AN INTERPRETIVE ENVIRONMENT FOR OPERATIONS SUPPORT SYSTEMS

Thaddeus J. Kowalski, Yean-Ming Huang, and Helen V. Diamantidis

We have developed an interactive C programming environment (`cens`) with integrated facilities to create, edit, browse, execute, and debug C programs. At the heart of `cens` is a C source-code interpreter, `cin`, that implements correct and complete C semantics; enables rapid prototyping; performs extensive error checks; facilitates incremental update; manages multiple software views; and provides a programmable command language. In this article, we discuss how a medium-sized software project, the switched access remote test system (SARTS), has benefited from using `cin` for debugging, software manufacturing, and rapid prototyping. Using SARTS as a case study, we also describe how the interactive environment catches errors and allows corrections "on the fly," thereby shortening the debug cycle by a factor of 500 percent.

Thaddeus J. Kowalski *is a member of technical staff in the Computer-Aided Information Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. Mr. Kowalski is currently examining how programmer productivity can be increased. His interests also include artificial intelligence, operating systems, computer-aided design, text-processing environments, and real-time systems. Mr. Kowalski joined the company in 1978. He has a B.S.E. from the University of Michigan, Ann Arbor, and holds the M.S.E.E. and Ph.D. degrees in electrical engineering from Carnegie-Mellon University, Pittsburgh, Pennsylvania.* ***Yean-Ming Huang*** *is a member of technical staff in the Special Services Automation Services Development Department at AT&T Bell Laboratories in Middletown, New Jersey. Mr. Huang is* (continued on page 50)

Introduction

In recent years, *increased programmer productivity* has become the war cry of almost every organization that creates and maintains software. One of the largest single factors in productivity is programmer experience,[1] both in the language or languages used and in the application area. No other factor comes close to the impact of experience on overall productivity. Experience, however, has a high market value; and experienced programmers tend to move on to greener pastures—either to another project or to other companies—leaving the task of maintaining and enhancing completed applications to less experienced programmers. Because the new programmers do not really understand the program and make many mistakes as part of their "learning curve," programmer turnover leads to maintenance headaches in many projects.

The C-language interpreter, `cin`, is oriented toward programmers who need help testing and debugging their code as they write it. It assists them by providing an environment where they can create

Panel 1. Terms and Acronyms in This Paper

ASCII	American Standard Code for Information/Interchange/Interexchange
LISP	programming language
PASCAL	programming language
PL/C	programming language
RTS	remote test systems
SABLE	software product administration system
SARTS	switched access remote test system
Smalltalk	programming language (Xerox Corporation)
SMAS	Switched Maintenance Access System

code, test it immediately, and execute it in debugging mode. `Cin` is also useful for experienced programmers who need to debug and manufacture software in a rapid prototyping environment. It facilitates their programming by:

- Strictly enforcing type checking
- Catching errors and allowing the developers to correct them "on the fly"
- Providing facilities for incremental compilation and update
- Interfacing with various product administration tools.

`Cin` decreases time spent in the debug cycle by allowing programmers to see the effects of their changes when they are made. Furthermore, it narrows the gap from code generation to system test by providing an environment where a module can be tested as soon as it is manufactured. Perhaps more important for large systems, `cin`'s ability to combine source and object code seamlessly minimizes run time by limiting the *interpreted* portion to routines under development, while the bulk of the program is executed as *compiled* code. Thus, programmers can use `cin` during maintenance to learn how existing modules operate by running them in interpreted mode, while the rest of the system runs in compiled mode. Finally, `cin` supports software reuse because new routines can be individually created, tested, and efficiently integrated with the rest of the working system.

`Cin` consists of the following:

- `cin_read`, an incremental parser and analyzer for source code
- `cin_compile`, an optimizer
- `cin_load` and `cin_unload`, an incremental loader for object and library code
- `cin_eval`, an evaluator
- `cin_print`, a universal printer.

These tightly coupled routines are the foundation of the debugger tool kit.

`Cin` command language is identical to the C programming language. Like the C programming language, `cin` is extended by predefined routines and variables. We have found this makes it easy to customize and interface `cin` to existing software product environments. See Panels 2 and 3 for a complete list of the `cin` routines and user-accessible variables.

