THE SOFTWARE DEVELOPMENT ASSISTANT

Charles C. Hayden, John C. Mitchell, Jishnu Mukerji, and Frederick A. Schmidt

The Software Development Assistant (SDA) is an exploratory software development environment. It employs new hardware and software technology to help developers of the Definity® 75/85 communications system manage the complexities of software design. We describe an integrated set of tools to be used during the design and implementation stages of the development process, and the architectural components on which these tools are based. We then discuss current work to develop new user capabilities and to incorporate new technology. The Software Development Assistant prototype is general enough to be adapted for use by other software development projects with modest effort.

Charles C. Hayden, John C. Mitchell, Jishnu Mukerji, *and* ***Frederick A. Schmidt*** *are associated with AT&T Bell Laboratories in Middletown, New Jersey. Mr. Hayden is a member of technical staff in the Advanced Integrated Systems Department, and is responsible for tools and techniques to enhance the productivity and quality of PBX software development. He joined AT&T in 1980 with a B.A. in physics from Reed College, Portland, Oregon, and an M.S. and Ph.D. in computer science from the University of Southern California, Los Angeles. Mr. Mitchell is supervisor of the Exploratory Systems Group in the Advanced Integrated Systems Department, and manages forward looking work projects related to Definity PBX products and development processes. He holds a B.S. and M.S. in mathematics and statistics from* (continued on page 89)

Introduction

The goal of the Software Development Assistant (SDA) project is to apply advances in computing and software technology to the software development process of the Definity 75/85 systems. Existing large software systems, such as Definity 75/85, continue to be upgraded. The upgrading process can be made more efficient by using newer technology better suited to the difficult task of enhancing a large software system.[1,2]

The SDA project has been focused on the time-consuming, repetitive tasks at the core of the development cycle: *learning*, *designing*, *coding*, and *unit testing*. As enhancements are added with each Definity release, its software base becomes more complex. This makes each release harder to change, harder for new or reassigned staff to learn, and harder to test. Also, much of the rationale[3] for design decisions is not documented, further burdening the process of learning how the system works.

Our approach in SDA for improving this situation follows two paths. One path is to create an integrated set of software development tools, the SDA Toolkit, that would substantially reduce the time these tasks require. The second path is to create a computing environment, the SDA Computing Platform, which provides computing capacity to

support the new environment at a reasonable cost, both in capital cost and administration and operations cost. It is important that SDA provide new capabilities without disrupting the existing development environment. Familiar tools and commands must still be available, but new ones make the job easier. This approach minimizes—and may even eliminate—discontinuity or breakdown in the user's perception of their work environment,[4] and therefore minimizes the retraining costs of moving the existing development staff to SDA.

The goals of the SDA Toolkit were born from problems in the current software development environment that affect developer productivity:

- The current tool set consists of independent tools with dissimilar user interfaces. Data is not easily transferred from one tool to the other.
- Many development tasks still require time consuming and error prone manual steps, such as printing and transcribing information, that break up the work flow.
- Ineffective communications among people and groups is a major cause of rework in large development projects.[8]

Our goals were to provide an integrated set of tools in a uniform environment that supports a maximal number of development tasks, including group communication. To achieve this we exploited the ideas of direct manipulation,[5] scientific visualization,[6] embedded selection, and multiwindow synchronized scrolling[7] within the user interface. We added capabilities that enable the project's design groups to collaborate more efficiently on design and implementation decisions, and to easily record their decisions.[9,10]. Wherever possible, we automated or eliminated clerical tasks that supported the development cycle.

To support these new tools for developers, we designed a low-cost, workstation-based, networked computing configuration that uses commercially proven hardware and operating system components. The existing development environment uses several mainframe UNIX® system machines that are accessed via character-based terminals or window-based AT&T 630 MTG terminals over AT&T Datakit® or ISN data switches. There are inherent limits in this configuration that make it incapable of supporting the new software tool environment we envisioned. Some of the limiting factors are:

- Small display size of user terminals
- Slow data communications speed (e.g., 9600 baud).
- Limited and slow graphical display capability
- Contention for computing resources that is common to timesharing systems.

The SDA computing platform consists of high speed networked diskless workstations, large resource servers, and terminals that support the X Window System™ software. (*X Window System* is a trademark of MIT.) Many elements of the existing environment were kept intact to reduce conversion costs and minimize disruption of ongoing development work. The combined SDA hardware and software system allows development staff to ease into the new environment using concepts similar to the previous environment.

Panel 1. Terms and Acronyms in This Paper

ASCII	American Standard Code for Information Exchange
`cia`	C Information Abstractor
CPU	central processing unit
CRT	cathode-ray tube
FDDI	Fiber Distributed-Data Interface
`gadd`	Graphical Animated Design and Debugging (graphical simulator prototype)
`gdb`	GNU debugger
ISN	Information Systems Network (AT&T)
MTG	Multitasking Terminal with Graphics
PBX	private-branch exchange
RPC	Remote Procedure Call (Sun Microsystems)
SDA	Software Development Assistant
NFS	Network File System (Sun Microsystems)
NIS	Network Information Services (Sun Microsystems)
XDR	External Data Representation

