.

The following sub-sections discuss the various ways in which object types and semantic information have been exploited in the pursuit of intelligence in the UNIX interface.

2.1. Semantic Checking

Forms of semantic checking such as type checking are commonly found in programming languages; however, they have received relatively little attention in the context of operating systems. By using the semantic information stored in object frames in conjunction with a simple *rule base* (described in the following sub-section), it is possible to implement type checking in UNIX. The key to a solution is to store type mapping information pertaining to UNIX tools in a rule base. In addition, the frame of a UNIX tool (or *command frame,* see section 2.3) is required to store a specification of the types of its possible input and output objects. The additional type information stored by the system can be used to check the validity of the given input objects in a tool invocation.

One major problem exists with type checking in operating systems. If the level of type checking is too strong, it becomes difficult to incorporate essentially generic tools such as editors (thus implying that specialised tools would have to be provided for each object type). If, however, the level of type checking is too weak, unreasonable operations will be permitted, e.g. displaying a binary object. We propose a flexible strategy to overcome this problem, based on a hierarchy of types. A simple hierarchy

† Special files are associated with physical devices. They are provided so that UNIX devices can be written and read in the same way as regular files. However, these are mentioned here only for completeness.

is shown in diagram 1.

Generic tools can now be incorporated by selecting a type at a high level in the hierarchy (e.g. an editor could take any input object with the attribute *text*). Other tools can be more specific, e.g. a C compiler should only be allowed to operate on an object with attributes *text:source:C*.

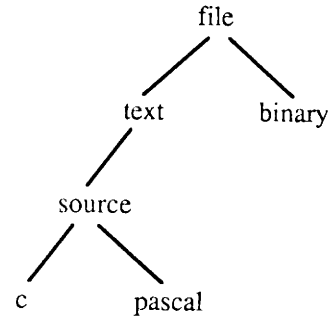


Diagram 1: A Hierarchy of Types

Thus, type checking can be used to provide more intelligence in the user interface. For example, it would be relatively straightforward to generate error messages of the form:-

Editing a non-textual file is not permitted.

2.2. The Rule Base

It is possible to implement type checking and type hierarchies (as described in section 2.1), through the additional use of a *rule base*. An example of a rule base is shown in diagram 2. In the diagram, three types of commands are introduced†:

* the TYPE command
  -- this allows the user to create new object types,
* the member of set (MS) command
  -- this allows the user to set up an arbitrary type hierarchy (the TYPE and MS commands in diagram 2 illustrate how the type hierarchy of diagram 1 would be built), and
* the RULE command
  -- this allows the user to add new rules to the rule base. A RULE command is expressed as a triple of the form <function name> <type mapping> <tool>, where *<tool>* specifies the UNIX tool to be invoked to produce the *<type mapping>* designated by *<function name>*.

† Other commands exist for removing types, member of set information and rules respectively.

```
TYPE file, binary, text
TYPE c, pascal, source,
TYPE unformated, formated
text, binary MS file
source MS text
c, pascal MS source
RULE 'format': unformated -> formated: nroff
RULE 'compile': pascal -> binary: pc
RULE 'compile': c -> binary: cc
RULE 'edit': text -> text: vi
RULE 'edit': text -> text: ex
```

Diagram 2: A Sample Rule Base

The rule base mechanism can be used in several ways to support intelligence in the user interface. Examples of such uses are now illustrated through a discussion of the various scenarios which may occur when the rule base mechanism is in operation. In the discussion, it is assumed that the rule base in use is identical to the one shown in diagram 2.

* Unambiguous Rules

  Given that a user has produced an object called "report" (of type *unformated*) and he/she now wishes to format the document, it is necessary to select the object "report" and then apply the *format* command. To determine which UNIX tool is required, the system searches the user's rule base for a command with the <function name> *format*. In this case, there is only one such rule, and the system unambiguously knows to apply the UNIX formatting tool, *nroff*.

* Generic Rules

  Assume now that a user has an object called "prog" (of type *pascal*) and he/she wishes to compile it. As before, the object is first selected and then the appropriate command is applied (*compile* in this case). Once more, the system searches the rule base for a rule with a matching <function name> field. In this case, however, it will find two rules with *compile* as the <function name> field. The system must then attempt to resolve this ambiguity in the following way. It interrogates the type of the object "prog". Having established that "prog" is of type *pascal*, it then examines the <type mapping> field of the two ambiguous rules concerned to determine if the input type of either rule matches *pascal*. In this case, one of the rules does have a matching input type, and so the system knows to apply the pascal compiler, *pc*.

* Ambiguous Rules

  In some cases, ambiguities arise that cannot be resolved by the system. For example, consider the case where a user wishes to edit a text file by applying the function *edit;* the system will be unable to select a rule from the rule base unambiguously using the strategies described above. Either of the rules concerned with the editors *ex* or *vi* could be applied. In this case the system must rely on the user

to resolve the ambiguity (section 5.1 describes how this is achieved).

Note that the rule base assumes a one-to-one mapping from input object to output object. This is particularly well suited to the direct manipulation mode of operation.

The rule base aids the user in a number of ways. In the case of unambiguous rules, the rule base assists the user by removing from him/her the need to know the specific details of a given UNIX tool. More significantly, the rule base can be used to support generic commands in UNIX. The power of generic commands is that users are no longer burdened by the need to know the names of various editors, compilers and debuggers (etc.) which UNIX offers. Rather, the user can simply choose to compile a given object, for example, without necessarily knowing its type or the name of its compiler. Finally, it is worth noting that in each of the above examples, the *system* is able to deduce from the mapping information contained in the rule base, the type of the object produced as a result of applying a rule.

Establishing a rule base from afresh could be a daunting task for many non-expert UNIX users. Subsequently, it is our intention that the system administrator would build a standard rule base, containing rules for common UNIX activities such as document preparation and software development. The UNIX tool indicated by the <tool> field of each rule would also have associated default parameters. The user would then be free to tailor his/her rule base to personal needs. Thus, for example, if a user had developed some tools for a new application, he/she could extend the rule base by adding new rules and possibly an enriched set of object types.

2.3. Command Frames

The traditional type of UNIX interface has a single scrolling terminal in which commands are issued and results are displayed. An iconic interface however has no scrolling terminal. Rather, it creates windows itself in which to execute commands. A mechanism is required which allows the interface to decide which type of window is required for each command.

This is achieved by using the frame associated with the object of type command. The *command frame* contains a range of information pertaining to commands, including type information. The command frame also specifies the size and type of window and subwindows required by a given command, and which outputs should be displayed in each subwindow.

A command may request the use of several types of window. For example, a command which produces output but has no input may request a simple display window with a scrollbar, a command such as a visual editor may request a terminal emulator. Other commands may themselves create and control windows and need no help from the interface.

Command frames are also used to store the type information mentioned above.

An interface such as this allows commands to make effective use of windows without knowing any details of the window system.

3. The User Interface

This section describes the main interface to the system which consists of an *iconshell.* The iconshell is a window which displays the contents of a UNIX directory as a set of icons. The user has the ability to select an icon by pointing at it with the mouse and can then issue related commands.

As shown in diagram 3 below, the iconshell consists of three subwindows: the top subwindow is used for error and help messages, the middle subwindow is a control panel from which general commands may be issued, and the bottom subwindow is used to display the icons. The title line of the window contains the name of the current directory.

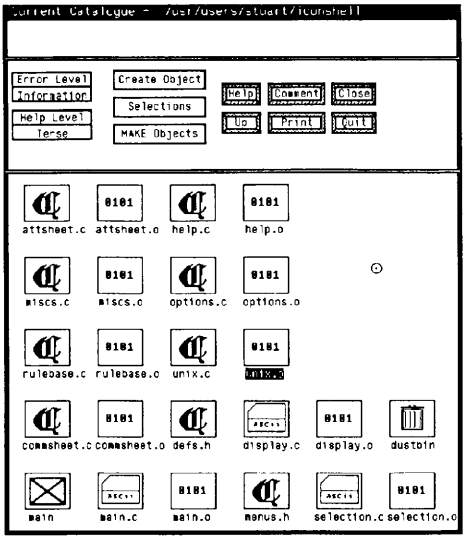


Diagram 3: An Iconshell Display

The contents of the directory are displayed as captioned icons, a different icon being associated with each object type. For example, the icon with the Oldenglish 'C' character represents a C source object, and the icon with the sequence of binary digits represents a binary object. We have attempted to associate a meaningful icon with each object type (and with commands associated with objects -- see section 5.1). Research has shown that the success of an icon to convey its intended meaning depends on the combination of a number of factors. These include the type of pictorial representation used, the underlying functionality of the object and how directly these map on to each other [16].

The control panel in the iconshell contains three types of items: *buttons, menus* and *state selectors.* Buttons are used to issue distinct commands (*Help, Quit,* etc). Menu items have a group of related commands in a pop-up menu (*Create Object,* etc). State selectors also have a pop-up menu. This, however is not used to execute commands, but rather to set the values of environment parameters. For example, the *Help Level* state selector is an attribute of the help system. It has two possible values, *Terse* and *Long,* which are contained in the pop-up menu. Setting the value to *Terse* causes

the help system, when invoked, to display a short message in the top subwindow. Conversely, setting the value to *Long* causes the help system to create a separate window with detailed help information.

These control panel items are used throughout the interface and other items are introduced in later sections. To maintain consistency in the interface, the three buttons on the Sun's mouse are always used for the same purpose, both in control panels and display windows. The left button is a selection button; it is used to select icons or issue commands. The middle button is used to display commands associated with an object. The right button provides more information about an object; this may be attributes of the object or an associated menu.

The display window in the iconshell not only displays the contents of the current directory, but also conveys other information to the user. It monitors the current working directory and any changes are immediately reflected in the display window. Objects whose types are related by a rule statement in the rule base are grouped together (therefore, in diagram 3, C source objects and their related binary objects are displayed together). If a related object becomes out of date, its icon caption is inverted (the source object *unix.c* above has been edited but not recompiled, the binary *unix.o* is therefore out of date and is shown inverted).

4. Interface to UNIX tools

An important objective of the iconshell is to support version and configuration control. This is achieved through the re-use of existing UNIX tools, i.e. SCCS and make. These should be integrated into the system while presenting a consistent interface to the user.

4.1. Version Control

One of the tools supported by the iconshell is the UNIX version control system, *SCCS* [15]. SCCS is a tool for storing and controlling changes to text files. All versions of a file are stored by recording differences (or *deltas*) between successive versions. Each time a significant change is made to a file, a new delta is recorded. To produce the latest version of a file, SCCS applies each delta in turn to the original file until all deltas have been processed. A major criticism of SCCS is that much of the responsibility for using the tool correctly and effectively remains with the users and *not* with UNIX. For example, although SCCS records and manages versions of files, it is still necessary for users to check out the required version from SCCS control when it is needed. SCCS also leaves the user to decide when to create new deltas and how to keep track of the various relationships that exist between files which together form a system configuration.

To ameliorate this situation, the iconshell automatically invokes SCCS when required. A user creates a new object in a directory through the *Create Object* menu in the iconshell control panel (see diagram 3). The iconshell issues an *SCCS create* command when a new text file is to be created. It also creates the SCCS sub-directory if necessary. Each time a text file under SCCS control is edited, the iconshell invokes SCCS to make a new version of the file and prompts the user for comments to be associated with the version.

Mechanisms are provided to allow the version history to be inspected or for commands to be issued on any version of an object. These are described in section 5.

4.2. Configuration Control

Another UNIX tool used by the iconshell is the *make* command [6]. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done.

The information required by *make* is stored in a special file called *makefile*. To use the *make* mechanism, the user must learn the format of the makefile language and maintain the makefile.

The iconshell provides a limited interface to the *make* tool through the *MAKE objects* and *Selections* menus in the control panel. A user can set dependencies between objects by selecting a group of objects in the display window of the iconshell and then selecting the *Set Dependency* option from the *Selections* menu. The iconshell will then prompt the user to select an object which the group is dependent on. The information is automatically stored in a makefile.

The user can also specify how new objects are made by selecting a group of objects and linking them together with a command. The iconshell examines the makefile to find which objects may be created from it. These objects are stored in the *MAKE objects* menu in the control panel (see diagram 3). Selecting an object from this menu causes it to be constructed if required.

5. Iconshell Subwindows - Commands and Attributes

This section describes how the iconshell is used to issue commands and to display and change the attributes of an object. The interface uses separate windows which appear when required. Section 5.1 describes the command window and section 5.2 introduces an associated window, the *command option sheet*. Sections 5.3 and 5.4 describe the attribute window and the interface to the SCCS version control mechanism.

5.1. Command Subwindow

To issue a command from the iconshell, an object in the display window is selected using the middle mouse button. This causes a command window to appear which contains all commands applicable to that object type. The command set corresponds to all rules in the rule base which have the appropriate source type in the <type mapping>. An upward closure of the type hierarchy is also performed to establish the full range of applicable commands. A typical command window is shown in diagram 4. The window is for a C source object. Once more, the window is divided into three parts: a message window, a control window and a display window.

The commands are displayed as captioned icons. Selecting an icon (with the left mouse button) causes the corresponding command to be executed.

By displaying all appropriate commands, the need for the user to remember cryptic command names or command line formats is removed. In addition, the user is

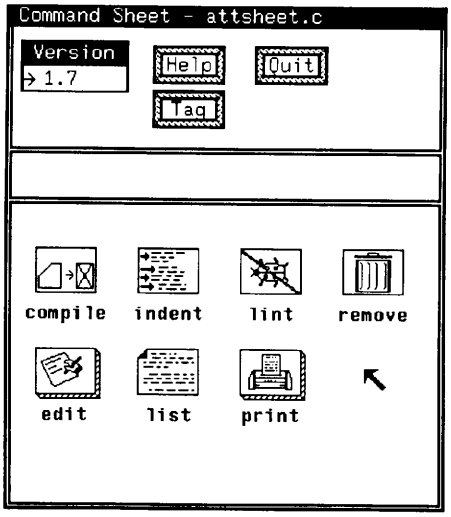


Diagram 4: A Command Window for a C Source Object

prohibited from issuing inappropriate commands (e.g. attempting to word process a C source object).

Notice that the generic name from the rule base is used as the icon caption, rather than the specific tool name (e.g. *compile* is used rather than *cc)*. Using the generic name improves the interface by reducing the complexity of the system [17]. The command window for another type of source object (e.g. a Pascal source object) may have a similar icon set although the underlying tools will be different.

As discussed in section 2.2, the system may be unable to resolve unambiguous rules in the rule base, i.e. where the <function name> and <type mapping> components of the rules are the same. For example, the *edit* command may be executed with *vi, ex, ed,* etc. In these cases, only one icon is displayed in the command window. The icon however has a shadow beneath it and selecting the icon causes the various alternatives to be displayed. In the diagram above, *print* is a multiple command and selecting it causes several different printers to be displayed. A *Back* button then appears in the control panel which may be used to restore the display.

The command window has an interface to the SCCS mechanism. The *version* box displayed in the control panel indicates which version of the current object the command will be applied to. The user may select a different version by locating the cursor in the version box and entering a new version number. When a command is selected, the new version is automatically copied into the current directory before execution. The user may choose to make this version the default version in future command windows by pressing the *Tag* button.

5.2. Command Parameters

Before a command is executed, the user has the ability to set command parameters. These correspond to normal UNIX flag or switch parameters. If flags are not set, then default values are used. These are stored in the command frames described in

section 2.3.

To set parameters, the attribute mouse button is pressed over a command icon in the command window. This causes a command option sheet to appear as shown in diagram 5 below.

