AN INCREMENTAL ENVIRONMENT FOR COMPUTER-AIDED DESIGN TOOLS

Judith L. Schmidt, Thaddeus J. Kowalski, and David S. Smull

We have developed an incremental computer-aided design library that provides run-time facilities to link, unlink, execute, and list functions. At the heart of the library is an incremental loader that helps designers and users customize their basic tool for their design environment. In this paper, we discuss how four computer-aided design tools have benefited from using the library for simulating C-language models, modifying and enhancing graphical user interfaces, extending command languages, and rapidly testing new functionality. Using SCHEMA and MIDAS as case studies, we also describe how the library improves quality and reduces time to market.

Judith L. Schmidt *is a member of technical staff in the Computer-Aided Engineering and Design Department at AT&T Bell Laboratories in Allentown, Pennsylvania, and* ***David S. Smull*** *is a supervisor in that department.*

Thaddeus J. Kowalski *is a member of technical staff in the Software Techniques Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey.*

Ms. Schmidt is currently working on increasing developer productivity through computer-aided software engineering tools. Her interests also include network management and artificial intelligence. Ms. Schmidt joined the company in 1988. She has a B.S.E.E. from The Johns Hopkins University (Baltimore, Maryland) and holds an M.S. in computer science from the University of Delaware (Newark). Mr. Kowalski is currently examining
(continued on page 110)

Introduction

Rapid simulation and testing is crucial to the productivity of AT&T's integrated-circuit and board designers and to the quality of their designs. AT&T computer-aided design (CAD) tools[1] provide state-of-the-art features to help designers produce complex circuits rapidly. As the CAD tools themselves become more complex, providing ever-increasing functionality to an expanding customer base, several problems become evident. (Panel 1 defines acronyms and terms used in this paper.)

The tools must serve a diverse user community that requires different features, options, and capabilities. This can lead to a proliferation of commands and feature sets that differ in subtle but significant ways. Further, a user interface that is customized for one community may not be effective in other environments. Simulators, which use C-language models to describe the behavior of components, cannot contain all available models without becoming too expensive and unwieldy. In addition, new C-language models are constantly being developed. To design CAD tools that span a large user base and satisfy these diverse needs would be an unmanageable task.

We have solved the problem by integrating an incremental loader[2] with the CAD tools. Through the incremental loader, users tailor these tools to suit their needs. Designers can add code to

Panel 1. Acronyms and Terms

API	application programmer's interface
CAD	computer-aided design
`cc`	C-language compiler
`cens`	C-language environment system
`ci`	C-language interpreter
`cin`	AT&T C-language interpreter
`lint`	C-language syntax checker
min/max	optimistic and pessimistic
`MOTIS3`	AT&T simulator
mouse	a pointing and drawing device that, when moved across a flat surface, causes the cursor to move on the screen. Buttons on the mouse are used for selecting actions or objects on the screen.
netlist	description of the electrical components in a circuit and their interconnections
R&D	research and development
`SCHEMA`	AT&T design capture tool

customize the basic tool for their environment. By linking code as needed, tool users can:

- Augment a simulator using C-language models for designs.
- Modify and enhance the tool's user interface.
- Extend the tool's command set.
- Rapidly test new functionality.

This paper describes the incremental loader and how it increases the productivity of AT&T designers.

Incremental Loader

The incremental loader is part of the AT&T C-language interpreter (`cin`), which is contained in the `cens`[3] programming environment. `Cens` was developed at AT&T Bell Laboratories in the Software Techniques Research Department and is supported by the Artificial Intelligence Systems Department. The incremental loader is based on the UNIX® system (Research Version, Eighth Edition[4]) `ld` command, enhanced to include the incremental loading of object files. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

With incremental loading, all files for a specific application do not need to be loaded at once; files can be loaded individually, as needed. The loader provides the basic commands to link a file into memory and execute functions. It maintains its own symbol table, which is updated as each file is loaded. Unlike `ld`, the `cin` loader allows undefined and multiply defined symbols.