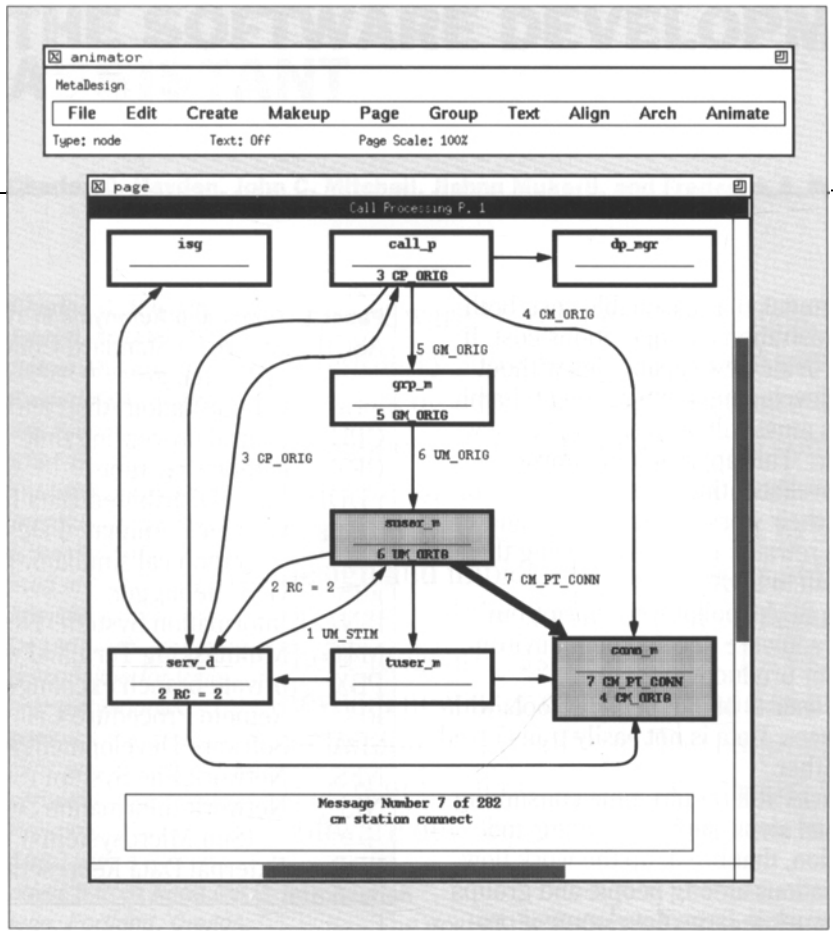

Figure 1. Design animator display showing interprocess communication. The bold arrow and shaded boxes represent the "current message" being viewed.

Our method for developing SDA followed the rapid prototyping paradigm. That is, we created prototypes quickly to do tasks in ways we felt were demonstrably better than current practice. We then showed these prototypes to developers to get feedback on important features and capabilities. This approach was essential to quickly deliver useful capabilities. We believe that SDA constitutes a new approach to large software systems development efforts in two ways. One is in the high degree of integration of the essential development tools needed for software development. The second is in the design of a networked computing environment that is able to support a large software development team. Though SDA's target was the Definity 75/85 PBX system project, we were careful to create an improved software development environment that could be adapted for use by other software projects.

The SDA Toolkit

The SDA Toolkit comprises a collection of tools that may be used in either of two ways:

- *Individually*, for highly focused activities such as learning the high-level software architecture or debugging a subroutine.
- *Concurrently*, as an integrated set for tasks that span many levels of software design, e.g., debugging a feature that is implemented across several communicating processes.

The following subsections describe the current prototype toolkit.

The Design Animator. The *design animator* graphically animates interprocess communication during software execution (see Figure 1). Users can capture, replay, edit, save, and print the animations. A diagram-oriented, interactive, graphics interface enhances a

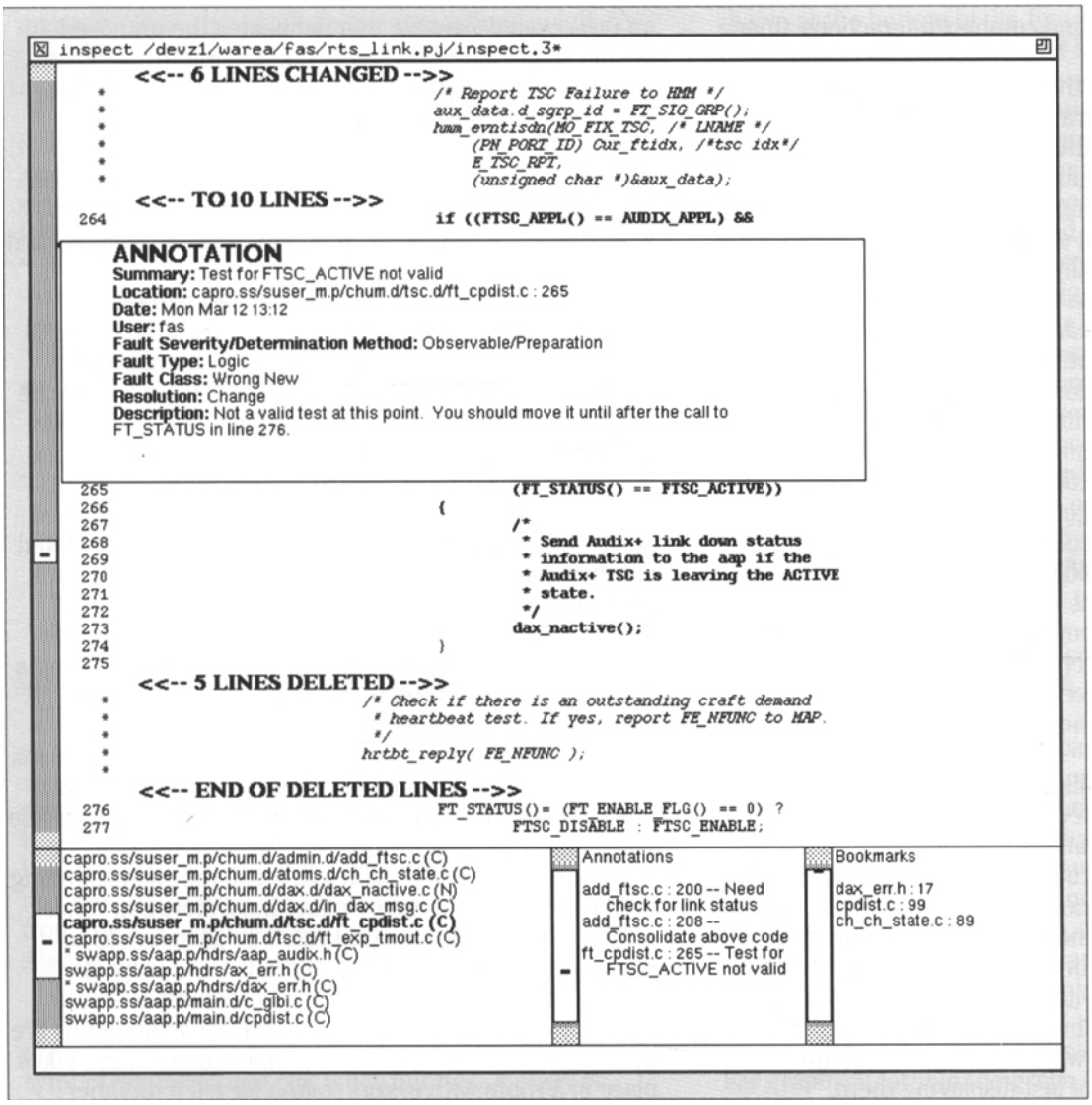

Figure 2. Annotator window display showing the inspection document (top) containing an annotation. At the bottom of the window are the file (left), annotation (center), and bookmark (right) navigation panels.