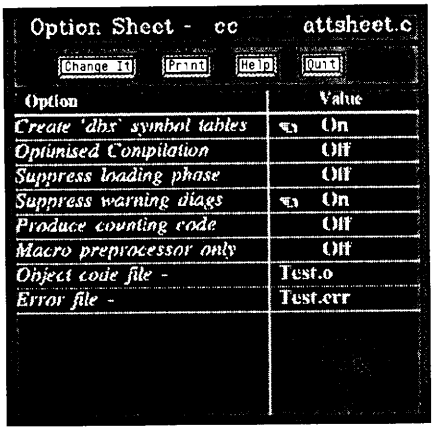


Diagram 5: An Option Sheet for the C Compiler

This is an option sheet for the cc compiler. Switch attributes can be set on or off and text attributes may be changed by typing into the appropriate area.

After attributes have been changed, they may be stored as default values for subsequent use of the tool by pressing the *Change It* button.

5.3. Attribute Subwindow

As mentioned earlier, the attributes of an object may be displayed by selecting the object with the right mouse button. This causes an *attribute window* to appear in which the information is shown. The attribute window is used to display information such as the size of an object, its access permissions, time created, etc. The attributes displayed depend on the type of the object.

The information contained in the attribute window is similar to that obtained from the UNIX *ls* command. The *ls* command (of BSD 4.2 UNIX), however, has *seventeen* possible parameters. These parameters may be applied in various combinations to return different attributes of the object. Although the command is capable of providing a large amount of information, most users tend to rely on a small subset of the parameters and, subsequently, fail to exploit the full power of the command.

The attribute window does not require the user to learn and remember parameter combinations and will always give as much information as possible. A typical attribute window is shown below in diagram 6. This is for a C source object.

The attribute window maintains the interface mode of direct manipulation by allowing the user to change attribute values at the point at which they are displayed. Attributes which are user changeable are shown in an attribute box preceded by an arrow. For example, the user may enter a new value for the *Name* attribute in diagram

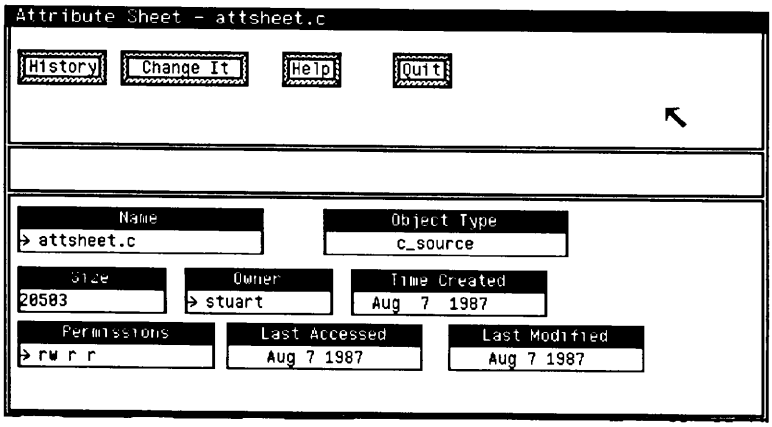


Diagram 6: The Attributes of a C Source Object

6. Pressing the *Change It* button will cause the new value to be stored. This allows all attributes to be modified in a consistent manner without requiring the user to learn and remember any command names.

5.4. Displaying an Object's Version History

The attribute window also allows information on object versions to be displayed. If an object is under SCCS control, the *History* button will appear in the attribute window (see diagram 6). Pressing this button causes the version history of the object to be displayed as shown below in diagram 7.

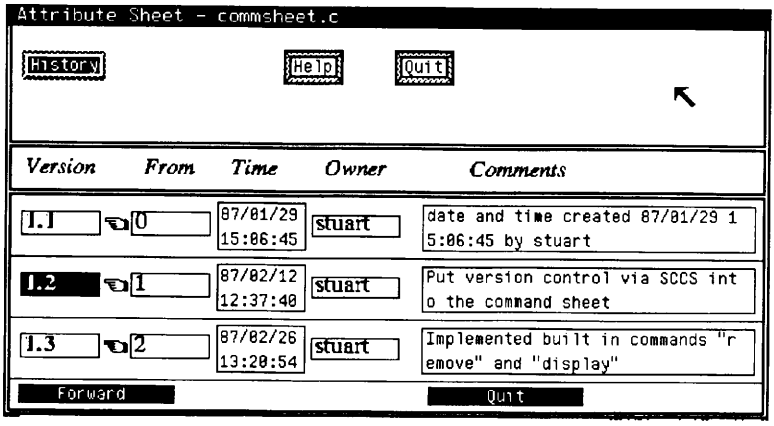


Diagram 7: A Typical Version History Window

The version which is currently stored in the working directory is shown inverted. This may be changed by selecting another version with the mouse. Once more, the

user is not required to possess knowledge of SCCS commands: only familiarity with the iconshell type of interface is necessary.

6. Implementation Status

The system has been implemented and runs on a Sun workstation. All the features described in previous sections have at least been partially implemented. A typical Sun screen layout is shown below in diagram 8. The screen has an iconshell with an associated command window and also shows several commands during execution.

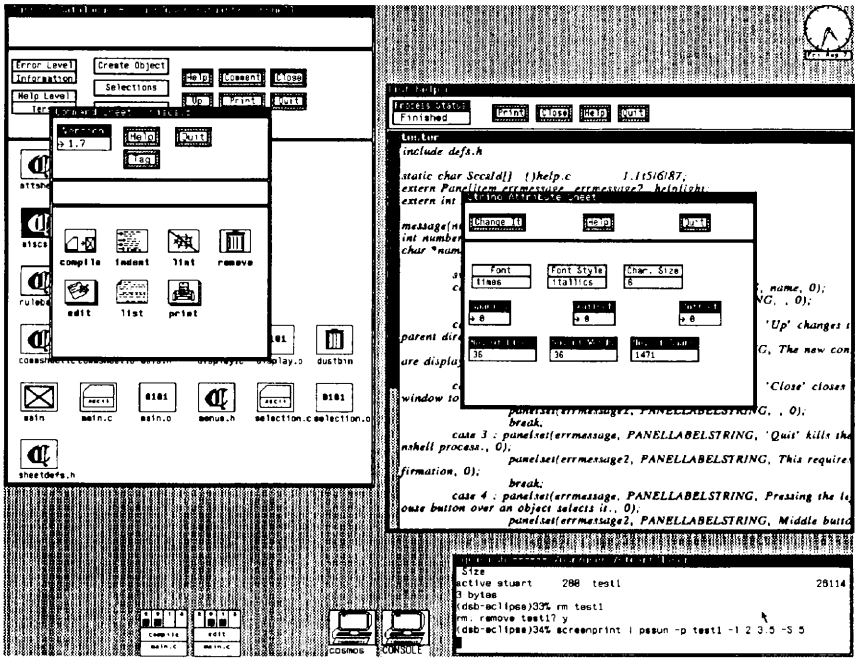


Diagram 8: A Sun Screen Layout

The current implementation provides only a limited set of execution windows. All execution windows have a control panel to provide status information and to allow commands to be applied to the window contents. A display window with a scrollbar is shown in diagram 8. This has been used to list a C source object. Notice that the window contents may be directly manipulated by the user. The diagram shows an attribute window for the displayed text. The user may change attributes such as *font style, font size,* etc. and other information such as the character count is displayed.

Two other commands are also shown during execution: compiling and editing a C source object. The windows however have been closed to icons. The current implementation uses dynamic icons for executing commands. The icons have areas which

are used as indicators. These show if an error has occurred, if a process is waiting for input or has produced output. Dynamic icons help users keep track of what they are doing and aid the monitoring and controlling of tasks operating at the same time [10].

7. The Iconshell -- Revisited

In this section, the major features of the iconshell are revisited. The main properties of the system are highlighted and the resultant benefits of the approach discussed.

Much of the power of the iconshell derives from the extra layer of *abstraction* that is added to the system. The iconshell takes the basic UNIX tool-set and command execution facilities and provides an extra layer of mapping from abstract functions on to actual UNIX tools (with this mapping being governed by the rule base). As in all Computer Science, an added layer of abstraction can encompass many extra features and provide a higher level of service to the user.

The first benefit of this mapping is that users are presented with a set of icons describing *what* tasks are appropriate rather than *how* the task is to be carried out. Intricacies of the command language can thus be hidden from the user. The more expert user, however, can always invoke the options sheet (section 5.2) for more sophisticated operations. This more abstract user interface is also ideally suited for icon display. As mentioned earlier, icons are known to be more effective if they describe operations at a more abstract level.

A second major benefit is that *genericity* can naturally be supported by the iconshell. Several related tools can be represented by the same function name with typing information being used to resolve ambiguities. This again raises the level of interaction with the user. In addition, it reduces the number of concepts to be learned by the user. In most cases, type information will successfully resolve ambiguities. However, if ambiguities remain, the user is asked to resolve the ambiguities by making a selection. As mentioned earlier (section 5.1), this is semantically meaningful at the user level.

A final major benefit stemming from the extra abstraction is the *partitioning* of the tool-set. A select group of tools (or more precisely functions) can be presented to the user in a given context. As soon as the user selects an object, the tool-set can be reduced to a much smaller number of applicable tools. This is particularly important given the current size of the UNIX tool-set and will be of considerable benefit to the naive user. One important factor is that the classification of the tool-set is achieved in a very dynamic way. The tool-set is not divided in advance into an artificial set of 'workbenches'. Rather, type information is used in real time to determine the relevant set of tools. Thus, the iconshell manages the tool-set as dynamic, over-lapping and changeable domains of interest.

It is really the three properties together which lead to the major benefits of the iconshell. The higher level of abstraction together with genericity and the automatic partitioning of the tool-set provide a strong basis for attaining a more sophisticated form of human-computer interaction.

8. Conclusions

The UNIX Operating System has come a long way in the last ten years. It is now recognised as a de facto standard in many fields of work. As such, UNIX exerts a large influence on the directions and rate of progress in computing.

As with all influential systems, the potential exists for UNIX to be either a *barrier to change* or a *vehicle for change*. Often, the difference between the two alternatives can be very subtle. There is no doubt that the pressures for UNIX to change will continue. User expectations are constantly increasing and new hardware advancements can bring unforeseen developments. It is therefore important that the UNIX community responds positively to change.

The last prime example of change with respect to UNIX, was the advent of distributed computing. It very quickly became clear in the UNIX community that earlier versions of UNIX were inadequate to meet the demands of distributed computing. For example, the inter-process communication model was inappropriate and there was little support for distributed applications. As a result, UNIX has undergone substantial revisions including the implementation of sockets [4] and the provision of distributed services such as NFS and Yellow Pages [20].

We would postulate that the next big change will be in areas concerned with the user interface. Such developments are helped by the advances in bit-map displays and the recent interest in human factors in computing. As with distributed computing, UNIX is lacking in its user interface. It is now clear that tools like the shell (and indeed command language interpreters in general) do not provide a long term answer to user interfaces. The question must therefore be posed as to the effect this will have on UNIX. Will the design of UNIX change as a result and, if so, in what way?

The system presented in this paper is one attempt to enhance the UNIX interface. It has given us some useful insights into the operating system requirements of human-computer interaction. Although it proved possible to develop the iconshell in the existing UNIX environment, the implementation would have been much easier if UNIX supported directly the association of attributes with objects.

It is possible to place this discussion in a wider context. Many recent projects in human-computer interaction [19, 18] and in other areas [13, 7, 2] are using a model consisting of objects, attributes and possibly inter-relationships. This is obviously a major thread in computing. Thus one interesting and topical question is whether UNIX can adapt to a more object-oriented world. Clearly, UNIX can never be an object-oriented system in the full sense. However, several features of objects could easily be adopted by UNIX. It is our experience that this could be of significant benefit in the provision of better human-computer interfaces.

9. Acknowledgements

The ideas concerning the rule base and frames of semantic information discussed in section 2 have been influenced by the design of the Cosmos distributed operating system [11]. Similar mechanisms have been used in Cosmos to provide support for several sophisticated programming environment functions [3]).

The style of interface using control panels, buttons, state selectors, etc. was developed as part of the Alvey Eclipse Project [1]. Further details of this interface can

be found in [14].

References

1. Alderson, A., Bott, M. F., and Falla, M. E., "An Overview of the Eclipse Project," *Integrated Project Support Environments (edited by J. A. McDermid)*, pp. 100-113, 1985.
2. Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D., "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering*, vol. 11, no. 1, pp. 43-59, January 1985.
3. Blair, G. S., Mariani, J. A., Nicol, J. R., and Shepherd, W. D., "A Knowledge Based Operating System," *The Computer Journal*, vol. 30, no. 3, pp. 193-200, June 1987.
4. Coffield, D. and Shepherd, D., "Tutorial Guide to Unix Sockets for Network Communications," *Computer Communications, Butterworth Scientific*, vol. 10, no. 1, pp. 21-27, 1987.
5. Fahnrich, K. P. and Ziegler, J., "Workstations Using Direct Manipulation as Interaction Mode - Aspects of Design, Application and Evaluation," *Human-Computer Interaction - INTERACT '84*, 1985.
6. Feldman, S. I., "Make - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, vol. 9, no. 4, pp. 255-265, April 1979.
7. Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A., and Shan, M. C., "Iris: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 48-69, January 1987.
8. Gittins, D. T., Winder, R. L., and Bez, H. E., "An Icon-Driven Interface to UNIX," *International Journal of Man-Machine Studies*, vol. 21, 1984.
9. Lodding, K. N., "Iconic Interfacing," *IEEE Computer Graphics and Applications*, March/April 1983.
10. Myers, B. A., "The User Interface for Sapphire," *IEEE Computer Graphics and Applications*, vol. 4, 1984.
11. Nicol, J. R., "Operating System Design for Distributed Programming Environments," *Ph.D. Thesis*, University of Lancaster, October 1986.
12. Norman, D. A., "The Trouble with UNIX," *DATAMATION*, November 1981.
13. PCTE, Project Team, *Overview of PCTE: A Basis for a Portable Common Tool Environment*, Esprit, 1985.
14. Reid, P. and Welland, R. C., "Project Development in View," *Software Engineering Environments*, Peter Peregrims , 1986.
15. Rochkind, M. J., "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 364-370, December 1975.
16. Rogers, Y., "Evaluating the Meaningfulness of Icon Sets to Represent Command Operations," *Proceedings of the Second Conference of the BCS HCI Specialist Group*, 1986.

17. Rosenberg, J. K. and Moran, T. P., "Generic Commands," *Human-Computer Interaction - INTERACT '84*, 1985.

18. Sibert, J. L., Hurley, W. D., and Blesar, T. W., "An Object-Oriented User Interface Management System," *ACM Computer Graphics*, vol. 20, no. 4, August 1986.

19. Smith, R. B., "Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic," *Human Factors in Computing Systems: CHI'87 Conference Proceedings*, April 1987.

20. Walpole, J., "An Overview of the Sun Network File System," *Alvey Eclipse Project Working Document ADN/WP/DH/7*, 1985.

Fast `bitblt()` with `asm()` and cpp

Bart N. Locanthi

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1 Introduction

`bitblt()` is both a fundamental building block of bitmap graphics and a demanding implementation challenge. `bitblt()`'s speed, or lack thereof, is an important factor affecting one's perception of a machine's interactive response.

Since the Alto [Thacker79], people have understood the importance of making `bitblt()` go fast and have devoted a great deal of effort and, unfortunately, ill-conceived hardware in this cause. Ill-conceived because hardware assists historically have compromised `bitblt()`'s generality in pursuit of speed.

Some processor or memory architectures inherently and actively obstruct the implementation of `bitblt()`. One that comes to mind on both counts is the IBM PC, for which the display byte order differs from that of the processor, and in which the processor isn't very good at dealing with bit-aligned data anyway. In these cases there isn't much hope but to include extra hardware, and even then one has to be very careful in the face of these built-in architectural disadvantages.

Usually, though, it seems that people don't try hard enough to make their software fast before they resort to hardware. Two problems with this approach are 1) having insufficiently high expectations for the resulting hardware and 2) not developing enough understanding about the problem to do a good job on the hardware.