There are two major benefits to running a program with an incremental loader:

- *Rapid prototyping.* Functions can be loaded and tested without the need to write driver programs.
- *Reduced turnaround time.* Only changed files need be relinked.

The Incremental Loader and CAD Tools. The incremental loader adds to the power and flexibility of AT&T CAD tools, enhancing the productivity of users in several ways. For example, users can link code to modify and extend the user interface and the command language. They can add menus and dialog boxes to the tool's interface, or add commands that are customized for their environment. The AT&T design capture tool, `SCHEMA`,[5] provides this functionality directly to users. Some groups have created entire products based on the incremental loading ability in `SCHEMA`. One such project, Carousel, is described later in this paper.

C-language models in AT&T simulators[6,7] provide a good example of how the loader directly affects designer productivity. These models allow designers to model the logical behavior of a circuit. The models are linked with the simulator to test a particular circuit. Before the incremental loading functionality was added, the models had to be linked directly with the simulator's object code. In effect, this technique created a new simulator, customized for a particular model. With the incremental loader, the models can be directly linked into the simulator at run time, eliminating multiple versions of the simulators.

Other tools use the incremental loader to link functions that provide a specific feature. For example, a

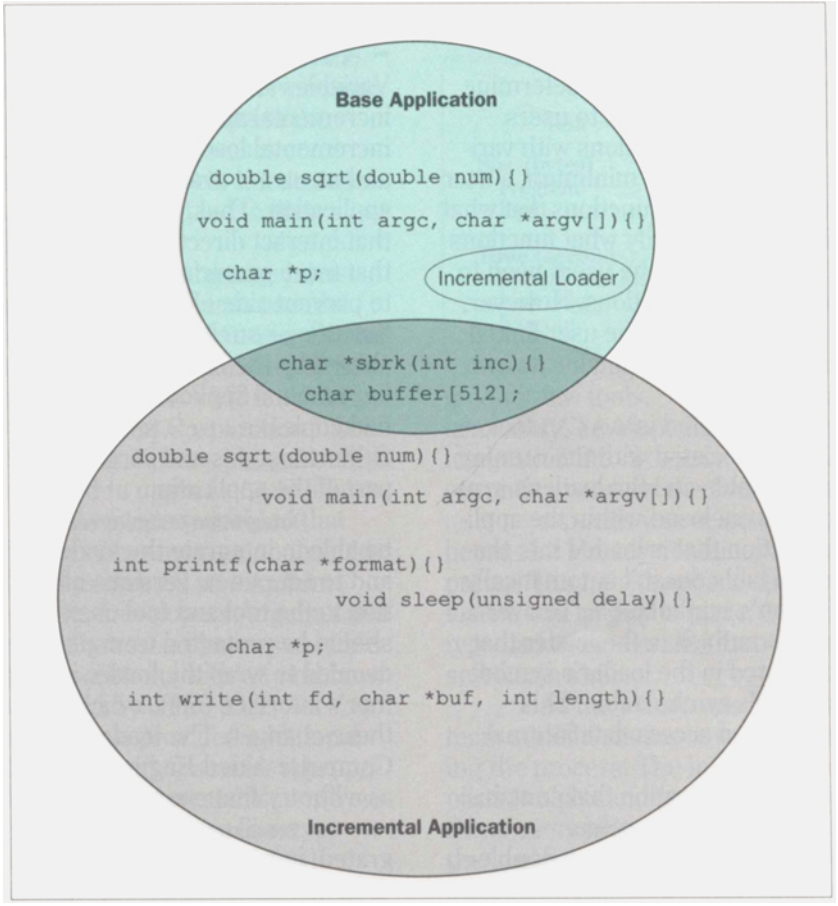

Figure 1. Application and loader symbol tables. Loader routines enable the application's symbol table to know about functions and data located in the loader's symbol table. Thus, users can link functions and access data internal to the application. Explicit function calls to the incremental loader make `buffer` and `sbrk` available to the incremental application. Other functions and variables are accessible to only one part of the application.

user selects a tool feature by clicking (i.e., depressing) a button on a mouse. The loader incrementally links, executes, and unlinks the functions without any further interaction. The tool provider can ship a small executable file, plus separate functions that supply extra capabilities. Only those functions that the user requests are linked and executed.