user's ability to understand complex interactions among system components.

Users describe their visualization of the system by drawing a directed graph that shows the processes and communications paths of interest. The animator then animates the graph, displaying in sequence only those messages occurring along the prescribed paths. As an option, the animator can show all messages transmitted in the system, dynamically creating processes and paths on the diagram as necessary. This mode is particularly useful when users are learning the behavior of a system and are unfamiliar with its architecture.

With the animator connected to the debugging port of an operating Definity 75/85 switch, developers view interprocess communications as they occur. By interacting with the switch and studying the resulting animation of the messages that are sent, the developers learn about how a particular switch feature is implemented. To enhance readability, the animator automatically translates into message names the hexadecimal codes output by the debugging port. This method of learning the software architecture would be impractical using the current practice of setting breakpoints at source code level, and then manually translating the textual debugger output.

The animator fills several needs in the software development process. Because it shows exactly what interprocess communications take place, and is guaranteed to be accurate, it provides access to a dynamic architectural document that never becomes outdated, and that can illustrate any combination of feature interactions. We have also generated a library of animation scripts for the major features of the switch. For educational purposes, these can be viewed offline from the switch. An enhanced version of the animator will be used to debug interprocess interfaces instead of just displaying them.

The Annotator. The *annotator* is a tool used in the software inspection process. Before software changes are introduced to the project base, they are inspected by a special team. Each team member individually reviews an *inspection document,* that delineates the proposed changes, and notes any faults that are found. The team then meets and agrees on a common list of faults that must be corrected before the changes are allowed.

Team members use the annotator before and during the inspection meeting to view and comment on an electronic version of the inspection document. The workstation display of the annotator window is illustrated in Figure 2.

The display's upper pane shows part of an inspection document. Differences between the project's software base and the new software are highlighted through fonts. Several mouse-based commands help the user move to the next or previous page, file, or change. The user may also insert bookmarks—temporary place holders to which he or she can return at any time. An annotation, such as the one illustrated, may be inserted at any point. Most fields in the annotation are completed automatically, or are optional. Ordinarily, the user completes only the summary and a brief description. This makes it easy to make short notes in a document. The description field may be expanded as necessary. Using a "cut-and-paste" feature, users may easily copy code excerpts into an annotation.

The bottom panes list the files (left), annotations (center), and bookmarks (right) in the inspection document. The file name in bold font contains the cursor. The user clicks the "mouse" on a file name, annotation, or bookmark inside these panes to scroll the corresponding item into view in the upper pane. Inspectors can "check off" files in the left pane (indicated by an asterisk) to keep track of what parts have already been inspected.

After each team member comments on his or her copy of the inspection document, the annotations are merged into a single document. The team meeting takes place in a room with a workstation for each member (though some or all participants could be at another location). The team members use the annotator in conference mode to view the merged document. The person documenting the team's decisions during the meeting

may add, delete, and change the annotations in full view of the other participants. The final annotations are reformatted to produce the meeting report.

Using an annotator instead of a paper-based method streamlines the report-making process, and integrates the browsing capabilities described in the next section. The specific benefits of the annotator include the following:

- Because readers tend to "jump around" while scanning an inspection document, both in preparation and in the meeting, the annotation tool makes reviewing much faster.
- The tool provides an easy way to search for changes in lengthy files that have only a few changes. It also has a convenient facility for tracking what has been inspected.
- The tool captures comments as they are first noted. Thus, it eliminates transcription during the meeting and retyping afterwards. The meeting can be further streamlined if individual comments are merged beforehand, and the merged version is made available in advance to team members before the meeting.

Browsers. A *browser* is a tool that allows rapid navigation and querying of the project's software base at various levels of detail. It is analogous to the way a book is browsed, i.e., by looking at its contents, index, and selected pages. Some browsers also allow editing of the displayed "pages." We have developed two software browsers as part of the SDA environment: one with a textual, the other a graphical user interface. Both browsers rely on a database that contains information abstracted from the project's software base by the C Information Abstractor `cia`.[11]

The Text Browser. The *text browser* (Figure 3) allows the user to view or edit the source code in the project's software base that is identified in responses to symbol definition and reference queries. A *query* is formed by selecting the symbol text in an annotator or source window, and selecting a query (e.g., "View Definition") from the text browser menu. The result of the query is displayed in a separate scrollable window shown at the bottom of the figure. The output of any previous query can be redisplayed by selecting its title in the right pane of this window. The source code defining a selected symbol can also be viewed and edited in a separate window. Users need not deal with the underlying directory structure of the project's software base. Since text in any window can be selected as the subject for a query, typing is minimized.

The text browser is designed to work with other SDA tools. Used with the debugger or text editor, it helps a developer decide where to make changes to a program. As Figure 3 illustrates, it is used by a code inspector to determine that a program has been properly changed, and that those changes do not conflict with other areas of the design. When using the design animator, a user can first select a displayed message name, then request to see the source code in the receiving process where that message name is referenced.

The Graph Browser. The current version of the *graph browser* graphically displays the static function call graph for a process (see Figure 4). Each function is displayed at the center of its own page. The functions that call it are shown on the left, and the functions it calls are on the right.