Interactive programming environments are not a new idea. Environments for the LISP,[2,3] PASCAL,[4–6] PL/C,[7,8] and Smalltalk[9,10] languages have existed for several years. These environments are designed to improve the productivity of programmers and the quality of their programs.[11–13] Within the last few years, work has started on various pieces of a C program environment, including syntax-directed editors,[14] smart compilers, interpreters,[15–17] debuggers,[18–19] and browsers.[20–21]

Our solution to the problem combines a multiwindow editor and browser, an on-line advisor, a C source-code interpreter, and an incremental object file loader. Our programming environment is not tightly integrated; that is, the pieces work separately as well as together. This means that any one utility—advisor, browser, editor, interpreter, or loader—can be used by itself, or with one or more additional tools, in a customized environment. Thus, we have created an open architecture for rapid prototyping.

Rapid prototyping means different things to different people. To us, it means that existing modules can be found and modified easily; that the effects of changes

Panel 2. `cin` User-Modifiable Routines

`cin_break`, `cin_unbreak`, `cin_stopin`, `cin_system`, `cin_return`	set breakpoints
`cin_step`, `cin_stepin`, `cin_stepout`	step through code
`cin_spy`, `cin_unspy`	watch variables for access or modification
`cin_wrapper`, `cin_unwrapper`	watch routines for calls and return
`cin_dump`	save the incrementally-built environment as an executable program
`cin_log`	record user sessions
`cin_find_ident`, `cin_info`, `cin_ltof`	provide information
`cin_view`	control the symbol tables of source and object code
`cin_quit`	exit
`cin_reset`	start a program from its initial state

Panel 3. `cin` User-Accessible Variables

`cin_argc`	number of arguments
`cin_argv`	arguments
`cin_err_fd`, `cin_in_fd`, `cin_out_fd`	`cin`'s file descriptors
`cin_filename`	name of the currently executing file
`cin_level`	number of times the interpreter has been invoked
`cin_libpath`	search path for libraries
`cin_lineno`	line number in the currently executing file
`cin_prompt`	user prompt
`cin_stack`	run time stack
`cin_views`	list of loaded source and object files

can be seen and tested immediately; that specifications can be created and tested through prototypes; and that the separate tools all work together as a single unit.

In this article, we focus on how a medium-sized software project, SARTS, has benefited by using `cin` for debugging, software manufacturing, and rapid prototyping. We introduce the SARTS development environment and describe how SARTS is manufactured. This discussion is followed by an outline of the novel method we used to transfer the `cin` technology from research to development. Next, we show how the interactive programming environment catches errors and allows corrections "on the fly," thereby shortening the debug cycle by a 500 percent. Finally, we conclude with a discussion of open problems and future work.

The Switched Access Remote Test System

SARTS, the switched access remote test system, is a computer-based remote access and test system for special-service circuits. It was designed to provide access and testing functions through a central interface located at a special-service center (SSC), and has been serving special-services customer needs for over 15 years.

A SARTS testing station consists of a bisynchronous terminal and telephone console. Through the terminal, the circuit tester accesses the minicomputer process controller, which processes the test commands and translates them into control codes. The process controller then sends the control codes to testing devices such as the remote test system (RTS), which accesses the circuit via the Switched Maintenance Access System (SMAS). Figure 1 depicts a simple SARTS configuration.