Detailed examples of how SCHEMA and the AT&T simulators use the incremental loading ability are discussed later.

Integration with AT&T CAD. In choosing the incremental loader, functionality was most important. Besides providing the desired functionality, the `cin` incremental loader fulfilled four basic requirements: robustness, availability on the appropriate platforms, internal support, and suitability for production-quality tools.

Once we decided that the loader was the right solution, we developed an integration plan. First, we specified the functionality required by the CAD tools. Next, we developed a method for integrating future

versions of the `cin` loader that minimizes disruption to the CAD tools, their developers, and the tools' users. Finally, we consulted with tool developers to determine how best to present the incremental loader to users.

Required functionality. After discussions with various CAD developers, we decided that, at minimum, a user needs to link and unlink files, execute functions, list what files and functions are linked, and specify what functions are active within each file. In addition, the users need to access various application data and functions. However, interaction between the application and the user-linked code is hampered because the loader maintains its own symbol table.

When the loader is integrated into a CAD tool, there are two symbol tables associated with the running tool: the application's symbol table and the loader's symbol table, which is completely enclosed within the application memory. Thus, a function that is loaded into the incremental loader's symbol table doesn't automatically have access to the application's symbol table. To overcome this problem, we used routines in the loader that make functions and data located in the loader's symbol table known to the application's symbol table. This allows users to link functions and access data internal to the application.

Figure 1 shows a base application that contains:

- A `main` function that doesn't return a value.
- A `sqrt` function that returns a *double* (i.e., a double-precision, floating-point value).
- A character pointer `p`. (A *character pointer* contains the address of a memory location that stores a character value.)
- The incremental loader.
- A character array `buffer`.
- An `sbrk` function that returns a character pointer.

Incrementally loaded into this base application is:

- A `sqrt` function that returns a double.
- A `main` function that doesn't return a value.
- A `printf` function that returns an integer.
- A `sleep` function that doesn't return a value.
- A character pointer `p`.
- A `write` function that returns an integer.

Variables and functions can be made available to the incremental application by explicit function calls to the incremental loader. In the example in Figure 1, `buffer` and `sbrk` are available to the writer of the incremental application. Thus, a user can write incremental functions that interact directly with the application, which means that more powerful functions can be written. In addition, to prevent side effects, it is also important that some functions—such as the memory-allocation functions—be shared by the base and incremental applications. If the incremental application and the base application both had copies of `sbrk`, serious problems could develop. Other functions and variables are accessible to only one part of the application.

Integration of future versions. CAD developers must be able to integrate the loader quickly and painlessly, and to adopt new versions of the loader with little disruption to the tool and tool users. In addition, the CAD tool should be protected from platform dependencies. So, we decided to wrap the loader in an application programmer's interface (API), which insulates the CAD tools from these changes. The loader and API are provided by the Computer Aided Engineering and Design Department as a library that can be linked with the application.

Presentation to users. Once the loader is integrated, it can be presented to users in two basic ways:

- In the *explicit approach*, the CAD tool provides a command line or graphic interface. The user executes specific commands to link files and execute functions.
- In the *implicit approach*, the linking ability becomes invisible to users. For example, if a user selects a button to run a certain command, the tool could automatically link the desired function, execute it, and unlink the file without additional input.

Technology Transfer

As the strategic business units within AT&T seek more benefit from basic research efforts, technol-

ogy transfer assumes greater importance within the R&D community. Before we discuss details of how the AT&T CAD tools use the incremental loader, it is instructive to see how the technology transfer between the research and development organizations was accomplished. The method we used worked well. It fostered a spirit of teamwork and cooperation, and led to direct benefits for both laboratories.