To see the page for a particular function, the user clicks the mouse on the correspondingly labeled arrow-shaped box. The user can use the text browser to open the appropriate source files that define any of the functions shown in the diagrams.

Because the graphic and text browsers access the same abstraction database, other reference relationships could be expressed in graphical form. These relationships include the use of global variables by functions, and the use of header files by C-language files.

Presenting these complex relationships in a familiar graphical form in the graph browser enhances understanding. While navigating through such a diagram, the user sees dependencies and heavily used functions and variables that otherwise would go unnoticed in a purely textual presentation of the same information.

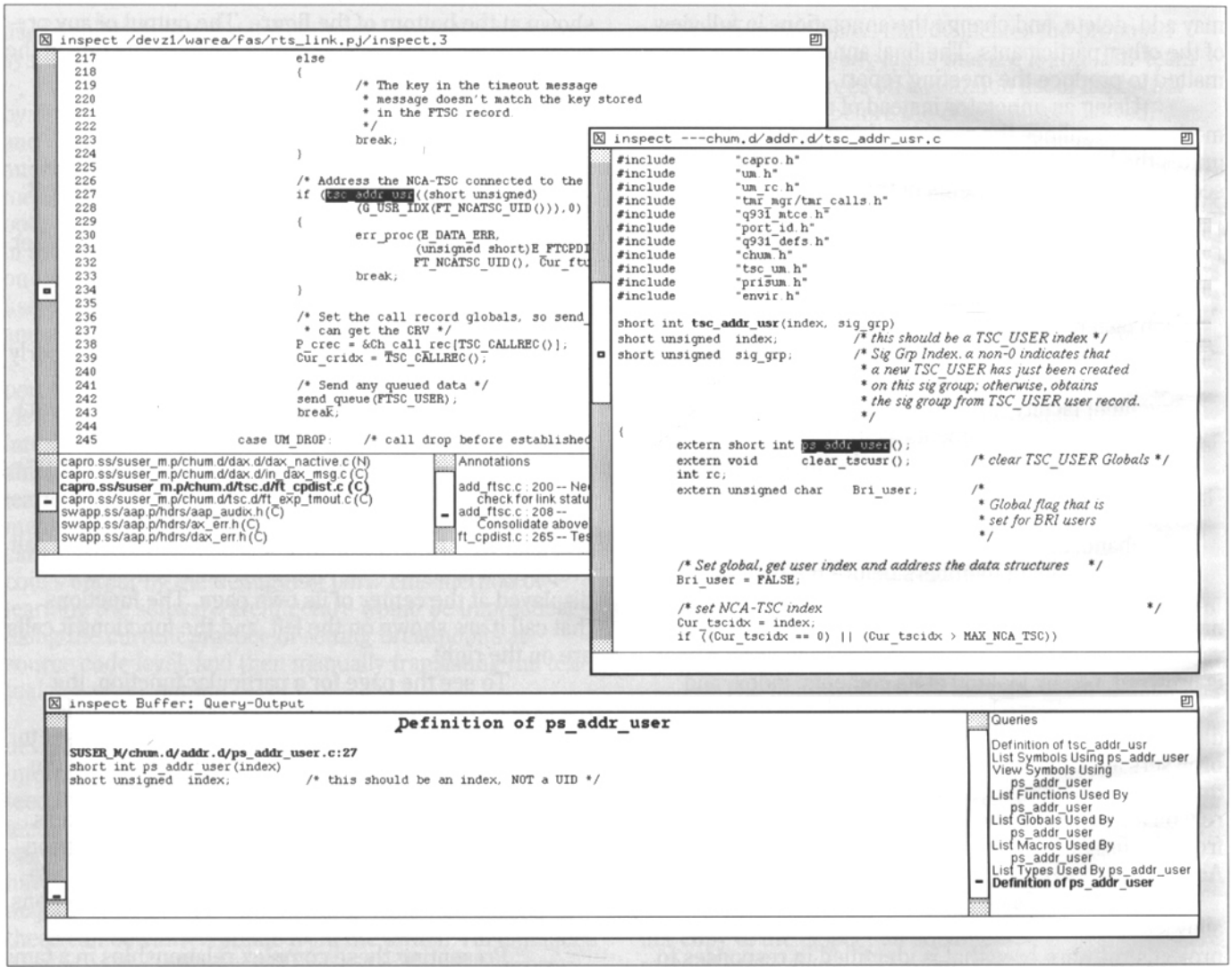

Figure 3. Text browser display showing a function name selected in an annotator window (top background), a source file window opened by the browser to the file defining the function (center right), and the query output window (bottom) displaying the result of a "View Definition" query on a function in the source window. Previous queries are listed in the right pane of this window.

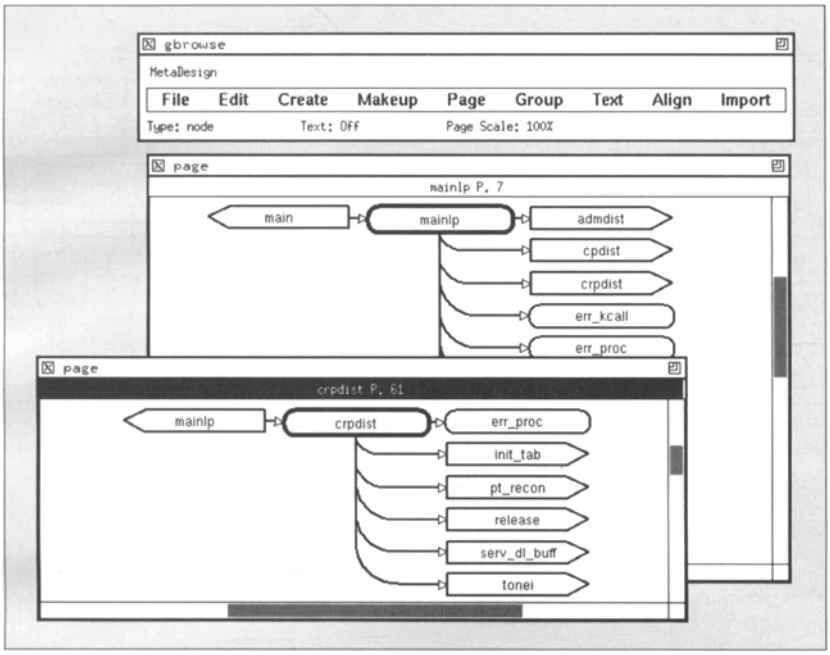

Figure 4. Graph browser display showing the function call graph for a portion of an AT&T System 75 PBX system process. Two pages out of 72 are shown.