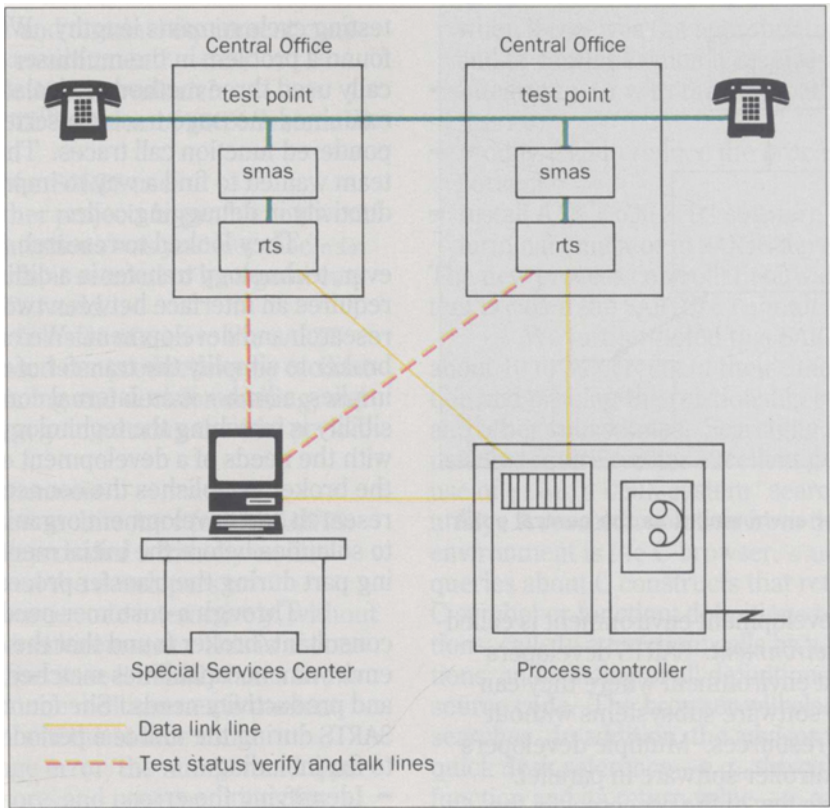

Figure 1. SARTS configuration, showing connections between the central offices, SSC, and process controller.

The SARTS Development Environment

For the SARTS development organization, it is costly and time consuming to set up equipment and real special-services circuits for each software developer to test. Moreover, software developers need to share limited laboratory resources such as process controllers, test devices, and circuits. Therefore, many simulated subsystems, such as the test station emulator and the RTS simulator, were developed to do unit and integration tests.

The SARTS software is 400,000 lines of C source code administered by the SABLE[22] software product administration system, running under the UNIX® operating system. It is manufactured using `nmake`. Each generic release in this view-pathing environment has an official node containing the approved software; all the other nodes contain changes only for the different versions of the generic release.

SARTS is composed of about 30 simultaneous processes up to 3 megabytes in size. These processes communicate through shared memory and messages. Tools have been built to allow a software developer to create libraries and subsystems for unit and integration tests. During testing, SARTS runs using the latest executable programs and data files along the view path.

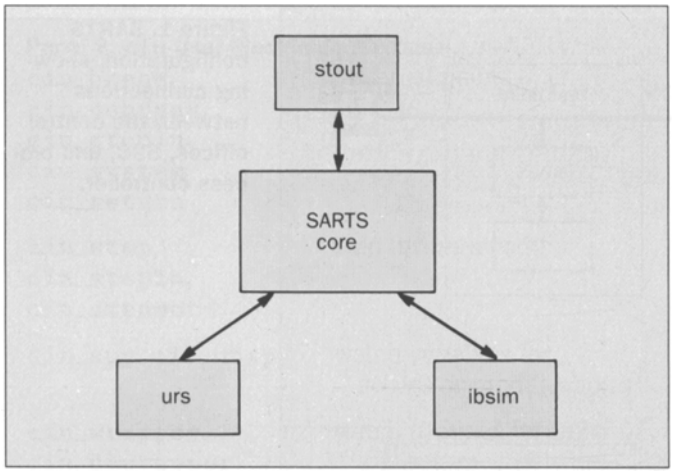

Figure 2. SARTS multiuser environment as the central point for stout, ibsim, and urs.

The standard development environment is called the *SARTS multiuser environment.* SARTS developers thus have a personal test environment where they can build process controller software subsystems without using scarce laboratory resources. Multiple developers can also test process controller software in parallel.

Figure 2 depicts a few of the processes in the SARTS multiuser environment. The UNIX system process `stout` is the test station emulator that accepts input from an asynchronous terminal and emulates the bisynchronous protocol with the rest of the SARTS core software. The `urs` process is the UNIX system RTS simulator. `Urs` can be programmed to generate different responses representing different measurements on a variety of circuits. The `ibsim` process is the inboard simulator for different kinds of remote measurement systems available in the SARTS environment.