This paper, then, is about squeezing the last microsecond out of software implementations of `bitblt()`. The environment for this exercise is a bitmap display based on the Motorola 68020 and a C compiler derived from an internal 68000 compiler. Both the `asm()` capability of the compiler and the C preprocessor will figure prominently.

2 Definitions

Our basic definitions haven't changed much from the time of the Blit [Pike85]:

```
typedef unsigned long Word;

typedef struct Point {
        short x,y;
} Point;

typedef struct Rectangle {
        Point min,max;
```

```
} Rectangle;

typedef struct Bitmap {
        Word *base;
        int width,ldepth;
        Rectangle rect;
} Bitmap;
```

`Rectangles` are defined to be "half-open". That is, the `Rectangle` {{0,0},{1,1}} is defined to contain only the `Point` {0,0}. This convention is similar to that for arrays in C and both makes tiling rectangles easy and minimizes opportunities for "off-by-one" errors in programming.

Bitmaps are one dimensional arrays of words that are given two more dimensions by the software that deals with them. The bitmap `width` is the number of words from the beginning of one scanline to the beginning of the next, and the `ldepth` is the log base 2 of the number of bits per pixel. Addressing is handled by the support function

```
Word *addr(Bitmap *b, Point p);
```

which returns the word in a bitmap containing a given point.

The primary bitmap manipulation function is

```
bitblt(Bitmap *dm, Point p, Bitmap *sm, Rectangle r, int fc);
```

where `sm` and `dm` are source and destination bitmaps, `r` is the source rectangle, `p` is the `min` of the corresponding rectangle in `dm` (the destination point), and `fc` denotes the function to be performed on the destination rectangle. For the purposes of this paper `fc` will be limited to the 16 bitwise logical operations defined for two variables.

3 Why it's Hard

`bitblt()` has to do a lot of things and get them all right. To begin with, source and destination bitmaps can be arbitrarily aligned. Naturally this means shifting is involved, but picking up a word on a bit boundary is more complicated than that, not to mention highly machine dependent.

A side effect of arbitrary alignment is that source and destination bitmaps often occupy a different number of words. On top of this, the first source word to be fetched in the inner loop might not actually be the first word in the bitmap, since it must correspond to the whole of the first word of the destination. Determining the inner loop count and the first source word are both easy off-by-one candidates.

`bitblt()` can alter only the destination rectangle, whose left and right edges seldom lie on word boundaries. So, the inner loop has to mask at these edges and deal with the case where both edges lie within the same word at the same time it's working with bit-aligned data and trying to go fast.

`bitblt()` has a two-dimensional version of the block move problem. The inner and outer loops must proceed left to right or right to left and top to bottom or bottom to top in order to avoid copying data over itself. In arranging these logistics one must cautious of the opportunities to make off-by-one errors.

`bitblt()` has to implement lots of different functions. This involves more than figuring out how best to write the 16 possible logical operations. Some of these operations involve only the source, or only

the destination, and so present opportunities for avoiding work. Note that source-only functions still have to fetch destination data at the left and right edges for proper masking.

`bitblt()` has to be fast. This must be attacked along several lines. It is important to put variables in registers, but most machines don't have enough registers to accommodate all the important variables in `bitblt()`. Most inner loops are short enough that it pays to unroll them. Most C compilers generate better code for `do-while` loops than for `for` loops. Characters account for the vast majority of calls to `bitblt()`, so special case character drawing loops are usually well worth the investment. Even so one has to avoid spending a lot of time deciding whether and which special case code to run.

4 Copying Bit-Aligned Words: The Source

Every machine has different capabilities for dealing with bit-aligned data. The worst case is when there are no bit field instructions and machine registers have the same width as the memory bus. One has little choice in this case but to fetch two words and shift them in two directions:

```
register Word *d,*s,m,a,b;
register l,r;
 ...
        b = *s++;
        m = (a<<l) | (b>>r);
        a = b;          /* fetch each source only once */
        *d++ op= m;
```

This is clearly a lot of work to be doing in an inner loop. This code is going to be a lot slower than a simple block move. Worse yet, the machines for which this approach is necessary usually also have small register files and take multiple cycles to shift data.

For a short time in the early 1980's the new wave of microprocessors had 16 bit memory busses but 32 bit registers. The 68000 is one such machine, and getting source data is slightly easier:[1]

```
typedef unsigned short Word;
register Word *d,*s,m;
register l,r;
 ...
        m = *s++;                       /* load 16 bits */
        asm("   ror     %m,%l");        /* rotate 32*/
        *d++ op= m;
        asm("   ror     %m,%r");        /* move to high 16 */
```

Compared to the previous example this code saves not only the two declared registers but also the temporary registers needed to hold `(a<<l)` and `(b>>r)` and the time needed to move partial results about. This approach is made possible by a subtle but critical feature of the 68000: loading the low 16 bits of a register does not affect the high 16 bits.

The 68020 is quite a nice machine for implementing `bitblt()`. The one instruction that makes the difference is `bfextu`, which can load a 32 bit register with a 32 bit quantity that may straddle a word boundary:

```
register Word *d,*s,m;
register l;
```

[1] `asm()` doesn't really allow variable names here. Call this progr. .mer's license.

```
...
        asm("   bfextu  %m,%s,0,%l");          /* get the source */
        s++;                                  /* increment pointer */
        *d++ op= m;
```

This code is faster than any of the alternatives despite the fact that `bfextu` has to go to memory twice to extract the 32 bit source. It also requires a minimum of registers. A more subtle advantage is that the initialization code has a relatively easy time of figuring out which source address is the first one.

5 Basics

Before moving on to actual `bitblt()` code, here are some simple loops and their execution times on the 68020 test vehicle. The intent here is to illustrate the basic costs of the several decisions one must make in designing inner loops. There are too many variables here to present a complete taxonomy, and peculiarities of the machine preclude conclusions based on superposition, but it is hoped that the examples presented illustrate enough to be interesting.

The first comparison is `for` loop versus `do-while`. There is no fundamental reason why these loops should take differing amounts of time, but even relatively clever compilers appear to have no compunction about putting two branches in a `for` loop:

```
/* for loop: 1.77us/loop */
register i;
for (i = 1000; i > 0; --i)
        ;

/* do-while loop: 1.13us/loop */
register i;
i = 1000;
do
        ;
while (--i > 0);
```

The next test pits `for` loop and `do-while` loop block transfers against each other.

```
/* for copy loop: 3.33us/word */
register i;
register *d,*s;
for (i = 1000; i > 0; --i)
        *d++ = *s++;

/* do-while copy loop: 2.63us/word  */
register i;
register *d,*s;
i = 1000;
do
        *d++ = *s++;
while (--i > 0);
```

Note also that the loop overhead is a significant fraction of the time spent.[2] At this point I'll write off the `for` loop and unroll the `do-while`:

[2] A newer 68020 compiler uses a byte displacement branch for this loop and comes up with a 2.0us/word loop time.

```
/* 4X unrolled do-while copy loop: 1.65us/word */
register i;
register *d,*s;
i = 250;
do {
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
} while (--i > 0);
```

Having a poor `bitblt()` engine is a mixed curse. In the following case it takes so much time to process a word, even in the simplest case, that unrolling and even `for` versus `do-while` decisions are moot:

```
/* the hard way: 4.73us/word */
register i,r,l,x,y;
register long *s,*d,m;
i = 1000;
do {
        x = *s++;
        m = (y<<l)|(x>>r);
        y = x;
        *d++ = m;
} while (--i > 0);
```

By contrast, using the 68020 `bfextu` instruction is almost as fast as a straight copy loop and unrolling is worthwhile:

```
/* bfextu copy loop: 3.35us/word */
register i,l,x,y;
register long *s,*d,m;
i = 1000;
do {
asm("   bfextu  (%s),%l,0,%m        ");
        s++;
        *d++ = m;
} while (--i > 0);

/* bfextu 4X unrolled loop: 2.63us/word */
register i,l,x,y;
register long *s,*d,m;
i = 250;
do {
asm("   bfextu  (%s),%l,0,%m        ");
        s++;
        *d++ = m;
asm("   bfextu  (%s),%l,0,%m        ");
        s++;
        *d++ = m;
asm("   bfextu  (%s),%l,0,%m        ");
```

The processor manual doesn't show this much variation due to branch displacement size but allows that the size could affect whether the instruction is executed in "best case" or not.

```
        s++;
        *d++ = m;
asm("   bfextu  (%s),%1,0,%m          ");
        s++;
        *d++ = m;
} while (--i > 0);
```

One last pair of examples shows the effect of using the 68020 decrement and branch instruction:

```
/* dbf copy loop: 2.06us/word */
register i;
register long *s,*d;
i = 999;
asm("loop:                                  ");
        *d++ = *s++;
asm("   dbf     %d2,loop        ");

/* 4X unrolled dbf copy loop: 1.56us/word */
register i;
register long *s,*d;
i = 249;
asm("loop:                                  ");
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
asm("   dbf     %d2,loop        ");
```

6 bitblt() in C

The code presented here is an example of a `bitblt()` written entirely in C that depends only on byte order. As such it is both portable and relatively slow. It should serve, then, as a standard below which one should never go.

`bitblt()` spends quite a lot of time cracking its arguments:

```
#define ONES   ((Word) 0xffffffff)
#define WSHIFT 5
#define WMASK  0x1F
#define WSIZE  32

void
bitblt(dm,p,sm,r,f)
Bitmap *dm,*sm;
Point p;
Rectangle r;
{
        register Word *s,*d;
        register Word mask,a,b,m,ls,rs;
        Word lmask,rmask;
        int i,j,qx,sw,dw,dy,nw;
        qx = p.x + (r.max.x-r.min.x) - 1;
        lmask = ONES >> (p.x&WMASK);
```

```
        rmask = ONES << (WMASK-qx&WMASK);
        ls = (r.min.x-p.x)&WMASK;
        rs = WSIZE - ls;
        nw = (qx>>WSHIFT) - (p.x>>WSHIFT);
        dy = r.max.y - r.min.y;
        sw = sm->width - nw - 2;
        dw = dm->width - nw - 1;
```

The sort of computation that goes on here is pretty independent of data structure. If instead of rectangles one chooses to think in terms of areas

```
typedef struct Area {
        Point p;        /* same as Rectangle.min */
        Point dp;       /* extent */
} Area;
```

the code would be marginally less complex here but would have surprises in store later. Best to pick a data structure convenient for use.

Before computing source and destination pointers we need to decide in which directions to scan. The top to bottom, bottom to top decision is straightforward and doesn't affect the inner loop. Scanning from right to left is necessary only if there is no vertical translation. In any event it is safe to ignore the whole scan direction question if the source and destination bitmaps differ.

```
        if (sm == dm) {
                if (r.min.y < p.y) {    /* bottom to top */
                        r.min.y += dy-1;
                        p.y += dy-1;
                        sw -= 2*sm->width;
                        dw -= 2*dm->width;
                }
                else if (r.min.y == p.y && r.min.x < p.x)
                        goto right;
        }
        r.min.x -= p.x&WMASK;   /* adjust for first destination word */
        s = addr(sm,r.min);
        d = addr(dm,p);
        if (nw == 0)            /* collapse masks for narrow cases */
                lmask &= rmask;
```

There are four basic inner loops. The two bits of distinction are right to left versus left to right, and bit aligned versus word aligned. I've put these loops into macros that take a function code expression as an argument. In the interest of compactness I unified the left and right edge treatments by convoluting the inner loop some. The first macro is the most common case, left to right scan and bit aligned:

```
#define ltor(tag,expr) case tag:\
        for (j = 0; j < dy; j++) {\
                mask = lmask;\
                a = *s++;\
                for (i = nw+1; i > 0; i--) {\
                        b = *s++;\
                        m = (a<<ls) | (b>>rs);\
```

```
                a = b;\
                expr;\
                d++;\
                mask = ONES;\
                if (i == 2)\
                        mask = rmask;\
        }\
        s += sw;\
        d += dw;\
}\
break
```

Putting a test in the middle of a buzz loop is definitely not a gateway to speed. Nor is using an expression involving a mask where none is needed. These deficiences are worth addressing when the source fetch code is improved.

The next macro also scans left to right but has a much simpler time dealing with word aligned data. This is the loop that would be executed when scrolling a screen.

```
#define wltor(tag,expr)         case tag:\
        for (j = 0; j < dy; j++) {\
                mask = lmask;\
                for (i = nw+1; i > 0; i--) {\
                        m = *s++;\
                        expr;\
                        d++;\
                        mask = ONES;\
                        if (i == 2)\
                                mask = rmask;\
                }\
                s += sw+1;\
                d += dw;\
        }\
        break
```

Code this bad would surely need to be sped up or have a special scrolling case written in. The latter is easily arranged in this case.

The `ltor` and `wltor` macros have duals for the right to left cases.

These loops are invoked from switch statements according to the function code. All the groundwork has already been laid for the left to right cases:

```
#define S       0xC
#define D       0xA
#define X       0xF

        if (ls&WMASK) switch (f) {              /* ltor bit aligned */
                ltor(0,         *d &= ~mask);
                ltor(~(D|S),    *d ^= ((~m |  *dst) & mask));
                ltor(D&~S,      *d ^= (( m &  *dst) & mask));
                ltor(~S,        *d ^= ((~m ^  *dst) & mask));
                ltor(~D&S,      *d ^= (( m |  *dst) & mask));
                ltor(~D,        *d ^= mask);
                ltor(D^S,       *d ^= (m & mask));
```

```
        ltor(~(D&S),    *d ^= (( m | ~*dst) & mask));
        ltor(D&S,       *d ^= ((~m &  *dst) & mask));
        ltor(~(D^S),    *d ^= (~m & mask));
        ltor(D,         *d);
        ltor(D|~S,      *d |= (~m & mask));
        ltor(S,         *d ^= (( m ^ *dst) & mask));
        ltor(~D|S,      *d ^= (~(m & *dst) & mask));
        ltor(D|S,       *d |= (m & mask));
        ltor(X,         *d |= mask);
}
else switch (f) {               /* word aligned */
        wltor(0,        *d &= ~mask);
        wltor(~(D|S),   *d ^= ((~m |  *dst) & mask));
        wltor(D&~S,     *d ^= (( m &  *dst) & mask));
        wltor(~S,       *d ^= ((~m ^  *dst) & mask));
        wltor(~D&S,     *d ^= (( m |  *dst) & mask));
        wltor(~D,       *d ^= mask);
        wltor(D^S,      *d ^= (m & mask));
        wltor(~(D&S),   *d ^= (( m | ~*dst) & mask));
        wltor(D&S,      *d ^= ((~m &  *dst) & mask));
        wltor(~(D^S),   *d ^= (~m & mask));
        wltor(D,        *d);
        wltor(D|~S,     *d |= (~m & mask));
        wltor(S,        *d ^= (( m ^ *dst) & mask));    /* scrolling! */
        wltor(~D|S,     *d ^= (~(m & *dst) & mask));
        wltor(D|S,      *d |= (m & mask));
        wltor(X,        *d |= mask);
}
```

Note the scrolling case `wltor(S...)` . Making scrolling fast is a simple and easily isolated matter. The right to left cases require minor adjustment before invoking the inner loops. Recall that these cases are extremely rare.

```
right:
        sw += 2*(nw+1) + 1;
        dw += 2*(nw+1);
        if (nw == 0)
                rmask &= lmask;
        s = addr(sm,Pt(r.max.x-1-(qx&WMASK),r.min.y))+1;
        d = addr(dm,Pt(qx,p.y));

        if (ls&WMASK) switch (f) {              /* rtol bit aligned */
                rtol(0,         *d &= ~mask);
                rtol(~(D|S),    *d ^= ((~m |  *dst) & mask));
                rtol(D&~S,      *d ^= (( m &  *dst) & mask));
                rtol(~S,        *d ^= ((~m ^  *dst) & mask));
                rtol(~D&S,      *d ^= (( m |  *dst) & mask));
                rtol(~D,        *d ^= mask);
                rtol(D^S,       *d ^= (m & mask));
                rtol(~(D&S),    *d ^= (( m | ~*dst) & mask));
                rtol(D&S,       *d ^= ((~m &  *dst) & mask));
                rtol(~(D^S),    *d ^= (~m & mask));
                rtol(D,         *d);
                rtol(D|~S,      *d |= (~m & mask));
```

```
                rtol(S,         *d ^= (( m ^ *dst) & mask));
                rtol(~D|S,      *d ^= (~(m & *dst) & mask));
                rtol(D|S,       *d |= (m & mask));
                rtol(X,         *d |= mask);
        }
        else switch (f) {               /* word aligned */
                wrtol(0,        *d &= ~mask);
                wrtol(~(D|S),   *d ^= ((~m |  *dst) & mask));
                wrtol(D&~S,     *d ^= (( m &  *dst) & mask));
                wrtol(~S,       *d ^= ((~m ^  *dst) & mask));
                wrtol(~D&S,     *d ^= (( m |  *dst) & mask));
                wrtol(~D,       *d ^= mask);
                wrtol(D^S,      *d ^= (m & mask));
                wrtol(~(D&S),   *d ^= (( m | ~*dst) & mask));
                wrtol(D&S,      *d ^= ((~m &  *dst) & mask));
                wrtol(~(D^S),   *d ^= (~m & mask));
                wrtol(D,        *d);
                wrtol(D|~S,     *d |= (~m & mask));
                wrtol(S,        *d ^= (( m ^ *dst) & mask));
                wrtol(~D|S,     *d ^= (~(m & *dst) & mask));
                wrtol(D|S,      *d |= (m & mask));
                wrtol(X,        *d |= mask);
        }
}
```

The actual macro invocations are trivially different from the left to right cases.