Architectural Components

In creating SDA, we have adopted components from projects at AT&T Bell Laboratories, commercial vendors, and universities. Our philosophy has been to seek out and use the best available technology for each required component. The following section describes the hardware components shown in Figure 5, and subsequent sections describe the software components shown in Figure 6.

SDA Computing Platform. The advent of workstations has made available many leading edge tools that are unavailable on mainframes. One of our goals has been to design a computing platform that will give developers of large software projects access to these workstation-based tools, but not deprive them of the advantages of using mainframes for large project development. The SDA computing platform attempts to meet this goal by configuring off-the-shelf software and hardware to create a nearly seamless, integrated environment containing both mainframes and workstations.

This hardware environment consists of the network of diskless workstations, file servers, mainframes and X Window System-compatible terminals depicted in Figure 5. AT&T Starlan 10™ hardware is used to connect diskless workstations in offices that have their primary file servers in a computer center. Starlan 10 hardware is also used for the backbone network that connects the file servers and existing mainframes. In the future, the fiber distributed-data interface (FDDI) will be used in the backbone network. The hardware environment operates using Network File System (NFS), Network Information Services (NIS) and Remote Procedure Call/External Data Representation (RPC/XDR) features of the client/-server software architecture from Sun Microsystems.[12]

Diskless workstations run the user's UNIX system processes, and provide a large-screen, multiwindow

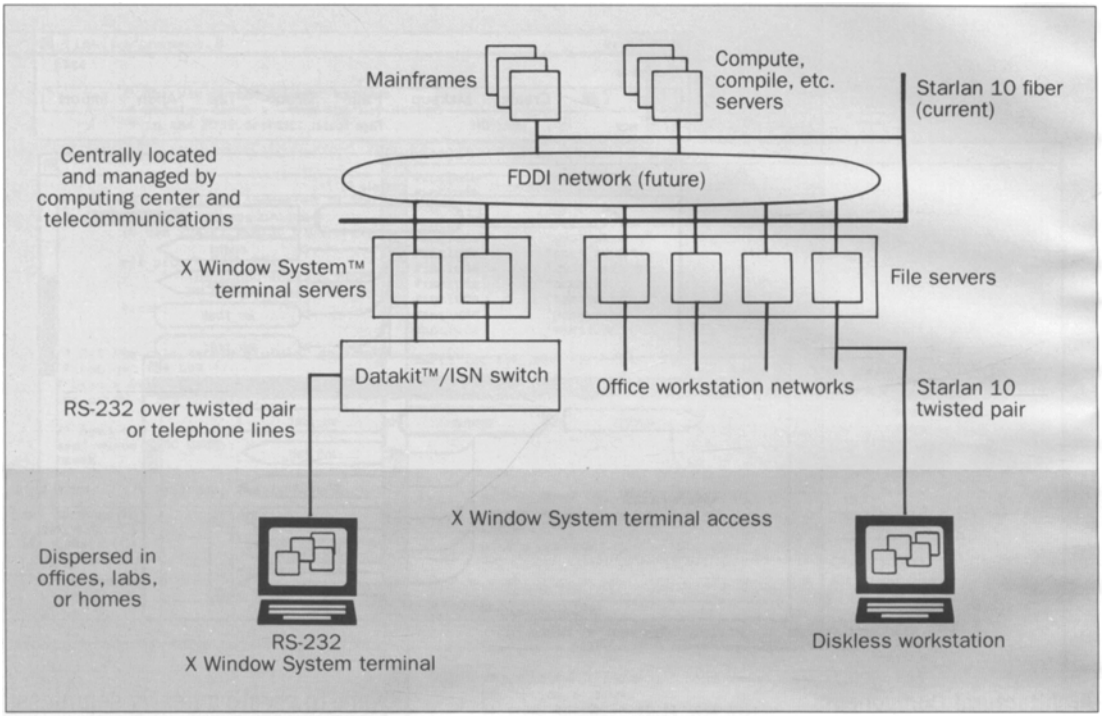

Figure 5. SDA computing platform, showing the distribution of processing and network hardware.

interface to the whole environment.[13] These workstations own and control resources such as the display, keyboard and mouse, and network interface. All other resources and services—e.g., static storage, database management, printing, mail—are accessed over the network from file servers and mainframes.

Diskless workstations offload all user interface processing from the backend mainframes and servers. This leaves the mainframes to execute transaction-oriented tasks for which they are best suited, especially those—e.g., database management, large compilations—that need high throughput to disk. Moreover, because all system administration is localized on the file servers, this design simplifies the operation, administration, and management of the networks.

Any user can log onto any of the terminals (i.e., diskless workstations or X Window System terminals). Users' logins are not bound to individual workstations or servers. When users log in, they are presented—as far as possible—with the same environment, independent of the location or the processor architecture of the workstation. This is important since the developer needs to use the computing platform from several different workstations in different rooms.