The Transfer of Interpretive Technology to SARTS

Although the SARTS multiuser environment gives SARTS developers an adequate testing capability, the testing cycle remains lengthy. When SARTS developers found a problem in the multiuser environment, they typically used three methods to isolate the problem. They examined message traces, inserted print statements, and pondered function call traces. The SARTS management team wanted to find a way to improve programmer productivity in debugging code.

They looked to research for a solution. However, technology transfer is a difficult problem because it requires an interface between two different cultures: research and development. We employed a technology broker to simplify the transfer of `cin`. As the name implies, a *broker* is an internal consultant whose responsibility is matching the technology available in research with the needs of a development organization. As a rule, the broker establishes the connections between the research and development organizations; matches needs to solutions before the initial meeting; and plays an ongoing part during the transfer process.

Through a customer needs assessment, the consultant/broker found that the `cens` programming environment capabilities matched SARTS's technology and productivity needs. She found that a typical error in SARTS during the unit test period would take many hours to fix, including:

- Identifying the error
- Finding the file and function
- Correcting the software
- Rebuilding the troubled subsystem
- Booting up the multiuser environment
- Submitting the changes to SABLE.

This cycle would be repeated for each error found in the multiuser environment.

The broker explained that SARTS was looking for a debugger to help developers troubleshoot their code. This debugger had to be introduced into the environment of a stable operations support system with minimal risk. Though `cin` was a technical match, it needed to meet requirements in other areas:

- Was it robust?

- Did it run on computer and terminal equipment compatible with the SARTS environment?
  - — Did it run on simple ASCII terminals?
  - — Did it run on the VAX™ version of UNIX System V?
- Was it documented?
- Would it be supported for SARTS use?
- Had it been used by other projects in production work?

The answer to all these questions was *yes*. Because `cin` met all the entrance criteria, it seemed to be a good candidate for use in the SARTS world.

Later, during the transfer of `cin` to the SARTS project, the broker increased our productivity by facilitating communications, planning and decision-making, and by keeping the momentum going during the trial period.

SARTS/`cin` Multiuser Environment

We designed a new architecture for the SARTS multiuser environment. It provided a friendly on-line debugging tool that allowed the developers to:

- Change and test the source code interactively, without restarting the multiuser environment. This is of considerable significance, because it usually takes five to ten minutes just to start the multiuser environment.
- Set multiple breakpoints in the source code.
- Check pointer and range error, the number and type of arguments, and the pre- and post-conditions for standard functions.
- Load the newest object and source files from the SARTS view path.

`Cin`'s debugging facility and incremental loader provide the desired features with minor modification. All that was needed was to:

- Modify `cin` to suspend the SARTS test station emulator (the `stout` process) when a UNIX system signal or a breakpoint in software is detected, and resume `stout` after appropriate action has been taken.
- Modify the SARTS multiuser environment to allow four to six developers to run simultaneously on a single machine.
- Modify the test station emulator to suspend itself when it receives the appropriate signal from `cin`, and to continue when it receives a signal to resume.
- Integrate `cin` with the view path build tool and `nmake`.
- Modify `cin` to reduce the process run size by 50 percent.
- Install AT&T 630 MTG software and a Hewlett-Packard terminal emulator in SARTS development machines.

The new process controller software development system is called the SARTS/`cin` multiuser environment.

We further noted that SARTS developers spend about 10 to 30 percent of their time searching for a function and probing the relationship between that function and other subroutines. Searching for a function in a file usually requires either excellent guesswork or liberal use of `grep`, a UNIX system "search a file for a pattern" utility. A research solution from the `cens` programming environment is the C browser, `samuel`. It can answer queries about C constructs that retrieve references to a C symbol or function; definitions of structures or functions; calls to a function; calls by a function to other functions; and a listing of all definitions of functions in the source code. The browser will also do `grep`-like string searches. In addition, the advisor feature provides a quick desk reference—e.g., the calling sequence of a function and its return value—to a catalog of previously written modules. It scans both user-defined reuse catalogs and the standard UNIX system libraries.