This `bitblt()` draws about 3400 characters per second, moves a 250x250 rectangle in any direction in about 23 milliseconds, and scrolls a 1024x1024x1 screen in 265 milliseconds. Faster than the terminal I'm using to prepare this paper, but not enough faster to justify the three year difference in technology.

7 Medium `bitblt()`

Most programming languages make it easy to handle special cases compactly. However, this usually entails making decisions in inner loops. Pulling decisions out of loops can make code run fast at the expense of replicating a lot of it. This is what macros are for.

In the previous section I kept some case analysis out of the inner loop by defining loop macros and instantiating them in case statements. In the interest of compactness I handled the edge conditions by bloating the inner loop some. This clearly has to change if we want speed.

Now that the form of the inner loop is under scrutiny, it is a good time to introduce the first serious machine dependency:

```
#define bfextu asm("    bfextu  (%s),%d4,&0,%m          ")
```

The `ltor` macro handles the inner loop and edge conditions separately. In doing so it orphans the case where only one word of the destination needs to be touched.

```
#define ltor(tag,edge,expr)     case tag:\
        for (j = 0; j < dy; j++) {\
                mask = lmask;\
                bfextu;\
```

```
            s++;\
            edge;\
            for (i = 0; i < nw; i++) {\
                    bfextu;\
                    s++;\
                    expr;\
            }\
            mask = rmask;\
            bfextu;\
            s++;\
            edge;\
            s = &s[sw];\
            d = &d[dw];\
    }\
    break
```

The inner loop expressions are generally much simpler than those for the edges, and often correspond to single machine instructions. The result is messy but worthwhile; `src` and `dst` have been turned into `s` and `d` to make the code fit on the page.

```
    if (ls&WMASK) switch (f) {              /* left to right, bit aligned */
            ltor(0,         *d++ &= ~mask,                  *d = 0;);
            ltor(~(D|S),    *d++ ^= ((~m |  *d) & mask),    *d++ = ~(*d|m));
            ltor(D&~S,      *d++ ^= (( m &  *d) & mask),    *d++ &= ~m);
            ltor(~S,        *d++ ^= ((~m ^  *d) & mask),    *d++ = ~m);
            ltor(~D&S,      *d++ ^= (( m |  *d) & mask),    *d++ = ~*d&m);
            ltor(~D,        *d++ ^= mask,                   *d++ ^=ONES);
            ltor(D^S,       *d++ ^= (m & mask),             *d++ ^= m);
            ltor(~(D&S),    *d++ ^= (( m | ~*d) & mask),    *d++ = ~(*d&m));
            ltor(D&S,       *d++ ^= ((~m &  *d) & mask),    *d++ &= m);
            ltor(~(D^S),    *d++ ^= (~m & mask),            *d++ ^= ~m);
            ltor(D,         *d++,                           *d++);
            ltor(D|~S,      *d++ |= (~m & mask),            *d++ |= ~m);
            ltor(S,         *d++ ^= (( m ^ *d) & mask),     *d++ = m);
            ltor(~D|S,      *d++ ^= (~(m & *d) & mask),     *d++ = ~*d|m);
            ltor(D|S,       *d++ |= (m & mask),             *d++ |= m);
            ltor(X,         *d++ |= mask,                   *d++ = ONES);
    }
```

Note the use of auto-postincrement. Some compilers complain about the use of side effects when a variable appears more than once in an expression. Our compiler happens to do the right thing here. The word aligned left to right case comes out very nicely. This will help scrolling:

```
#define wltor(tag,edge,expr)   case tag:\
        for (j = 0; j < dy; j++) {\
                mask = lmask;\
                edge;\
                for (i = 0; i < nw; i++) {\
                        expr;\
                }\
                mask = rmask;\
                edge;\
                s = &s[sw];\
```

```
        d = &d[dw];\
}\
break

else switch (f) {                       /* left to right, word aligned */
        wltor(0,        *d++ &= ~mask,                  *d = 0;);
        wltor(~(D|S),   *d++ ^= ((~*s++ |  *d) & mask), *d++ = ~(*d|*s++));
        wltor(D&~S,     *d++ ^= (( *s++ &  *d) & mask), *d++ &= ~*s++);
        wltor(~S,       *d++ ^= ((~*s++ ^  *d) & mask), *d++ = ~*s++);
        wltor(~D&S,     *d++ ^= (( *s++ |  *d) & mask), *d++ = ~*d&*s++);
        wltor(~D,       *d++ ^= mask,                   *d++ ^= ONES);
        wltor(D^S,      *d++ ^= (*s++ & mask),          *d++ ^= *s++);
        wltor(~(D&S),   *d++ ^= (( *s++ | ~*d) & mask), *d++ = ~(*d&*s++));
        wltor(D&S,      *d++ ^= ((~*s++ &  *d) & mask), *d++ &= *s++);
        wltor(~(D^S),   *d++ ^= (~*s++ & mask),         *d++ ^= ~*s++);
        wltor(D,        *d++,                           *d++);
        wltor(D|~S,     *d++ |= (~*s++ & mask),         *d++ |= ~*s++);
        wltor(S,        *d++ ^= (( *s++ ^ *d) & mask),  *d++ = *s++);
        wltor(~D|S,     *d++ ^= (~(*s++ & *d) & mask),  *d++ = ~*d|*s++);
        wltor(D|S,      *d++ |= (*s++ & mask),          *d++ |= *s++);
        wltor(X,        *d++ |= mask,                   *d++ = ONES);
}
```

The right to left cases can't use auto-predecrement (the only C decrement mode that really corresponds to an addressing mode in the machine) because the side effect destroys the meaning of the expressions. Our compiler does the wrong thing in auto-postdecrement cases anyway, and the reader is spared the resulting ugliness.

Not all single word loops draw characters, and not all characters do not span word boundaries. Still, there is no better name for the following macro:

```
#define chars(tag,expr)         case tag:\
        if ((i = dy) > 0) do {\
                bfextu;\
                s = &s[sw];\
                expr;\
                d = &d[dw];\
        } while (--i > 0);\
        break
```

Invoking this macro is comparatively easy. One simply ANDs the left and right masks together and passes an edge expression to `chars`.

This `bitblt()` draws about 5400 characters per second, moves a 250x250 rectangle in any direction in about 10 milliseconds, and scrolls a 1024x1024x1 screen in 91 milliseconds. The larger the area, the bigger the improvement. This version occupies 8152 bytes of code, up slightly from the 7800 bytes the all C version takes.

8 Killer `bitblt()`

One way of coping with code expansion is to compile it on the spot. This approach promises compactness at the expense of the overhead of compilation. This overhead is an unaffordable luxury for small cases such as characters.

Although special purpose, a `bitblt()` compiler still has to deal with compiler-like issues such as temporary register allocation and choosing address modes. In this compiler I take the easy (rhymes with 'sleazy') approach of letting the existing C compiler generate expression code so that I can concentrate on the loops and cases.

Lisp devotees are fond of saying 'program is data'. Appropriate use of macros can give C programmers this illusion:

```
#define boilerplate(f) \
f(){\
        register Word *s,*d,m,lmask,rmask;\
        register Word shift;\
        int sw,dw,dy;\
asm("   data    1");

#define asmtab(lab,stat)        \
asm("lab:");\
        stat;\
asm("   short   0");

#define code(n,c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,ca,cb,cc,cd,ce,cf)          \
boilerplate(n/**/fun)\
        asmtab(n/**/0, c0);\
        asmtab(n/**/1, c1);\
        asmtab(n/**/2, c2);\
        asmtab(n/**/3, c3);\
        asmtab(n/**/4, c4);\
        asmtab(n/**/5, c5);\
        asmtab(n/**/6, c6);\
        asmtab(n/**/7, c7);\
        asmtab(n/**/8, c8);\
        asmtab(n/**/9, c9);\
        asmtab(n/**/a, ca);\
        asmtab(n/**/b, cb);\
        asmtab(n/**/c, cc);\
        asmtab(n/**/m, cd);\
        asmtab(n/**/e, ce);\
        asmtab(n/**/f, cf);\
asm("   text");\
}

#define labtab(n)       \
extern short \
n/**/0[],n/**/1[],n/**/2[],n/**/3[],n/**/4[],n/**/5[],n/**/6[],n/**/7[],\
n/**/8[],n/**/9[],n/**/a[],n/**/b[],n/**/c[],n/**/m[],n/**/e[],n/**/f[];\
short *n[] = {\
n/**/0,n/**/1,n/**/2,n/**/3,n/**/4,n/**/5,n/**/6,n/**/7,\
n/**/8,n/**/9,n/**/a,n/**/b,n/**/c,n/**/m,n/**/e,n/**/f,\
}
```

The point here is to make an array of zero-terminated code objects indexed by function code. `boilerplate()` is there to assure correspondence between the registers used in `bitblt()` proper and the code put in the array.

Now we can compile a few arrays of expressions. As before, there are four basic inner loops, differentiated by left to right versus right to left scanning of bit aligned versus word aligned data. In this code the bit aligned cases assume the source data has been loaded into a register, whereas the word aligned cases get it directly from memory:

```
#define funs(nam,dptr,sptr)    code(nam,\
        *dptr = 0,                              /* 0 */\
        *d = ~(*dst|sptr); dptr,                /* ~(D|S) */\
        *dptr &= ~sptr,                         /* D&~S */\
        *dptr = ~sptr,                          /* ~S */\
        *d = ~*dst&sptr; dptr,                  /* ~D&S */\
        *dptr ^= 0xffffffff,                    /* ~D */\
        *dptr ^= sptr,                          /* D^S */\
        *d = ~(*dst&sptr); dptr,                /* ~(D&S) */\
        *dptr &= sptr,                          /* D&S */\
        *d = ~(*dst^sptr); dptr,                /* ~(D^S) */\
        *dptr,                                  /* D */\
        *dptr |= ~sptr,                         /* D|~S */\
        *dptr = sptr,                           /* S */\
        *d = ~*dst|sptr; dptr,                  /* ~D|S */\
        *dptr |= sptr,                          /* D|S */\
        *dptr = 0xffffffff                      /* F */\
)\
        labtab(nam);

funs(_lrbit,d++,m)
funs(_rlbit,--d,m)
funs(_lrword,d++,*s++)
funs(_rlword,--d,*--s)
```

Note how the S mode word aligned cases degenerate to auto-increment/decrement copy code that maps directly onto the 68020 architecture.

Using clever address modes isn't as important for edge conditions, which have the added complexity and time penalty of dealing with masks. The edge cases collapse to two:

```
#define edge(nam,mask) code(nam,\
        *d &= ~mask,                            /* 0 */\
        *d ^= ((~m |  *dst) & mask),            /* ~(D|S) */\
        *d ^= (( m &  *dst) & mask),            /* D&~S */\
        *d ^= ((~m ^  *dst) & mask),            /* ~S */\
        *d ^= (( m |  *dst) & mask),            /* ~D&S */\
        *d ^= mask,                             /* ~D */\
        *d ^= (m &= mask),                      /* D^S */\
        *d ^= (( m | ~*dst) & mask),            /* ~(D&S) */\
        *d ^= ((~m &  *dst) & mask),            /* D&S */\
        *d ^= (~m & mask),                      /* ~(D^S) */\
        *d,                                     /* D */\
        *d |= (~m & mask),                      /* D|~S */\
        *d ^= (( m ^ *dst) & mask),             /* S */\
        *d ^= (~(m & *dst) & mask),             /* ~D|S */\
        *d |= (m &= mask),                      /* D|S */\
        *d |= mask                              /* F */\
)\
```

```
        labtab(nam);

edge(_left,lmask)
edge(_right,rmask)
```

Now we have the pieces to construct code templates for the inner loops. The form of the templates will be a sequence of literal code segments and pointers to previously defined code tables. The `bitblt()` compiler will depend on these templates having a fixed format and will copy the code and follow the pointers in building the actual doubly nested loop it will jump to later. The template format also calls for unrolling the inner loop four times:

```
#define X4(x)  x x x x
#define        acc(stat)       stat; asm("     short   0")
#define bfextu asm("   bfextu  (%s),%shift,&0,%m    ")

#define template(nam,c0t,f0,c0b,c1,f1,c2t,f2,c2b)      \
boilerplate(fun/**/nam)\
asm("nam:");\
        do {\
        acc(c0t);\
asm("   long    f0");\
        acc(c0b);\
        X4(acc(c1); asm("       long    f1");)\
        acc(c2t);\
asm("   long    f2");\
        acc(c2b; } while (--dy > 0));\
}
```

Invoking the `template` macro involves the use of some bizarre spacing for the sake of the C preprocessor; I hope the result isn't too unreadable. This is the left to right, bit aligned loop, which uses the left edge, scans left to right with `lrbit`, and finishes with the right edge for each scan line:

```
template(_lrshift,
        bfextu;
        s++,_left,
        d++,
                bfextu;
                s++,_lrbit,
        bfextu;,_right,
        (char *) s += sw;
        (char *) d += dw
)
```

Note the casts at the end. Although our C compiler is clever enough to use a load-effective-address instruction to implement `s += sw;` , this instruction is slower than the simple add instruction that results from adding prescaled `sw` and `dw` quantities to character pointers. Not all compilers allow casting of lvalues.

Here are the remaining three cases:

```
template(_rlshift,
        bfextu,_right,
        ,
```

```
                s--;
                bfextu,_rlbit,
        s--;
        bfextu;
        d--,_left,
        (char *) s += sw;
        (char *) d += dw
)

template(_lrword,
        m = *s++,_left,
        d++,
                ,_lrword,
        m = *s,_right,
        (char *) s += sw;
        (char *) d += dw
)

template(_rlword,
        m = *s,_right,
        ,
                ,_rlword,
        m = *--s;
        d--,_left,
        (char *) s += sw;
        (char *) d += dw
)
```

The four templates defined so far all assume the destination takes at least two words. The one word destination case is special in several respects. First, there is no need for auto-increment/decrement, since only one source and one destination will be fetched per scan line. Second, it requires only a singly nested loop. Third, and most important, this case most often corresponds to the use of `bitblt()` to draw characters.