The file system appears as a single directory structure, though it is distributed among several file servers and mainframes. Directories containing the project software base on the mainframes are accessible from the SDA environment using the same names as on the mainframes. Resources such as printers or the mail system—i.e., resources common to the mainframe and SDA environment—are also accessed using the same

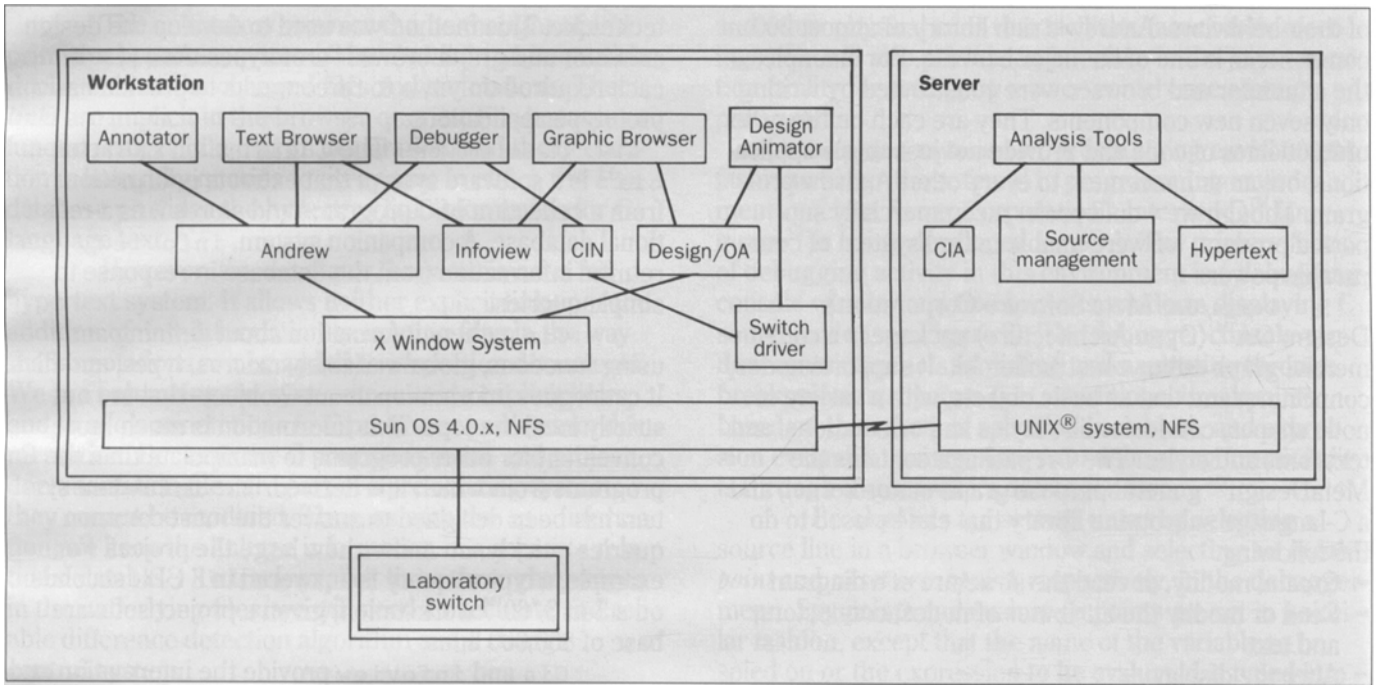

Figure 6. SDA software architecture. The top software layer represents SDA tools. Lower layers are tool construction and operating system software components that support the tools.

names from either environment, and using the same basic interfaces. Besides the basic interface, however, each resource in the SDA environment acquires a more sophisticated, window-based, uniform user interface. This dual interface reduces learning time to move to the SDA environment.

X Window System and Andrew. The X Window System[14] software provides a standard environment for multiple applications to share a workstation screen, keyboard, and mouse. Each application can open one or more screen windows, and can display text or graphics in its windows. The widespread acceptance of this software allows for the use of applications and toolkits from many providers. All SDA components use the X Window System software as their graphics environment.

The Andrew™ system[15] provides an object-oriented applications environment. (Andrew is a trademark of IBM.) Where the X Window System provides only low level input and display drawing functions, Andrew's components range from menus, scroll bars, and selection panels to spreadsheets, graphics editors, and specialized text editors. The Andrew system supports the nesting of arbitrary types of data, and automatically provides for the proper software component to operate on each type. In this way, even existing applications can operate properly on data types introduced later.

As in other object-oriented programming systems, new kinds of components may be created as specializations of existing components. It is easy to modify even objects of great complexity by overriding only a few

of their behaviors. Andrew's rich library of almost 200 components is one of its major benefits. For example, the annotator and browser were constructed by writing only seven new components. They are each on the order of 2,000 lines of code and provide not just single applications, but an enhancement to every other Andrew program. Though we would prefer a commercially supported product, we were unable to find system of comparable power.

Design/OA. Meta Software Corporation's Design/OA™ (Open Architecture) package is a commercial graph editor construction kit. It supports nodes, connectors, and text as basic objects, with a variety of node shapes, connector line styles and orientations, and text fonts and styles. The OA package contains the MetaDesign™ general purpose graph editor kernel, and a C-language subroutine library that can be used to do the following:

- Create, modify, or read the structure of a diagram
- Read or modify the attributes of nodes, connectors, and text
- Alter the display
- Activate any of the MetaDesign menu items.

Call back functions can be specified so an application can intercept user events such as selection of diagram objects or menu items using the mouse. Additional menus can also be added. Applications developed using this package have access to the standard UNIX system environment for accessing files and communicating with other processes.

We have found this package has two important advantages in our prototyping work. First, because the editor environment manages all details of graphics creation, editing, and display management, we could quickly build graphical tools that use this user interface model. Second, since the OA subroutine library presents the same user interface model as the MetaDesign editor, the editor itself can be used—before writing the applications software—to experiment with user interfaces and graphical representations. This is a valuable rapid prototyping technique. This method was used to develop the design animator and graph browser prototypes. As a result, each required only two to three weeks to provide basic prototype applications.

Cia and Infoview. The C Information Extractor `cia`[11] is a software system that extracts information from a collection of C programs and stores it as a relational database. A companion system, `infoview`, returns information from the database in response to simple queries.

`Cia` gathers information about defining and using functions, global variables, macros, types, and files. It catalogues information about C objects that are not strictly local in scope. This information is much more convenient for other programs to manipulate than the C programs from which it is derived. The `infoview` system has been designed to answer the most common queries quickly, no matter how large the project. For example, a typical query is answered in 3 CPU seconds on a Sun 3/60™ workstation, given a project software base of 800,000 lines.