The advisor's database is simple to create and maintain. Several examples have been created by developers to simplify using large library packages (such as the X library). `Samuel` is currently being tested by SARTS developers on a limited number of AT&T 630 MTG terminals.

An Example in the SARTS/`cin` Multiuser Environment

In the past, SARTS developers used message traces, print statements, and function call traces to debug their code. Each time a fix was tested, the system had to be brought down and restarted. Once the SARTS/`cin`

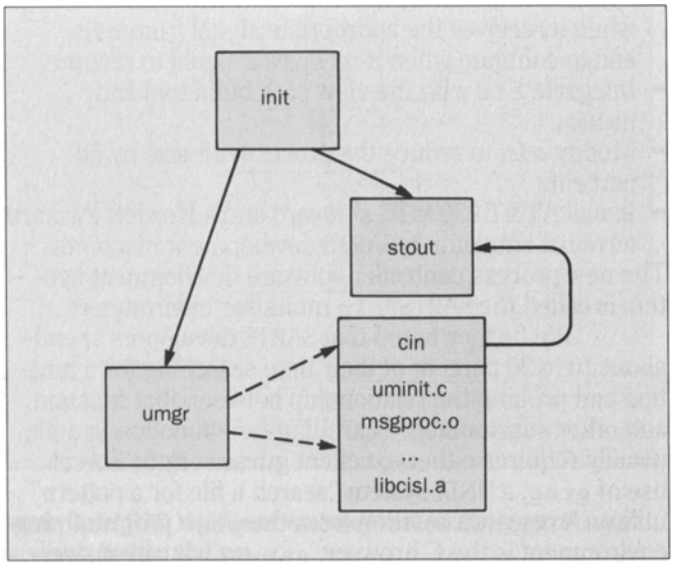

Figure 3. SARTS/CIN multiuser environment shown when user interface manager (umgr) process dies.

multiuser environment was introduced, the developers could use an environment that never needed restarting. This section provides a glimpse of the current debugging process.

In Figure 3, we are running the SARTS multiuser environment when the process `umgr` (user interface manager) dies. We may use the software debugger, `sdb`, to determine that `umgr` died at the function `umgr_init`, in the file `uminit.c`. We then copy the `uminit.c` file in the subdirectory `src/umgr`, and invoke the view path build tool, `cinbldmu`. This instructs `cin` to load `uminit.c` and all the `.o` files from the view path. The ability to combine source and object code seamlessly is a key feature of `cin`: the programmer can choose to interpret only those routines currently under development, while executing the bulk of the program as compiled code. This facility guarantees that, even in large systems, the productivity benefits of debugging with `cin` can be obtained for a small price.

Now we are ready to run SARTS/`cin` again by typing `tinit` to start the environment. This time `cin` is running the `umgr` process in interactive mode. During the initialization of `umgr`, `cin` detects an integer divide by zero, suspends `stout`, takes over input/output control, and prompts the user for input. At the line number shown by `cin`, the function `umgr_init` has an assignment statement '`average = total/time;`.' The developer types the variable name '`time;`,' and `cin` returns with value `(int)0`. To correct the problem, the developer must set the variable `time` equal to value 1. This is done by entering '`time = 1;`' after the `cin>` prompt. Now the developer can continue to test by typing '`cin_return();`.' `Cin` will instruct `stout` to continue. Note, too, that the developer can set breakpoints at various places in the source code. This causes `cin` to suspend `stout`, and permits the developer to change the source code at the breakpoint without bringing down the multiuser environment.

This example shows that a developer does not have to shutdown the SARTS multiuser environment to change the value of the variable `time`. This saves from five to ten minutes of the developer's time whenever an error is found and fixed. There is a further gain in productivity because the developer's concentration stays focused on the error and its solution; there is no 10-minute delay. A study done at Hewlett-Packard shows that it can take as much as 20 minutes to recover from even short interrupts in a programmer's train of thought.

Other Projects

The tools in the `cens` interactive C programming environment have been mixed and matched to form several customized programming environments throughout AT&T.