Characters involve so little memory that the overhead of compiling would be significant compared with the time spent actually moving bits. Even decoding the arguments to `bitblt()` and deciding on the character case is significant. There just isn't time to compile character code, but fortunately there is only one case for each function code. All `bitblt()` has to do in these cases is index by function code and jump right in:

```
#undef asmtab
#define        asmtab(lab,stat)        \
asm("lab:");\
        bfextu;\
        (char *) s += rmask;\
        stat;\
        (char *) d += dw;\
asm("   dbf     %dy,lab");\
asm("   rts")

edge(_chars,lmask)
```

The rest of this `bitblt()` is straightforward, messy and not included here. The reader may complete it as an exercise.

Killer `bitblt()` draws about 5400 characters per second, moves a 250x250 rectangle in any direction in 5.4 milliseconds, and scrolls a 1024x1024x1 screen in 46 milliseconds. We got the character case right the last time, but there was still plenty of room for improvement in handling larger areas. Equally dramatic is the reduction in code size to 2764 bytes.

The results for summarized here for comparison.

version	all C	medium	killer
chars/sec	3400	5400	5400
move 250x250 (ms)	23	10	5.4
scroll 1024x1024x1 (ms)	265	91	46
size (bytes)	7800	8156	2762

9 Conclusion

`bitblt()` is hard to get right and hard to make go fast. Its implementation is also extremely machine dependent. The programming ethics of simplicity and compactness are especially important in the face of so many things to get wrong and the absolute requirement for speed. `asm()` and cpp are strange but surprisingly effective tools in this endeavor.

For all its hackiness, I claim that the result is reasonably portable across processors, probably more so than across assemblers and C compilers. This version took me less than a week to write and has so far been the mostly easily maintained version of `bitblt()` that I have yet written. And it's *fast!*

References

[Pike85] R. Pike, B. N. Locanthi, J. F. Reiser, "Hardware/Software Trade-offs for Bitmap Graphics on the Blit", *Software - Practice and Experience*, vol 15, no 2, 1985.

[Thacker79] C. P. Thacker et al, "Alto: A Personal Computer", in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, A. Newell, ed., McGraw-Hill, 1982.

DES - Support for the Graphical Design of Software

Stephen Beer and Ray Welland
Dept. of Computer Science
University of Strathclyde
Glasgow G1 1XH, Scotland
stephen@uk.ac.strath.cs

Ian Sommerville
Department of Computing
University of Lancaster
Lancaster LA1 4YR, England

Software design methods such as JSD, MASCOT and Structured Design have been in existence for some time now and most of these methods utilise graphical as well as textual notations for describing designs. These graphical forms quickly convey the overall structure and interconnections of a design more easily than straightforward textual descriptions. However, the full impact of design diagrams employing these graphical notations has been severely restricted in the past due to the lack of automated facilities for production and maintenance of such diagrams. This contrasts markedly with CAD developments in other engineering fields.

This paper describes DES (the Design Editing System) - a system which investigates how such graphical support may be provided in a *generalised* way for software engineering purposes. DES comprises of 3 tools: a shapes editor (SHAPES) for defining the shapes of a method, a language (GDL) for describing a software design method and a graphical design editor (DE) which is driven by tables generated from the first 2 tools. The system is implemented in C on a Sun workstation using the pixrect graphics layer and the panel user interface package.

At a very high level of abstraction a design diagram can be viewed as a number of symbols and a number of rules concerning the physical and logical constraints on those symbols. A novel feature of DES is that it is not geared towards any specific method. Rather, the tool builder defines the syntax, semantics and shapes of a design method using the high level tools SHAPES and GDL. This increases the applicability of the system since MASCOT and JSD users, for example, may utilise the same system facilities.

This paper presents each of the tools, emphasising how method specific checks may be specified in GDL and enforced during design editing sessions.

1 Introduction

A major aim of software engineering is concerned with bringing methodical practices into the various phases of the software life cycle. At the present moment there are a very wide variety of methods within the phase of software design. Examples of such methods include JSD [Jackson83], Structured Design [Constantine79], Petri-Nets [Peterson81] and MASCOT [MASCOT80]. Many of these design methods have associated graphical techniques to complement or replace textual design descriptions. Structured design, for example, provides for two diagram types: structure charts and dataflow diagrams.

These graphical techniques have been in use for some time now but are not as widespread as one initially might expect. One reason often put forward is that a design diagram lacks formality and cannot capture the same amount of detail as a straightforward textual description. This is true in many cases, but their major use is in conveying overall system structure and interconnections in a more easily digested form. The absence of *automation* from the design process, we believe, is a key reason as to why these graphical techniques are not in common use.

To date, support for the creation and maintenance of design diagrams in other fields of engineering has been extensive. Electronics has seen systems supporting the design of logic circuits which can then generate pin connections. CAD in the area of mechanical engineering allows metal components to be designed graphically; the design information is then passed to a cutting machine which automatically cuts the part to specification. Support for graphical techniques within the area of software engineering, however, has been acutely scarce.

One reason for this is the sheer number of methods that already exist and the number of methods likely to be developed in the future. Many reasons can be stated for the existence of each, among these being the area of application. For example, JSD is more suited for the area of data processing tasks whilst MASCOT is aimed particularly at real time embedded applications.

The work described in this paper has been taking place within the context of ECLIPSE [Alderson85] which aims to provide an integrated project support environment for different approaches to the software process and tools to assist in software design. A requirement of ECLIPSE is that many different graphical techniques should be supported and that designs created can then be captured in the project database and manipulated by other tools. A common user interface across design support tools is another requirement.

One approach to providing design support is to build a specific design editor for each method from scratch. However, given that we have a situation where many different methods *do* exist and more *will* be devised it seems much more sensible to abstract the general concepts of design methods into a design editing system which has the capability of being tailored for a specific method.

Our work here at Strathclyde has been concerned with just that. We have been experimenting and building a design editing system which can be tailored at a very high level to cater for many different methods. This paper will describe our system and emphasise how a design method can be described more formally.

The next section in the paper presents an overview of our system and compares some related work in the area of describing software design methods. This is followed by brief descriptions of each of the three constituent tools. The task of checking a design diagram finally concludes the paper.

2 A Design Editing System

A large number of graphical design techniques can be characterised as exhibiting a graph-like structure composed of design objects. Each design object can be classified into one of the following:

node	denotes a software component, state, etc.
link	denotes flow of control or data, etc.
label	denotes the textual and graphical annotations of a design

These techniques whilst sharing the common properties of graphs tend to differ with respect to:

- the actual symbols used to represent design objects
- rules concerning how a design diagram may be constructed

Some of these rules refer to syntactic constraints such as ensuring that the name label of a node appears completely within the perimeter of the nodes' symbol. Other rules refer to the semantic properties of a design - for example, in designing a dataflow diagram it is generally considered "good" design to have no more than 8-10 nodes on any one diagram.

The DES approach to providing design support for different methods is to provide tailoring tools to describe the essential features of a method and a generic design editor which is configured by this method specific information. This is achieved by providing the following three distinct tools (depicted in Figure 1):

SHAPES	a graphical shapes editor allowing the representation of each design object to be defined.
GDL	a language for specifying the "grammar" of a method.
DE	a generic design editor configured by tables from the previous tools

In this type of system there are two types of user. The **method administrator** is the person responsible for *setting up support* for a design method. The **end-user** is a person who uses a design editor to *create design diagrams*. The tasks performed by the method administrator in tailoring an editor for a specific method are:

- Describe the types of design objects (node, links and labels) by writing a GDL description
- Define how each object is to be represented using the SHAPES editor
- Tables created using the GDL compiler and SHAPES are then used as input to the generic design editor, thus configuring it for a specific method.

End-users can now create design diagrams using the tailored DE. Diagrams may be stored and retrieved at any time thus ensuring that the task of diagram maintenance is made easier.

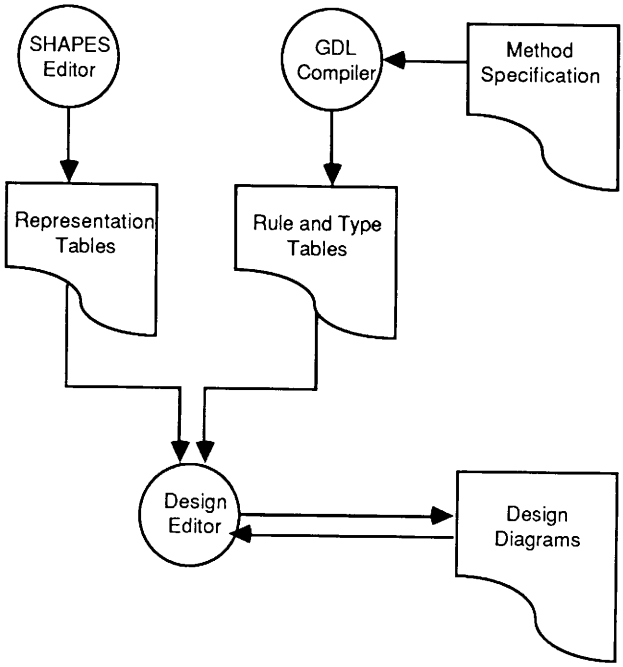


Figure 1 - The Design Editing System

The tailoring technique has been used successfully in many other areas where general properties are abstracted and specific differences described. A particularly useful and commonly known analogy is available from *compiler-compiler* systems. Such systems are used to automatically generate compilers for languages specified with high-level tailoring tools. In this area the observation has been made that compilers in general possess the same common stages of scanning, lexical analysis, semantic analysis, etc.. What they actually differ in are:

- the lexical tokens of the language (eg. keywords, identifiers, operators, etc.)
- rules referring to how a syntactically legal program may be constructed from the available lexical tokens

The **lex** [Lesk75] and **yacc** [Johnson75] tools of UNIX provide a means of defining such language specific features. The lex tool takes a specification of the legal tokens of a language and generates C code to perform the scanning and lexical analysis phase of compilation. The yacc tool takes a specification, describing the grammatical rules for constructing a legal program, and generates code to accomplish the remaining stages of compilation. The lex and yacc generated code can then be linked, together with user-written code, to produce a working compiler for the specified language.

The DES approach in providing method support is similar. The major difference is that the output from the tailoring tools are *tables* encapsulating method information rather than C code. These tables are then used to "drive" a generic design editor.

Related work in providing graphical support for design methods shows slightly different approaches. The work of [Woodman86] is inspired from the development of picture grammars in the field of pattern recognition. The principle idea is that specific patterns can be described using a grammar and the task of recognising a new scene corresponds with trying to parse it according to known pattern definitions. Rather than return a value of "parse failed" or "syntax error", a weighting is returned and used in deciding how close a scene fits a known pattern. Their application to software engineering diagrams is similar. Essentially it consists of specifying what a dataflow diagram, for example, should look like. The grammar approach in DES is similar except that the definition of the methods' lexical tokens is separated from the specification of the method rules. In their work the symbol definition is an integral part of the grammar.

From Figure 1 it can be seen that the DE is essentially syntax driven - the syntax of the method "drives" the editor. Another grammar based approach in defining a method is contained in the SEGRAS-Lab [Kramer86]. This system provides graphical support for Petri nets within a syntax directed editing environment. Their grammar normally used for generating textual syntax directed editors has been extended to allow context-sensitive constraints to be specified. The end product here is a syntax-directed graphical editor. Our approach differs here in that actual diagram construction by an end-user is accomplished with a non-restrictive interface.

3 Defining Method Symbols - SHAPES

The purpose of the SHAPES editor is to allow a method administrator to define the representation (symbols) of each design object of a design method. Once created, the symbols can be stored in method specific libraries which are used as input to the generic design editor. The SHAPES user interface (Figure 2) consists of:

- a control panel for selecting commands
- a **shapelist** for storing/selecting shapes
- a scrollbar for scrolling through the shapelist
- a drawing area for constructing new shapes (symbols)

Most interaction is via a mouse device except for the task of naming a symbol prior to storing it on the shapelist. The shapelist initially consists of a number of **primitive** shapes. This list can be extended by adding new user-defined symbols. The primitive shapes provided are text, ellipse, rectangle (right-angled and round cornered), triangle, diamond and line with circle and square being special cases of ellipse and rectangle. Whilst it is not possible to produce every conceivable shape of every method from a combination of these primitives it is possible to generate a very large percentage of them.

To define a new shape the method administrator first selects a primitive shape from the shapelist using the mouse. This shape is then instantiated in the drawing area by defining its enclosing boundary, again with the mouse. Any number of shapes can then be added and all, or any subset, of these can be moved, stretched or deleted. Once created, a symbol can be entered into the shapelist along with its name.

At any time during a SHAPES session the user may store the current shapelist in a shape library for later use as input to the DE.Therefore a shapes library consists of a number of user-defined shapes each identified by a name and described in terms of basic, primitive geometric shapes.

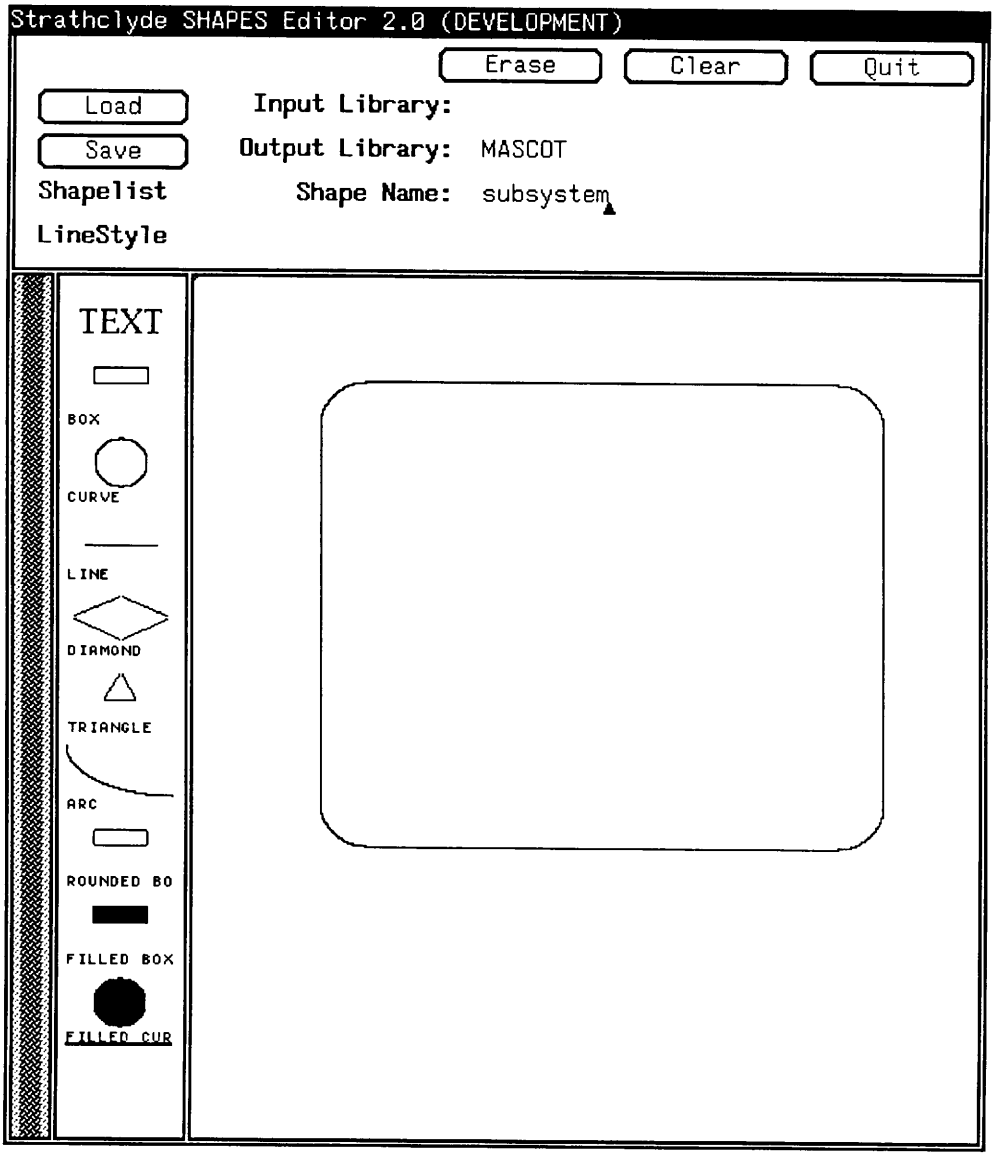


Figure 2 - The SHAPES Editor

4 Describing a Method - GDL

The notation we have developed for describing a software design method is the Graph Description Language. A method administrator describes a method by writing a GDL

description which is then transformed into tables using the GDL compiler. The compiler has been constructed using lex and yacc and also employs the facilities of cpp - the C preprocessor.