`Cia` and `infoview` provide the information and the query facilities for the browsers previously described. Furthermore, they are used by other tools in the project. We are studying tools that will analyze inspection documents, looking for possible errors. These tools involve cross-checking of global project information and are made possible by the `cia` database.

Current Work

We currently are investigating several SDA features and architectural components that are important to providing an environment for higher quality software development.

Hypertext. The integrated annotator/browser was inspired by ideas arising from recent work on hypertext systems[16]. *Hypertext* refers to a computer tool for creating and reading documents composed of linked chunks that can be read in a flexible order determined largely by the reader.

The annotator/browser can be thought of as a primitive hypertext system. The *chunks* in an inspection document are the changed files and annotations. The *links* are implicit in the browser queries. For instance, a function call is linked to its declaration, and the declaration is linked by queries to every use. In effect, the `cia` database provides the hypertext link information for C language text.

However, the annotator/browser is not a full hypertext system. It allows neither explicit links nor the addition of new information to existing files in the way that annotations can be added to inspection documents. We are enhancing the system to provide explicit links, and to allow annotations to C programs and system documentation. Because the C program source and project documentation files are part of a pre-existing system, they cannot be modified. Thus, as with the `cia` information, we keep the linkage information in a separate relational database. To track explicit links through changes in the underlying files, we will need an accurate and reliable difference-detection algorithm and a change control system.

Meeting support. We are developing extensions to SDA that allows participants in a meeting, possibly in different locations, to jointly execute a single X Window System-based application. This is similar to the *Rapport multimedia conferencing system,*[17] which can also set up and manage a telephone conference call among meeting participants.

The joint program execution facility will be used with the SDA annotator to support the inspection meeting. As was previously mentioned, meetings take place in a room equipped with a workstation for each participant. The displays are placed below eye level so that participants can still see one another.

We are experimenting with holding inspection meetings with one or more remote participants. Eventually, we may be able to hold structured meetings, such as inspections, without the participants having to leave their offices. We will have to compensate for the loss of visual and other physical cues that facilitate face-to-face meetings. We will also provide an interface in the joint execution facility for setting up a voice conference among participants.

Debugging Environment. A debugging environment fashioned after those found in programming environments such as Xerox Corporation's Interlisp-D™ language[18] is being developed for SDA. The primary focus of debugging activity in this environment is a debugging-console window and text browser windows displaying C source code (see Figure 7). The debugger-console window contains panels for listing and navigating through breakpoints, spy-points (i.e., conditional breakpoints based on the values of specified variables), and expression evaluation points. Access to debugging functionality is through the browser's debugger menu.

Breakpoints are set and unset by selecting a source line in a browser window and selecting *set breakpoint* and *unset breakpoint*, respectively, in the debugger menu. Spy-points and evaluation points are set in a similar fashion, except that the name of the variable to be spied on or the expression to be evaluated is typed into the appropriate panel in the debugger console window. Whenever any of these marked points is encountered during execution, the corresponding source line is highlighted in a browser window. Any variables or expression values that have changed are then displayed in a highlighted mode in the appropriate panel in the debugger console window.

The tools used for providing these features are the C source-code interpreter `cin`,[19,20] and GNU Debugger `gdb`.[21] The user interface is built as a specialization of the browser class using the Andrew toolkit. Therefore, it provides the same user interface paradigm as the other browser tools. The use of `cin` as an underlying tool also assists incremental code development.

Summary

In a large, mature software project, the developer's main task is to correctly design and implement

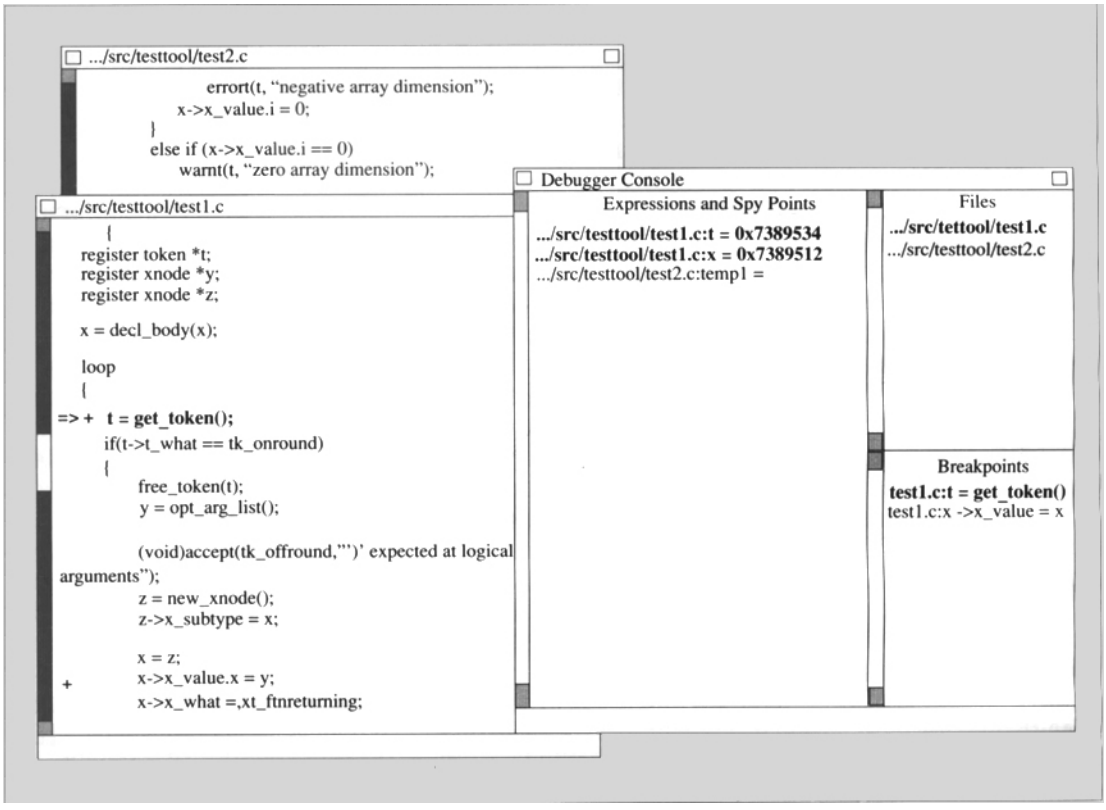

Figure 7. SDA debugging environment, showing the source files being interpretively debugged (top and bottom left), and the console window (right).

changes in the project's software base. The goals of SDA are to:

- Help the developer analyze and make the changes.
- Help visualize the system through graphics.
- Hide or make manageable irrelevant details such as computing environment architecture and file organization of the software base.
- Optimize the developer's time by providing highly integrated, workstation-based tools.