Interactive Graphics. `Cin` is used to develop the Xt-based OPEN LOOK® "widget" set. `Cin` provides

incremental compiling and loading for developing and testing Xt widgets. It has reduced the typical load and test time from 3 minutes to 30 seconds. It also eliminates the need for disk storage of large numbers of executable unit tests. Similarly, `cin` is also used for MIT X Windows™ widget synthesis and editing.

Large Multiprocess Systems. `Cin` is used to debug and test the SARTS multiuser environment. When combined with `sable` and `nmake`, `cin` provides a software manufacturing environment where errors are caught and corrected "on the fly" without reloading the multiuser environment. `Cin` has replaced the five- to ten-minute reload of the multiuser environment with an incremental update of under a minute. A similar effort is going on with the AT&T Definity® 75 development team.

Unit Testing. `Cin` is used to build a unit test environment for the AT&T AUTOPLEX® cellular telecommunications system. It provides a simulation of the diskless standalone environment needed to test and integrate software modules. Regression, coverage, error-path, low-level integration, and unit tests are performed, and unit test histories are recorded.

Incremental Loading And Compiling. `Cin`'s incremental loader is used to add customized functions at run time in AT&T's schematic capture system, `schema`; a simulation system, `midas`; and AT&T's place and route tool, `ltx2`. `Schema` uses `cin`'s incremental loader to decrease execution time by a factor of more than 27 (i.e., 2700 percent) and memory usage by a factor of 260 percent. The interpreter is under consideration as a simulation engine for behavioral synthesis work.

Integrated Debugging Environments. `Cin` is used as a debugger for the software development assistant environment.[23] Combined with the object generation system, MIT X Windows, and the `Andrew` system, it provides a software manufacturing environment for unit testing in the Definity 75 system.

Program Generation Environments. `cin`'s incremental loader is used to add synthesized functions at run time in the BriefCase project. Combined with an object-oriented database, the loader provides software environments where object-oriented database properties, written in the C language, can be modified and then reused dynamically.

Future Work

We are currently using `cens` to explore interface design, pedagogical methods, and programming environment technology. Instead of creating an application-oriented language, we provide an interpretive C-language interface that allows programmers to use C code to enter, manipulate, and debug an application. This design choice both decreases training time for the application and increases reuse of existing interface code. `Cens`, used with `bonsai`,[25] facilitates learning the C language by helping students quickly pinpoint programming errors; by providing data structure animation facilities; and by reducing the number of languages they must master. (Because the command language of `cin` is C, the burden of teaching yet another debugger language to the novice programmer disappears). This easy-to-use instructional environment helps students concentrate on programming concepts and algorithms rather than language syntax. Finally, our current research in programming environment design explores the use of distributed interpretation. For example, a distributed version of `cin` could reduce the debugging and maintenance cycle for switch-development applications like the Definity 75 system. We are also interested in developing a programmer's toolbox. Currently, we are enhancing `cens` with additional facilities for large multiprocess systems, e.g., intelligent testing, reverse execution, and graphical interfaces.

Conclusion

In this article, we have focused on how a medium-sized software project, SARTS, has benefited by using `cin` for debugging, software manufacturing, and rapid prototyping. SARTS uses `cin`'s breakpoint facilities

to debug multiuser environment code. The developers have benefited from `cin`'s strong type checking of C code, starting a phased clean-up of their code while handling maintenance requests. They have also benefited from an average decrease of five minutes or more in the "error fix to retry" time. The introduction of `cin` into SARTS has shown that `cin` can be incorporated into a stable operations support system with minimal risk. The first of these payoffs was expected, but the last three were additional benefits.

We have outlined the novel method used to transfer the `cin` technology from research to development. Having a leading edge research area requires the most modern methods to transfer technology in a timely and seamless fashion to development. Research gains from more timely feedback; development gains from new technology. These benefits provide strong motivation for technology transfer matches.