The best way of describing the GDL is by example and the following fragment describes some of the design objects of the MASCOT design method used in an editing session shown in Figure 3.

```
type PATH           is LINK ( src : NODE; dst : NODE )

type JUNCTION       is NODE ( parent : owned by  SUBSYSTEM;
                              in_path : in set of PATH;
                              out_path : out set of PATH )

type PORT           is JUNCTION
type WINDOW         is JUNCTION

type SUBSYSTEM      is NODE ( junctions : owner of set of JUNCTION )

for PORT            use SYMBOL ( MASCOT.port )
                    ++ ACCESS_INTERFACE ( STRING )
                    ++ JUNCTION_NAME ( STRING )

for  SUBSYSTEM      use SYMBOL ( MASCOT.subsystem )
                    ++ TEMPLATE_NAME ( STRING )
                    ++ COMPONENT_NAME ( STRING )

assertion Junc_name_enclosed ( SUBSYSTEM ):
                    inst i:
                    forall j; Member ( Dependents ( i ), j ) and GetType ( j ) = JUNCTION:
                    Encloses ( GetLabel ( i, SYMBOL ), GetLabel ( j, JUNCTION_NAME ) )
```

The type JUNCTION is in fact a place holder in the type hierarchy so as to avoid unnecessary repetition.

Types

In describing a design method the main task is to assign some type to each design object of the method (ie. node, link or label). The base types of the language are **node** and **link** (ie. the basic constructs of any graph) and a hierarchy of types is formed from these. In the example shown the type PATH is introduced stating that any instance of a PATH should have parameters *src* and *dst* of type NODE. The type SUBSYSTEM is based on NODE and has only dependent (or child) nodes. A child node is always associated with some parent node and any operations affecting the parent also affects its children. In our example a child node of a SUBSYSTEM node is of type JUNCTION which in effect is a PORT or a WINDOW. Having introduced the node and link types attention is now turned on how label types are defined.

Labels

The *for-use* declaration is the construct for specifying the label types to be associated with a node or link type. In the example, a PORT type has one symbolic (iconic) label and two textual labels.The reserved word SYMBOL indicates that this label is the one used to represent an instance of the PORT type on a diagram. The name *MASCOT.port* refers to a symbol named *port* stored within a shape library named *MASCOT*.

Assertions

The assertion construct is used to define the syntactic and semantic constraints of a design method. In the example, we have chosen to describe the constraint that the JUNCTION_NAME label of every child JUNCTION node of a SUBSYSTEM must be completely enclosed within the SUBSYSTEM boundary. This is a syntactic constraint which ensures that in Figure 3, for instance, the labels "access1" and "access2" both appear within the "Processing Subsystem" symbol boundary. Semantic constraints could refer to the number of JUNCTION's present within a SUBSYSTEM, so as to reduce design complexity, for example. Further details of the GDL and its capabilities can be found in [Beer87 and Sommerville87].

5 The Generic Design Editor (DE)

The envisaged user of a DE tailored for a certain method is a software designer having knowledge of the design method. Rather than interacting via simple graphical drafting terminology (eg. box, circle) the DE uses the same terminology as the designer. A designer selects a TEMPLATE or a PROCESS, for example, and adds it to a design diagram as opposed to selecting a box or a line. Hence the designer interacts in terms of the **design objects** of a method.

The major facilities of the DE allow the user to:-

- add, move or delete images representing design objects
- create a design diagram larger than the designers' workstation window,
- view the total diagram at a reduced size
- annotate the diagram only, not the underlying design, with text, boxes or lines.
- utilise a grid facility for aligning objects

The object oriented approach to user interaction has been pursued throughout the development of the DE. The functions available to the designer are applied to a currently selected object from the design. The current selection may consist of a node, link, label or combination of. Functions are applied consistently across all objects wherever it is sensible to do so. It is nonsense to edit the graphical symbol of a node but sensible to textually edit its labels.

The implication here is that the designer points at an object to make it the current selection and then applies some editing function, such as delete or move. The converse to this philosophy is the function-oriented approach where a function is first selected followed by the objects to which the function is to be applied. The choice of interface essentially depends on whether user-interaction is more natural with the object or the function.

The DE has been implemented on a SUN workstation running under the Suntools window environment. The user interface (see Figure 3) consists of a tool window subdivided into the following four subwindows:

- drafting area where a design is constructed,
- a control panel giving access to the editing functions and object types,
- two subwindows containing scroll bars. These allow the the drafting area to be moved around the total area of the diagram.

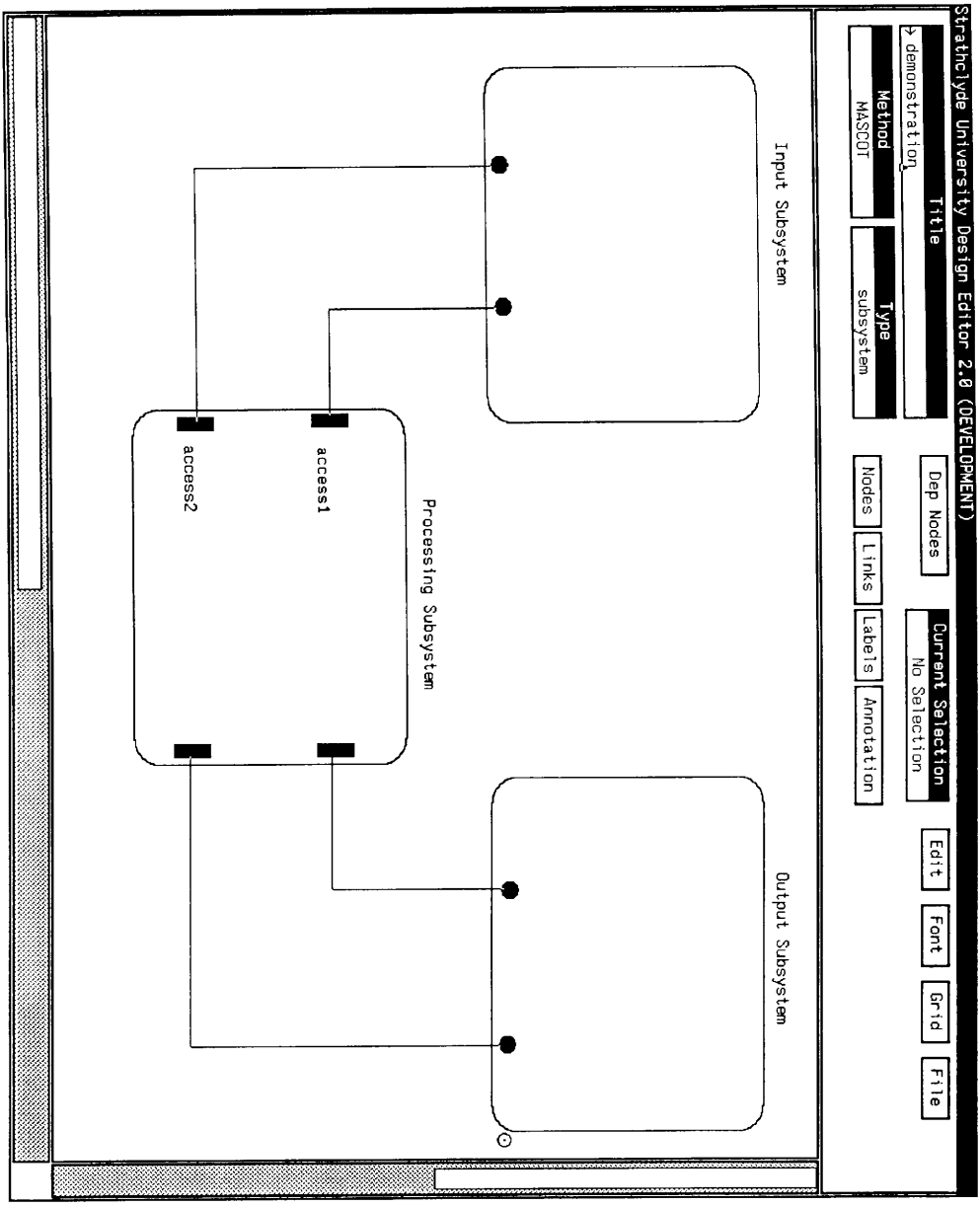


Figure 3. The DE User Interface

Editing functions are selected from a control panel [Reid86] containing pull down menus and "soft" buttons. The designer selects an node, link or label type from the menu and then adds this to a design by fixing the position for its graphical image on the diagram.

The concept of a **label** is used within the DE to associate either a name or a graphical symbol with a node or link. Therefore, the label is a generic object for capturing textual and graphical descriptions. Labels have a defined enclosing boundary, contain a value and may be manipulated in the same fashion as other objects of the design.

As mentioned previously, the DE has built-in knowledge that the design must be in the form of a graph. The one restriction implicitly enforced by the DE is that a link must originate from and end at a node. This prevents a design from being created where dataflow links, for example, lead to or originate from nowhere.

This graph knowledge is used in other situations. For example, each node can have an associated set of input and output links and labels. When a node is selected as the current object then its links and labels are automatically selected as well. Subsequent functions applied to a node are also applied to its links and labels, for example, if a node is deleted then all its associated links and labels are also deleted- it makes no sense for them to reference a non-existent node. In the same way a move operation automatically moves all links and labels associated with a node.

6 Design Checking

In a software design editing system the provision of drafting facilities for automating diagram production is important. What is even more important, from the point of view of ECLIPSE, is that the underlying design is captured in the project database. A neat looking diagram is of no use, and indeed may be harmful, if it is incorrect. Design checking therefore is a major function of the DE and is one aspect in which it differs from a straightforward drafting tool.

Other design support tools have demonstrated the need for method specific checking of a design [Jones86, Stephens85] but a novel feature of the DE is that checking is enforced at three levels and at various times throughout an editing session. These checks are all closely integrated with the GDL description of the method being supported and the three levels can be defined as :-

- "connectedness"
- layout and semantic constraints
- completeness checking

In the case of a node the strong typing of parameters can be used to enforce correct design automatically. This is so because each node has associated links defined as being either **in** or **out**. Subsequently, in the DE, at the time when a link is added to a design the parameter lists of both the source and destination nodes can be checked. This check ensures that the link type is consistent with the legal types of the source node's **out** links and also with the destination node's **in** links.

The method rules to be enforced in the DE are called **assertions**. These are compiled by the GDL compiler into rule tables which drive the DE. These assertions could be enforced at different times in an editing session, as described later, but we believe that the best approach is to allow the user to specify **when** checking should take place. At this time any object in error is highlighted and made the current selection. An appropriate message is displayed in the control panel and the designer can then apply functions to the erroneous object to correct the design.

Type checking can be enforced mainly through assertions if the link parameters are specified as the generic type NODE and appropriate assertions on connections are written. Alternatively, a strongly typed description removes the need for such assertions but increases the number of checks each time a link is added to the design. This GDL trade-off effectively means striking a balance between continuous checking (closer to syntax-directed editing) and user-initiated checking.

In a GDL representation expression a label can be specified as being either compulsory or optional. This information on the optionality of a label is transmitted to the DE through the GDL generated tables and the presence of mandatory labels is checked at an appropriate time. This is an example of a **completeness** check which can only be carried out under the control of the user.

In interacting with the DE the end-user has freedom to construct a design diagram in any appropriate manner. This is in contrast to other editing systems which provide a syntax-directed user interface [Kramer86]. Design checking occurs in a non-obstructive manner. If an assertion is violated then an appropriate warning message is output and the offending object highlighted. The end-user can then choose to fix or ignore this error rather than being forced into a fixing it before proceeding with the design.

The specific timing of checks is a contentious subject. The basic philosophy behind the DE is that the designer should be given as much freedom as possible to construct a design diagram is his or her own way. This is achieved by provided an object-oriented, modeless interface with most of the design checking initiated at user specified times. Checking in the DE can be classified on its timing during an editing session as follows:

1. **implicit / restrictive** There is implicit continuous checking throughout an editing session because certain editing operations are restricted at certain times. For example, at the point when a node is selected only certain label types are made available to the designer. This ensures that, by restriction, the designer cannot associate a label of incorrect type with a particular node.

2. **immediate** As soon as an editing operation alters a design certain assertions will be executed immediately. For instance, these would include checking that a link has source and destination nodes of the correct type.

3. **user-initiated** Rather than providing a syntax directed editor where the user is forced to construct a design in a certain manner, the DE allows the designer freedom to develop a design in whatever manner is favoured. Checking can be called at the designers' convenience and may include assertions concerning completeness and consistency, assertions about spatial arrangement of objects and completeness of label sets.

7 Conclusions

Presented in this paper has been a description of some applicative research into providing automated support for graphical diagrams within software design methods. The system can be tailored at a very high level to support different methods. So far, descriptions for methods such as JSD, MASCOT, state transition diagrams and dataflow diagrams have been defined.

The novel aspect of this work is the ability to be able to specify and enforce method specific checking within a **generic** design environment. The true worth of a production-level version of this work would be obtained within an integrated project support environment requiring many different design methods to be supported.

Acknowledgements

The work described here was funded by the Alvey Directorate, UK. Thanks are due to our collaborators in the ECLIPSE project namely Software Sciences Ltd., CAP Industry Ltd., Learmonth and Burchett Management Systems Ltd., and the University College of Wales at Aberystwyth.

Personal thanks are also due to Alistair Blair, Stevie Keith and Jim Reid for providing good, constructive criticism of this paper.

References

[Alderson85]
Alderson A., Falla M. E., Bott F., *"An Overview of ECLIPSE"*, Integrated Project Support Environments, J. McDermid (ed) Peter Perigrinus London (1985)

[Beer87]
Beer Stephen, Welland Ray, Sommerville Ian, *"Software Design Automation in an IPSE"*, To appear in Proc. of 1st European Software Engineering Conference, Strasbourg, France (Sep 1987)

[Constantine79]
Constantine L. L., Yourdon E., *"Structured Design"*, Prentice-Hall (1979) Englewood Cliffs, NJ

[Jackson83]
Jackson Michael, *"System Development"*, Prentice-Hall(1983)

[Johnson75]
Johnson S. C., *"Yacc: Yet Another Compiler Compiler"*, Computing Science Technical Report, No. 32 Bell Laboratories Murray Hill, New Jersey (1975)

[Jones86]
Jones John, *"MacCadd, An Enabling Software Method Support Tool"*, Proc of 2nd Conference of British Computer Society Human Computer interaction Specialist Group, Harrison (ed) pp. 132-154 Cambridge University Press (23-26 Sep 1986)

[Kramer86]
Kramer Bernd, *"Interactive Graphical Specification Using he Syntax-Directed SEGRAS* Nineteenth Annual Hawaii International Conference on System Sciences, Bruce D. Shriver (ed) pp. 420-429 (1986)

[Lesk75]
Lesk M. E., *"Lex - A Lexical Analyser Generator"*, Computing Science Technical Report, No. 39 Bell Laboratories Murray Hill, New Jersey (Oct 1975)

[MASCOT80]
MASCOT, *"The Official Handbook of Mascot"*, Mascot Suppliers Association Malvern, UK (1980)

[Peterson81]
Peterson James L., *"Petri Net Theory and the Modeling of Systems"*, Prentice-Hall (1981)

[Reid86]
Reid P., Welland R., *"Software Development in View"*, Software Engineering Environments, Ian Sommerville (ed) Peter Peringrinus London (1986)

[Sommerville87]
Sommerville Ian, Welland Ray, Beer Stephen, *"Describing Software Design Methodologies"*, The Computer Journal, Vol. 30 No. 2 pp. 128-133 (1987)

[Stephens85]
Stephens M, Whitehead K, *"The Analyst - A Workstation for Analysis and Design"*, Proc of 8th International Conference on Software Engineering, IEEE Computer Society Press London (28-30 August, 1985)

[Woodman86]
Woodman M, Ince D, Preece J, Davies G, *"A Grammar Formalism as a Basis for the Syntax-Directed Editing of Graphical Notations"*, Open Univeristy Technical Report 86/19, (1986)

Cleaning Up UNIX† Source
- or -
Bringing Discipline to Anarchy

David Tilbrook
Zalman Stern

Information Technology Center
Carnegie Mellon University
dt@andrew.cmu.edu
zs01@andrew.cmu.edu

ABSTRACT

Many people are coming to the realization that the distribution of UNIX software is not just a matter of creating a tar tape. Approaches to software and distribution management vary from simple disciplines, conventions and procedures to full scale Integrated Programming Support Environments (IPSEs). This paper discusses some of the problems and issues involved in the management of software, concentrating primarily on the exporting of source to remote locations.