Our goal is to allow developers to concentrate on the more essential design engineering aspects of their task.

Informal exposure of the SDA toolkit to developers has yielded positive feedback and additional functionality requests. The annotator is seen as an efficient electronic alternative to the current paper based inspection process, and the integrated text browser is viewed as critical to its usefulness. The design animator is perceived as a valuable learning tool for developers unfamiliar with the software architecture. With this encouragement, we are currently conducting a more formal trial with a Definity development team responsible for the design and implementation of a specific software release. The number of participants will grow from four to twenty as the trial proceeds. All developers will use the SDA computing platform and will use the SDA tools wherever

appropriate during the development cycle. We plan to gather quantitative as well as qualitative data from this trial so we can objectively evaluate the productivity of the trial developers compared to the non-SDA development community. In addition to providing an environment for the Definity software development community to use, we also use SDA as the development environment for SDA itself.

Acknowledgments

The original prototype for the animator was based on a graphical simulator prototype, `gadd` (graphical animated design and debugging) developed by the 5ESS® project[22]. We would like to acknowledge the following people, whose technical contributions and close working relationships have been key to implementing the SDA environment: Doug Blewett, Yih-Farn Chen, Steve Goldsmith, Ted Kowalski, Tom Pedersen, and Harvey Waxman. We also thank Alec Feiner, Ron Jantz, and Dennis Weiss for their support and encouragement of this work. Finally, we thank the Definity development staff, who donated some of their time to help us better understand their work and how we might make it easier.

References

1. B. W. Boehm and P. N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1462-1477.
2. F. P. Brooks, Jr., "No Silver Bullet," *Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.
3. T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *Computer*, Vol. 22, No. 7, July 1989, pp. 36-48.
4. T. Winograd and F. Flores, *Understanding Computers and Cognition*, Addison-Wesley, New York, 1987, pp 163-179.
5. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, Vol. 16, No. 8, August 1983, pp. 57-68.
6. *Computer*, Vol. 22, No. 10, August 1989.
7. B. Shneiderman, P. S. Shafer, R. Simon, and L. Weldon, "Display Strategies for Program Browsing: Concepts and Experiment," *IEEE Software*, Vol. 3, No. 3, May 1986, pp. 7-15.
8. B. Curtis, H. Krasner, and N. Iscoeo, "A Field Study of the Software Design Process for Large Systems," *Communication of the ACM*, Vol. 31, No. 11, November 1988, pp. 1268-1287.
9. E. Greif, *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufman Publishers, Inc., San Mateo, California, 1988.
10. J. Conklin and M. L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transactions on Office Information Systems*, Vol. 6, No. 4, October 1988, pp. 303-331
11. D. G. Belanger, R. J. Brachman, Y.-F. Chen, P. T. Devanbu, Peter G. Selfridge, "Toward a Software Information System," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 22-41.
12. Sun Microsystems, Inc., *Network Programming*, Revision A, Part No. 800-1779-10, May 9, 1988.
13. J. F. Reisel and B. Shneiderman, "Is Bigger Better? The Effects of Display Size on Program Reading," *Proceedings of the Second International Conference on Human-Computer Interaction*, Honolulu, Hawaii, August 1987, Vol. 1, pp. 113-122.
14. R. W. Scheifler and J. Gettys, "The X Windows System," *ACM Transactions of Graphics*, Vol. 5, No. 2, April 1986, pp 79-109.
15. J. Morris, M. Satyanarayanan, M. H. Conner, J. H Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, Vol. 29, No. 4, March 1986, pp. 184-201.
16. J. Conklin, "Hypertext: A Survey and Introduction," *IEEE Computer*, Vol. 20, No. 9, September, 1987, pp 17-41.
17. S. R. Ahuja, J. R. Ensor, and D. N. Horn, "The Rapport Multimedia Conferencing System" *Proceedings of Conference on Office Information Systems*, Palo Alto, California, March 1988, pp. 1-8.
18. W. Tetelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1975.
19. T. J. Kowalski, Y. M. Huang, and H. V. Diamantidis, "An Interpretive Environment for Operation Support Systems," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 42-51.
20. D. G. Belanger, G. D. Bergland, and M. Wish, "Some Research Directions for Large-Scale Software Development," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp 77-92.
21. R. M. Stallman, *Gdb Manual*, Free Software Foundation, Cambridge, Massachusetts, 1987.
22. M. Moser, "Understanding Complex Software Systems Using GADD—A Tool for Graphical Animated Design and Debugging," *Proceedings of the Second International Conference on Human-Computer Interaction*, Honolulu, Hawaii, August 1987, p. 284.

Biographies (continued)

Michigan State University, East Lansing, and joined AT&T in 1965. Mr. Mukerji is a distinguished member of technical staff in the Advanced Integrated Systems Department, where he is responsible for conceptualizing, prototyping, and deploying development environments for enhancing PBX developers' productivity and the quality of the systems they produce. He

joined AT&T in 1982 with a B.Sc. in physical sciences and an M.Sc. in physics from Birla Institute of Technology and Science, Pilani, India, and an M.S. and Ph.D. in computer science from the State University of New York, Stony Brook, New York. Mr. Schmidt is a distinguished member of technical staff in the Advanced Integrated Systems Department, where he is responsible for exploratory development for the Definity 75 PBX, and also is prototyping software engineering tools such as the design animator and text browser. He joined AT&T in 1970 with a B.S. in computer science from the New Jersey Institute of Technology, Newark, and an M.S. in computer science from New York University.

(Manuscript received January 5, 1990)