References

1. C. Jones, *Programming Productivity*, McGraw-Hill, New York, 1986.
2. W. Teitelman and L. Masinter, "The INTERLISP Programming Environment," *Computer*, Vol. 14, No. 4, April 1981, pp. 25-33.
3. E. Sandewall, "Programming in an Interactive Environment: The 'LISP' Experience," *Computing Surveys*, Vol. 10, No. 1, March 1978, pp. 35-71.
4. N. M. Delisle, D. E. Menicosy, and M. D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Software Engineering Symposium on Practical Software Development Environments*, April 24, 1984, pp. 49-56.
5. S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *ACM SIGPLAN Notice*, Vol. 19, No. 5, May 1984, pp. 30-41.
6. M. H. Brown and R. Sedgewick, "A System for Algorithm Animation," *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 177-186.
7. J. Archer, Jr. and R. Conway, "COPE: A Cooperative Programming Environment, Technical Report TR 81-459, Cornell University, Department of Computer Science, June 1981.
8. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, Vol. 24, No. 9, September 1981, pp. 563-573.
9. A. J. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1985.
10. L. Tesler, "The Smalltalk Environment," *BYTE*, August 1981, pp. 90-147.
11. W. Teitelman and L. Masinter, "The INTERLISP Programming Environment," *Computer* Vol. 14, No. 4, April 1981, pp. 25-33.
12. S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *ACM SIGPLAN Notice*, Vol. 19, No. 5, May 1984, pp. 30-41.
13. A. R. Feuer, "si - An Interpreter for the C Language," *USENIX Summer Conference Proceedings*, USENIX Association, Portland, Oregon, June 1985, pp. 47-55.
14. J. R. Horgan and D. J. Moore, "Techniques for Improving Language-Based Editors," *ACM SIGPLAN Notices*, Vol. 19, No. 5, May 1984.
15. S. Kaufer, R. Lopez, and S. Pratap, "Saber-C: An Interpreter-Based Programming Environment for the C Language," *USENIX Summer Conference Proceedings*, USENIX Association, San Francisco, California, 1988, pp. 161-171.
16. A. R. Feuer, "si - An Interpreter for the C Language," *USENIX Summer Conference Proceedings*, USENIX Association, Portland, Oregon, June 1985, pp. 47-55.
17. D. G. Belanger, G. D. Bergland, and M. Wish, "Some Research Direction for Large-Scale Software Development," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 77-92.
18. E. Adams and S. S. Muchnick, "Dbxtool A Window-Based Symbolic Debugger for Sun Workstations," *USENIX Summer Conference Proceedings*, USENIX Association, Portland, Oregon, June 1985, pp. 213-227.
19. T. A. Cargill, "The Feel of Pi," *USENIX Winter Conference Proceedings*, USENIX Association, Denver, Colorado, January 1986, pp. 62-71.
20. J. L. Steffen, "Interactive Examination of a C Program with Cscope," *USENIX Winter Conference Proceedings*, USENIX Association, Dallas, Texas, January 1985, pp. 170-175.
21. D. G. Belanger, G. D. Bergland, and M. Wish, "Some Research Direction for Large-Scale Software Development," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 77-92.
22. S. Cichinski and G. S. Fowler, "Product Administration Through Sable and Nmake," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 59-70.
23. C. C. Hayden, J. C. Mitchell, J. Mukerji and F. A. Schmidt, "The Software Development Assistant," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 76-90.
24. D. G. Belanger, G. D. Bergland, and M. Wish, "Some Research Direction for Large-Scale Software Development," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 77-92.

Biographies (continued)

currently working on the Switched Access Remote Testing Systems development. His interests include computer aided software engineering tools, network management systems,

software manufacturing, and testing environment. Mr. Huang joined the company in 1983. He has a B.S. from Taiwan Normal University, Taipei, an M.S. in Computer Science from the University of South Carolina, Columbia, and a Ph.D in Mathematics from the University of Minnesota, Minneapolis.

Helen V. Diamantidis *is a member of technical staff in the Software Architecture Department at AT&T Bell Laboratories in Liberty Corner, New Jersey. Ms. Diamantidis is currently analyzing what it would take to run her center as a business and what change agents need to be put in place to increase software productivity from a process point of view. Her interests also include long range planning on productivity for software development in the R & D community, economics-driven technology transfer, analysis and management of software development processes, and organizational dynamics. She has a B.S. in mathematics from the City College of New York and an M.S. in computer science from Stevens Institute of Technology, Hoboken, New Jersey.*

(Manuscript received January 13, 1990)