1. Introduction

This paper is a discussion of techniques for managing medium size software distributions (e.g. 4.3bsd[1]) and for the installation of that software at remote locations. For the most part, the paper is based on the strategy originally formulated by David Tilbrook to manage and distribute the D-tree software [Tilbrook 86]. In this paper the authors attempt to extract the relevant points that should be applicable to most UNIX based software projects. The Information Technology Center plans to use this strategy to manage the Andrew software produced at Carnegie Mellon University. If this endeavor is successful, these techniques will be applied to other software developed at CMU as part of the effort to produce a CMU distribution tape.

The paper is divided into eight sections. The next two sections introduce the goals and objectives of Software Mangement and some of the guidelines to be followed in implementing a Software Management system. These sections are followed by a discussion of some of the more important aspects of such an implementation, namely the key personnel and their responsibilities, the contents and structure of the supporting database, and some of the more important tools. The final section presents some prognostications as to the effectiveness of the strategy and its future.

2. What is Software Management

"Software Management" is the discipline of managing, maintaining, and distributing a body of software in source form. The ultimate goal is unremarkable and painless installation of software and its subsequent upgrades at a remote site. The notion behind this is that software should be judged on its inherent worth, not the amount of hassle involved in getting it to run.

---

† UNIX is a trademark of Bell Laboratories.

[1] Approximately 4300 files and 850,000 lines of code.

Software Management is a different discipline from Software Engineering. While, the primary goal of both Software Management and Software Engineering is increased reliability of the subject software, the disciplines have very different approaches to achieving this goal. The software engineer is primarily concerned with creating, transforming and validating descriptions of a system. The software manager, in contrast, is primarily concerned with the management of those descriptions so as to make them available to other parties. Furthermore, the software engineer should be concerned with the elimination of semantic errors (e.g., bugs), whereas the software manager tries to ensure that the system, when installed remotely, is semantically equivalent to the system as delivered by the supplier.

Achieving software reliability through Software Engineering is largely a matter of the application of techniques to facilitate the creation, modification, transformation and validation of system descriptions. For example, the use of programming conventions, high level languages, and formal design systems all enhance the clarity of the descriptions and might facilitate the use of formal validation techniques.

Achieving software reliability through Software Management is largely a matter of providing consistent representation and control of the software source. The provision of such a representation and the controlling mechanisms should facilitate the rectification of problems and the distribution of the solutions in a timely and well-controlled manner.

The rest of this section outlines some considerations when designing a source representation.

- The representation must permit the easy extraction of a subset of the software for transmission to another location. The extraction mechanism must support the specification of the files to be extracted in a variety of ways, such as version numbers, subset specifications, or a list of changed files since the last appropriate extraction.
- The organization [sic][2] of the source must be done in such a manner as to allow personnel to manage, install and test the source without an extensive understanding of the software's purpose or function.
- The addition and/or removal of source files must be simple. Extensive modification of any secondary files (e.g., system construction files). Ideally, the mere act of adding or deleting a file will cause the software construction tools to do the right thing.
- The procedures to create and install a distribution must be usable by a large number of people. If a software distribution is to succeed it cannot require or demand extensive understanding by the installer. To do so would impose limitations on the number of installations that could be performed by demanding more resources than are either merited or available.
- Solutions to distribution or installation problems must be applied across the entire source database in a consistent and easily-stated manner. For example, the mechanisms used to specify installation controls (e.g., installed file attribute specification), environmental variations (e.g., file naming mechanisms), search paths, versions of compilers to be used, include paths, and library paths must be expressed in a way that they can be easily modified by the user. Any significant change in the values of those controls has to force the re-creation and installation of any affected files. Achieving this objective is important since minimizing the mechanisms used to specify such controls will increase the flexibility of the installation and facilitate the adaptation of the distribution to different requirements (e.g., different compilers, operating systems, or machines).

3. General Principles

The principles described in this paper are guided by two considerations. On the one hand, the installation process should proceed flawlessly. On the other hand, there is Murphy's law. Despite all efforts to create a fail-safe distribution, installation will take place in an alien environment, which makes it likely to fail in new and different ways.

[2] Since both authors are now being paid in American currency, some concessions have been made.

"Keep It Simple, Stupid" (a.k.a. the KISS principle). Another way of stating this is "Don't do what you don't understand." The area of Software Management is new and full of hard problems. The goal is to increase the reliability of software, especially at a remote site, not to solve all the problems in the field. Clients will not appreciate research being done on their systems. Therefore, the distribution system should be based on proven, well understood tools. Furthermore, if the software management system is too complex to understand, it will not be used successfully. On the one hand, to try to keep things simple, specialized tools should be avoided. Including such tools in the distribution can be very costly. On the other hand, a specialized tool will often greatly simplify a task by centralizing all of that task's complexity in one place. A good tool must fit well into the distribution as a whole and place minimal demands on the software manager. Attempts to solve the software distribution by writing more complex versions of *make*(1) are distressing in that they require very complex human generated Makefiles. In effect, they address the wrong problem, thereby adding many complex Makefile throughout the distribution.

"If it won't take off in Poughkeepsie, don't expect it to land in East Podunk!" The first clue that a mechanism is too complex is that it does not work reliably in the development environment. If it is the least bit difficult for the software management staff to use a tool, it will be a failure in the installation environment. Makefiles which seem to employ magic are an excellent example of this phenomena. Many people struggle to make a Makefile work without realizing how fragile it is. Such "solutions" will only cause problems when they leave the development organization.

"Too many cooks spoil the broth." If a Software Management system requires more than one "cook", it is unlikely to achieve some of the goals discussed earlier. It must be designed and implemented to be completely managable by a single person. This is a key point that will be discussed at length in the next section.

4. Roles and Responsibilities

Any Software Management system must accomodate three distinct responsibility classifications: the software gatekeeper, the software supplier, and the client site installer (i.e., the client). There are other types of personnel associated with the distribution (e.g., the end-user is conspicuously absent from the above list). However, at this time, we are, concerned with the above three classes only.

4.1. The Gatekeeper

One of the guidelines established in the previous section was that a single person be able to manage of the software. To ensure consistency, to facilitate communication and to centralize efforts, there must be one well identified individual who is assigned the authority and responsibility to manage and protect the software database. This person is called the **gatekeeper**.

The gatekeeper stands between the supplier and client, protecting both parties' interests. When examined with respect to the goal of increased software reliability, the gatekeeper's primary responsibility is to be the best source of the best information about the distribution's status, availability and compatibility. Hence her primary responsibility is to maintain and protect the integrity of the software database, and to ensure that the information required to install, use and maintain the subject software is available, accurate and (we hope) useful.

The gatekeeper need not be a programmer. Her responsibility is the management of source, not the creation of it. Understanding its functionality might be useful but is not necessary. She should have a technical and administrative support group to aid in the preparation and validation of the database and to process the details of exporting and monitoring distributions to remote clients. She will also require the support of an "editorial board" which will act as the "arbitrators of taste." This board is a group of senior technical advisors and management who can advise the gatekeeper on the suitability or necessity for inclusion of a software package in the distribution.

The gatekeeper's responsibilities do not include component testing. She should expect and demand that software submitted for publication has been tested to an acceptable standard by the software supplier or some other party on their behalf. The gatekeeper will test the system, but

only with respect to its construction and installation within the context of the entire distribution. Ideally, the gatekeeper should have access to a secondary groups to do quality assurance testing, but this is not essential.

In order to fulfill her responsibility, the gatekeeper will have to elicit the cooperation of both the supplier and, eventually, the client, as covered in the following sections.

4.2. The Software Supplier

For the most part, it is assumed that the supplier is accessible to the gatekeeper, in that the latter may call upon the supplier to provide extra information and help where the delivered source is insufficient. In such cases it is also assumed that the software supplier will have sufficient interest in the proper distribution of their software to incorporate the changes required to make the it conform to the structure and style of the database. If this is not the case, the gatekeeper may choose to reject software or to acquire alternative assistance to reshape the software as she requires. The point is the supplier must accept the following realities:

- their role is deliver software in its complete source form to the gatekeeper along with the additional construction information in acceptable or an agreed upon manner ("read the Makefile" is not acceptable);
- once delivered to the gatekeeper, the gatekeeper is in control of that version of the software;
- the gatekeeper can expect (and sometimes demand) the supplier to be prepared to rectify problems in previously released versions (up to an acceptable level) so as to be able to provide minor fixes to the client where a full upgrade is either impossible or unacceptable to the client;
- the supplier should attempt to inform the gatekeeper of any substantial activity that might affect the delivery of future releases of the software.

It is difficult to imagine a programmer who would reject such conditions, but it is rumoured [sic][3] that some of the breed have been known to be uncooperative on rare occasions.

The gatekeeper, in turn, will ensure that the supplier's software is protected, freely readable (though not writable) by the supplier, and that any changes required to install the source at remote locations or any software failures are reported to the supplier in a meaningful manner ("You got it wrong, bimbo!"). Furthermore the gatekeeper will attempt to provide a service whereby redundant bug reportings and/or queries are handled by an appropriate support group so as to not interfere with the supplier's work unnecessarily.

4.3. The Remote Site Support

Normally, little can be demanded of the site that receives a distribution. Therefore we must examine the gatekeeper's relationship with such a site. A remote site should be asked to identify a person responsible for the remote location management of the software. Hence, the gatekeeper should attempt to foster and promote a cooperative relationship with this person. This can be done by providing easy to implement mechanisms for the reporting of problems (e.g., 4.3bsd *sendbug*(1)), a hot-line and, above all, meaningful and helpful responses to queries and problems. Effort should be made to make such reporting worthwhile to the remote site as the return will be significant.

As is the case with many aspects of this paper, the above would appear to be obvious, but unfortunately there are far too many examples to indicate that these homilies are often ignored.

5. The Database

The authors believe that formal and rigourous databases should be used to manage all the information about a software system across all phases of the software's life cycle (i.e., from conception to retirement). Such a database and the integrated tool set (e.g., compilers that extract source

[3] There is only so far we will go.

from that database) would provide much better control for both the Software Engineer and Manager.

But, at this time, there are number of impediments to the use of such databases. It is currently impossible for us to employ such a database without violating some of the guidelines discussed in the previous section. The authors feel that we are still a long way from understanding how such a database would be constructed. There are concerns as to how it would be used without imposing severe constraints on the programming staff.

This section examines the D-tree distribution information and its structuring. It is offered to illustrate the range of the required information and its uses, rather than as an example of the way it should be done. Attention is paid to the rational for the particular representation.

The information is split into the following classifications, although the division is at times arbitrary.

- The Source – the software as a body of text and data prior to transformation to the installed form.
- Boot strapping information and site parameters – information used to establish the site and system dependent information and to initialize the tools and environment required to do the installation;
- Construction Controls – the information used to control the creation and installation of the software;
- Distribution Management Information – data used to control, prepare, distribute and monitor the software with respect to remote sites;
- Installation Documentation – details and guidelines to aid the installer of the software;
- User Documentation – information regarding the use and purpose of the subject software.

5.1. The Source

In the large perspective, source can be considered as all the information that is required to distribute, control, install and use software. For the purpose of this discussion, "source" only refers to that information that is directly transformed into an installable component (e.g., the expression of a program in the C language or a data file).

The source files are stored in a directory tree. The source directory architecture strategy is difficult to describe. It is highly dependent on the relationships between components and the mechanisms used to convert the source into the installable form. However, some points must be emphasized:

- Naming conventions must be adopted that facilitates the identification of a file's type and function and the mapping of the installed file back to its sources. For example, the file that contains the "main" entry point has the same base name as its installed form (e.g., "main.c" is to be avoided).
- The directory tree should be designed to facilitate the creation of rational sub-distributions and the expression of dependencies using directory names alone.
- The directory dependency graph must be acyclic.
- Directories should be monochromatic except where it leads to severe fragmentation. By a monochromatic directory we mean that **ALL** the files of the directory are processed in the same manner.

To store past versions of the source file, there should be a versioning mechanism (e.g., SCCS or RCS). Such a mechanism must be used, in a consistent manner, for all source files that can be modified. There must be a deterministic way to map the name of any source file to the file used to administer its past versions and vice versa. It must be possible to fully install the system in the absence of the versioning mechanism.

5.2. Boot strapping and site parameters

The site parameters are those system and location dependent settings that must be specified at installation time by the client. As much as is possible, such information is isolated and centralized in a well identified directory (e.g., in the D-tree it is called *magic*). Site parameters, are specified in two configuration files (one for the machine and one for the site) that contain variable value pairs. In addition to the configuration files, the *magic* directory contains all those files that need to contain a site parameter. The first step of the installation is to copy such files to a new file, replacing the variables in the input file by the value specified in the configuration files. The directory also contains command files to install such files in the required location. In addition to the site parameters, *magic* contains the commands to boot the tools that are required to build and install the rest of the system. The *magic* directory is the most difficult to create as it must be carefully designed to present as few problems as possible. It is used before the site parameters are in place and the distribution tools are usable.

5.3. Construction Controls

By construction information, we mean the names of target directories, header and library search paths, symbolic mappings for libraries, specifications for construction tools (e.g., $CC), options, directory dependency lists, and construction specifications required to create and install the system.

In the D-tree, two conventions are used to specify such information.

Files called **LclVars** contain simple variable value pairs to specify information that applies across a directory tree. All such named files from the root down to the current directory are processed to retrieve the settings. The more local settings override the settings closer to the root. A mechanism to select or suppress settings within the LclVars files, based on the system and options list, is provided.

The actual construction processes to be applied within a specific directory is specified via a *pmak* control file, usually called **PMC**. This file normally contains a line that specifies a shell command to be interpreted to create the full construction description file (the pmak script). The PMC file may also contain lines to specify exceptions and local overrides. The semantics of PMC files are described at length in [Tilbrook 86].

The directory dependency graph (used to control multi-directory constructions and to build subset distributions) is contained in a special form of PMC file. The list of libraries used by a program is specified (symbolically) within the main source file for the program. The expression of all other dependencies are created by processing the source or special rules files provided as part of the pmak system.

5.4. Distribution Management Information

The distribution management information may be divided into five classifications: the distribution file list (a list of all the files in a source form distribution), a validation database used to check the latter for the appearance or loss of files, a set of snapshots for remote installations (i.e., the list of files and their respective version numbers sent to a remote location), the change audit trail with checkpoints indicating releases to remote sites, and the version number database. The use and form of most of these files should be relatively obvious. It must be emphasized that the creation and protection of above information is a crucial gatekeeper responsibility.

5.5. Installation Documentation

This class of information is provided as an aid to the remote site installer and system administrator. Each source directory should contain a file called **Read_me**. These files contain information pertaining to the source files contained within the directory. For example, any special instructions or warnings are provided via this file. It should also provide a list of all the contained files plus brief descriptions, where the actual use is not obvious. Read_me files should be brief but complete as installers invariably do not read them as carefully as one might wish. The Read_me files

are considered part of the source .

Three other forms of information are provided for update tapes and these are: the list of removals, the delta comments, and the update commentary. The list of removals names all the files that are to be removed from the previous distribution (renaming files is done by removing the old file and creating a new one). This information is automatically included by the update tape generating procedure. The delta comments are also automatically included in any update tape. These comments are extracted from the SCCS administration files for all "delta"s included in the update. The update commentary is normally manually produced by the distribution manager and should treated as a new source file that is delivered to the remote site.

5.6. User Documentation

Traditional UPM style manual sections (in -man format) are provided for every installed tool as part of the source tree. These manual sections are kept in either a sibling or child directory (called **man**) of the directory that contains the main component or in the directory itself. Manual section files are always suffixed by a single digit, as in *manual.1*. They should never use the suffix '1' (a common habit in net.rubbish) as that as that suffix is reserved for *lex*(1) source files.

6. Version Numbers

Version numbers are very important when distributing software to a remote site. They are the key link from the programs at the remote site to the software manager's database. As with any other database key, version numbers must be expressed in a consistent form. Version numbers must also be available in any situation where a user will be communicating with the distribution organization. Among these instances are bug reports and general questions.

In order to ensure the universal existence and consistency of version numbers, they are generated automatically by a specialized tool as part of the construction process. To meet the requirement of accesibility, version numbers are compiled into each program's binary. The version number string has the following components:

V@(#)Name 1.2.3(4) 17 Aug 87 05:06:07

V@(#): string used to find version string in binaries and/or core files using what(1).

Name: capitalized name of the program

1.2.3(4): Edition, Revision, Level and compilation numbers collectively referred to as the Version number (see below).

17 Aug 87 05:06:07: The date and time the version string was created (usually immediately prior to compilation).

The Edition number (1.2.3(4)) corresponds to the edition number of the hard copy Andrew guide and will be changed by the documentation group when a new edition of the manual is to be released.

The Revision number (1.2.3(4)) is used to indicate major revisions or modifications to the system within the edition. It is may be incremented by the creator once the documentation is ready. It reset to 1 whenever the Edition number is incremented.

The Level number (1.2.3(4)) is used to indicate revisions which are discernible by the user but do not merit or require an upgrade of the documentation. The programmer may increment the level number at his/her own discretion. The Level number is reset to 1 whenever the Edition or Revision numbers are changed.

The compilation number (1.2.3(4)) is incremented automatically every time the program is compiled. It is reset to 1 whenever a higher level number is changed.

Each program has one of these version number strings compiled into its data section. This way, one can use the /fIwhat/fP(1) program to extract the version number from a binary or a core dump. Although it is not of paramount importance, it is also convenient to be able to see the version number while the program is running. It should be remembered though that the version

number is for use by the programmer and software manager. It should not clutter up the user interface of a program or get in the way of the user.

7. Preparing a Distribution

Much of the work that goes into preparing a distribution can be characterized by the phrase "rigorous application of common sense." The methodologies outlined here are not magical, or even necessarily original. Unfortunately, it is not clear that these methods are employed for most distributions. Therefore we outline the following four steps to preparing a system for distribution:

- Deciding which software into the distribution.
- Identifying the source for the chosen software.
- Laying out the distribution database.
- Establishing conventions for the distribution as a whole.

The first step in organizing a collection of software into a distribution is to figure out what is to be included. A good distribution should be easily manageable and place minimal "stresses" on the environment into which it is installed. It should must make minimal assumptions about the target environment. For these reasons, one must consider very carefully what goes into the final product. Programs should be evaluated in terms of their dependencies (costs) versus their usefulness (benefits). If possible, seldom used programs should be eliminated since they are not worth their dependencies. Keeping track of many small programs can easily make the task of preparing a distribution unmanageable. It is often easier to add pieces to the distribution after the basic structure is in place and the necessary components are provided.

After the dead wood has been culled from the system, the next step is to identify the sources for the chosen components. In sizable development efforts, it is easy to lose track of where all the sources are. Although the task of finding the sources for a program may seem trivial, it often isn't. For example, the Andrew system has been developed on a distributed filesystem with over fifteen gigabytes of storage. Within this system, there are six separate operating system distributions for four machine types, and two complete sets of Andrew software (one for the campus released software and one for the development versions). It would be for all practical purposes impossible to match the released software to source. Even within the development environment, we find things like twentyfive versions of the *install*(1) program. The process of identifying the source for programs will involve lots of tree combing and such. This is largely the gatekeeper's responsibility, but will require good communication with the software supplier.

Now that there is some idea of what constitutes the distribution, a database in which to put the source must be designed. The representation of this database will be a filesystem tree. Its basic structure is dictated by the dependency graph of the software. The very first things to go in will be the distribution tools, which are fairly much the same from one distribution to the next. The rest of the tree will depend on the interrelationships of the many pieces of software in the distribution. Fortunately, usually there is already a running system to start from, so the process of organizing the tree can be lazy evaluated to some extent. It is not necessary at first to have all of the dependencies in the tree. It is permissible to reach out into the existing system to grab components. This gradual conversion process should make things easier for both the gatekeeper and the supplier. The direction should not stray from gaining a disciplined approach though. One should ensure that software remains buildable during every step of the copying process. The best way to do this is to frequently rebuild the system. Is important that development not be going full blast while this is happening. This process is complete when the entire system is organized into a source tree.

The final step is to do a global cleanup of the software system. Since the entire system is now in a "semi-organized" tree, it is possible to take a global view of the software. This is a good time to establish certain conventions across the entire distribution. The gatekeeper should identify site parameters (i.e. pathnames where data files are found, optional configurations) and coordinate with the software supplier to establish a minimal set. All such parameters should be collected in a single directory to make it easy for the remote site to configure the system. Conventions for

portability should be established. The gatekeeper may have to suggest alternate coding conventions for use within the development environment. A trivial example of this is the use of *strchr*(3) under System V vs. *index*(3) under 4.3bsd. The distribution may provide a library which has a *strchr* function for portability. The software supplier should be encouraged to use the portability library and *strchr* instead of *index*.

After the distribution has been put together, it is very important to test the construction process. It should be possible to build the entire system at a site which has never received any software from the supplier. Having a friendly beta test site where this can be done is invaluable. If possible, the distribution should be tested on all flavors of machines and operating systems it is intended to run on. Any dependencies of the software that are assumed should be carefully enumerated so that they can be announced as prerequisites for the distribution.

Usually, a distribution is prepared from an existing system. The above outlines a method of getting from the chaos of a development system to the "order" of a distributable system. Having put all this effort into creating order, it is a good goal to keep it that way. The gatekeeper will do well to try and reimpose some of these conventions on the software supplier. Hopefully this can be done in an undisturbing way. The software supplier's interest in a disciplined software approach is to increase the quality of their software. An organized source tree makes it much easier to track bugs and recognize when they are fixed. The requirement that any change to the software must be explained to the gatekeeper before it goes into the distribution will help encourage good record keeping. Historical records of the software's evolution are very useful for gaining an understanding of how the system works.

8. The Tools

This section does not describe specific tools. Rather it discusses changes to the D-tree tools since the last paper, some approaches that have proven to be essential to manage thousands of files, and the source versioning system choice (e.g., SCCS or RCS).

8.1. The D-tree Pmak tools

[Tilbrook 86] paid extensive attention to the tools used to support the construction and installation of software at a remote site. These issues will not be discussed in this paper, other than to state that eighteen months later the basic reasoning has remained the same: there is the need for a better approach to the system construction process.

In fact, some aspects of the approach have been considerably strengthened. The primary goal of the use of *pmak*(1) and its script generators to control construction can now be justified by stating that its use has permitted extensive simplification and/or centralization of the information required to control the construction process. This has been accomplished primarily by creating a mechanism for expressing various construction controls (e.g., include and library search paths, library mappings, target path names) in a simple generalized syntax. At the ITC, new script generators have been created or are under construction to deal with some of the new requirements imposed by the Andrew system and its components.

There has been considerable work on the dependency problems. Since one of the objectives established earlier in this paper was the simple update of a remote distribution, the mechanisms for ensuring that a generated script is accurate and up to date have been substantially improved. *pmak* will soon be able to infer dependencies even within a dynamically changing construction environment.

One of the objectives stated in the 1986 paper was to use whatever version of *make*(1) was provided by the host system. However, we have now reached the decision that this policy is no longer viable and imposes too many restrictions, thus future distributions will provide their own alternative to *make*. The major reason for this departure is that few *make*s can adequately or easily handle what is an absolute necessity at the ITC, the creation of objects for multiple machines from a single copy of the source. This requirement, coupled with the desire to eliminate the gymnastics required to cope with the variations among the various implementations of *make*

have forced us to design and develop yet another variation. But, it must be pointed out that this version adheres very strongly to the K.I.S.S. principle, and includes some of the key features of *mk* [Hume 87]. Also the reader should note that because this version will always be front-ended by *pmak*, it can be very simple minded (namely, compatibility is not a concern).

8.2. E2BIG

Another area in which the tool set has to be considerably enhanced when dealing with a distribution is to cope with the frequent occurrence of the message:

> Arguments too long.

(i.e., errno = = E2BIG).

Far too many of the tools cannot cope with the file lists of even a medium sized distribution. Many of the tools (e.g., *ls*, *bm*, *grep*) have to be modified to process an input list of files as opposed to an argument list specified on the command line. Alternatively one may have to create new tools with enhanced functionality for efficiently dealing with large files lists.

8.3. The File List Creation System

In the section on the database, the major file lists were explained. One of the gatekeeper's major responsibilities is to ensure the accuracy of these lists, hence a set of tools are required. Rather than to attempt to explain all the processing that is required, the following is a list of all the file lists that are produced during the process used to maintain the D-tree database:

,a.all:	all files in the subject directory
,a.dirs:	all the directories in the subject directory
,c.cfiles:	all the p-files with SCCS/p. removed
,c.gfiles:	all the s-files with SCCS/s. removed
,d.gfiles:	existing g-files
,f.+files:	all +files or +directory files (temporary file)
,f.commas:	all comma files or comma directory files (temporary file)
,f.else:	all files not identified by flsplit
,f.pfiles:	existing SCCS p-files
,f.sfiles:	existing SCCS s-files
,f.tmps:	temporary files identifiable by name
,m.pnos:	p-files with no s-file (s-file might have been lost)
,m.regs:	registered distribution files that don't exist (might have been lost)
,m.sfiles:	registered s-files that don't exist (might have been lost)
,m.snog:	missing g-files (s-file exists but g-file doesn't)
,m.tmps:	registered temporary files that don't exist
,n.files:	unidentified (new) files
,n.sfiles:	unknown (new) s-files

Normally any empty ",[nm].*" file will be removed. If any such files are not empty, they contain the names of files that will have to be processed to bring the major file lists into sync with the existing database.

8.4. SCCS vs. RCS

The current D-tree system uses and depends on the SCCS system. For the most part this is due to historic considerations in that it has been managed that way since its life on PWB (circa 1978), which was the first system to include these tools. However, due to the unavailability of the SCCS system on bsd systems, many people at CMU use RCS, thus we have been forced to evaluate its possible use for large scale distributions. The choice is not an easy one; choosing the lesser of two evils never is. Both have advantages and disadvantages, as outlined below.

- RCS's interface is slightly more attractive than raw SCCS, However, Eric Allman's SCCS interface is better than that offered by RCS and offers other advantages.

- RCS's ability to include the history in a source file would appear to be very useful for the distribution of small software systems but becomes costly when considering thousands of files. SCCS can be modified to provide a similar mechanism, but it is cumbersome to use.
- SCCS is buggy in some of its error condition handling and needs some fixes to deal with non-vaxen (unbelievable as this may seem) due to byte order dependent code.
- SCCS's keyword scheme makes it undesirable to ship sources with expanded keywords to external software developers, whereas RCS can deal with expanded keywords.
- RCS does not check the return values of all its writes for error codes thereby endangering the V-file. SCCS checks the return code of every write by redefining *write*(2) thus guaranteeing user notification on write failure. Unfortunately neither system checks the uses of *close*(2), which on the ITC VICE file system can cause unreported failures.
- RCS does not have any check sum in the V-file thus there is no mechanism to check its integrity. Every SCCS s-file contains a check-sum which is checked on every access.
- RCS writes the locking information into the V-file whenever the file is checked out with a lock, which, when combined with the last two points make the V-files very vulnerable. Even worse, from the software manager's viewpoint, the fact that the file is locked is not obvious (i.e. it requires parsing the V-file to tell if a file is locked). SCCS creates or modifies a supplementary file (a p-file) which avoids endangering the s-file and makes the fact that a file is "checked out" conspicuous.
- RCS's scheme for version management is inferior to that of SCCS for a variety of reasons. It is based on line numbers which are somewhat unreliable in the face of failure. RCS needs to do multiple editing passes of the file to build old versions, hence it is much slower. Finally, the RCS administrated source (i.e. ,v file) is not expressed in a form that can be viewed directly to determine when or how a change took place, whereas that of an SCCS file can.
- RCS provides a binding scheme across multiple files, whereas SCCS does not.
- Neither system has a scheme to to deal with file removals or renaming.
- RCS always appears to have been extensively hacked, a good clue something is wrong, whereas SCCS is amazingly stable.
- RCS offers a merge facility. SCCS does not.

For the gatekeeper, the better choice would appear to be SCCS, partly because of the greater reliability and partly because of the use of a p-file instead of a line in the v-file to indicate work in progress. On small source systems this may not seem important, but when dealing with large systems, the ability to determine what is being edited using *find*(1) is very important. The advantages offered by RCS (e.g., embedded history, proper management of expanded keywords, and releases) can all be dealt with in a satisfactory manner using supplementary tools.

At the ITC, the suppliers will probably continue to use RCS for their personal development, however, the gatekeeper will use SCCS for the above reasons and since a mature set of well integrated distribution maintenance and creation tools exist for SCCS and not for RCS.

9. Conclusions

There is a limit to the effort anyone will spend installing or upgrading a piece of software. Overstepping this limit results in user impatience and frustration, which ends up in the software not being evaluated on its own merits. Software management techniques, such as those described in this paper, will substantially reduce the effort involved in installation and upgrades. Hence, these techniques are necessary to the success of any sizable project.

There are two ways in which software developed within a university research environment can be valuable. The first is through the traditional method of publication of papers in technical journals or presentation at conferences. The second is through "publication" of the software itself, that is making the software available in an installable form. This allows the objective evaluation of the concepts behind the project through its use. However, the costs (both in time and people)

ıvolved in creating a proper distribution are often too high. A coherent Software Management system will reduce these costs and avoid the pitfalls of installation that prevent proper evaluation of the delivered software.

The techniques described in this paper have been used effectively elsewhere. However, we are just beginning to use them at the ITC. It has been an interesting experience watching the transformation of programmer attitudes towards Software Management. The ITC software staff consists of many high quality, conscientious, and by necessity opinionated, programmers. Initially, their perception of Software Management was negative in that they felt threatened and questioned its value. After three months of exposure to the principles outlined in this paper, they are very enthusiastic, and believe that it will make their job easier. Furthermore, there is a consensus throughout the ITC that Software Management will improve the overall quality of the system and make the large scale distribution of the Andrew system a reality.

Acknowledges

Many thanks to Jennifer Robertson of the ITC for her help and patience.

References

D.M.Tilbrook and P.R.H. Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April, 1986, USENIX Atlanta Conference Proceedings, June 1986.

Andrew Hume, *Mk: a successor to make*, USENIX Phoenix Conference Proceedings, June 1987.