SCHEMA contained a partial C-language interpreter (`ci`), which compiled source files into pseudo-object files. SCHEMA and `ci` were available under Digital Equipment Corporation's VAX/VMS™ operating system and the UNIX operating system. (VAX/VMS and VMS are trademarks of Digital Equipment Corporation.) At the time that the VMS™ environment was phased out, the CAD developers also wanted to upgrade `ci` to allow user-defined structures. The developers decided that `cin`, designed in the UNIX system environment, would result in a faster product and presented an excellent opportunity for software reuse. Developers Angelina and Fajardo contacted Kowalski, the researcher responsible for `cin`. They discussed the problems with `ci` and how these problems might be solved by using `cin`.

Although `cin` would have provided additional functionality for SCHEMA, they decided a more appropriate solution was to separate the incremental loader from `cin` and incorporate it into the `ci` function library. Kowalski provided the incremental loader, while the developers created a prototype replacement `ci` function library and benchmarked the solutions. Table I shows that the new routines used 60-percent less memory and 96-percent less time than the original `ci` system to execute a 71-line program. With this success, Christman (another developer) assisted in porting the incremental loader to the Amdahl computer's UNIX time-sharing operating system.

A few months later, one member of each laboratory assumed direct responsibility for the transfer. Kowalski provided the basic code, expert knowledge, and technical support as the loader was integrated into the first few tools. Schmidt, a developer from the CAD laboratory, developed the API functions, performed the integration into SCHEMA, and served as the single point of contact between the two laboratories.

Table I. Size and Time for `ci` System

Test	Original system	System with incremental loader
Size (bytes)		
Program	83564	32784
Input	5984	1272
Time (seconds)		
Compile and link	3	3.3
Link only	< 1	< 1
Execute	27	< 1

SCHEMA was selected for the initial integration because it already contained a partial C-language interpreter. Thus, the interface appropriate for an incremental loader was largely in place. Moreover, the initial prototype had demonstrated that the integration was feasible and worthwhile.

As the loader was integrated into SCHEMA, the team added features in response to needs that arose during the process. The integrated product was then supplied to alpha and beta users, in turn. Feedback from these users helped the team to streamline and enhance the loader interface. Once this initial effort was completed, the loader API was released to all developers. The individual tool developers then worked to integrate the tool, with help from the CAD laboratory developer who was responsible for the loader.

As feature requests and errors were reported, some were fixed within the CAD laboratory, and others were relayed to research. One example of a major request was a port of the loader to a Sun Microsystems' Sparc® chip-based architecture. (Sparc is a registered trademark of SPARC International, Inc.) After discussions between members of research and development, the CAD laboratory agreed to handle the port, with

guidance from the research staff. The ported code was then integrated into `cin`. The result was a big win for both development and research.

Examples of Loader-Tool Interaction

Two of the first tools to take advantage of the incremental loading ability were SCHEMA and MIDAS,[6] AT&T's logic simulator with min/max timing, which produces both an optimistic and a pessimistic timing simulation. These tools use the loader in many different ways to help designers increase productivity.

Design Capture. SCHEMA, AT&T's design capture system, is a powerful and flexible application used for design jobs that include full custom integrated circuits (i.e., circuits designed at the transistor level), circuit packs, backplanes, and equipment rack cabling.

SCHEMA users have been able to link code through `ci` for some time. The interpreter, `ci`, recognized a subset of the C language. It allowed a user to access a set of internal SCHEMA functions that could be called from a user-linked function. (The functions were made available to the internal application through explicit calls to the loader, as depicted in Figure 1.) Users could also customize their environment by adding or redefining user-interface objects (i.e., menus, panels, and buttons) and creating new commands. The available functions included display-control routines, data structure access routines, and built-in command-execution routines. The functions permitted users to:

- Obtain access to the internal database.
- Change the user interface by adding buttons, menus, and dialog boxes.
- Retrieve information on graphics primitives, symbols, pages, and coordinates, as well as other variables.

However, `ci` had disadvantages. It only recognized a subset of the C language; i.e., data structures were missing. This led to some awkward coding styles. For example, multiple, parallel arrays had to be used instead of a linked list of data structures. As a result, it was difficult to maintain and enhance the code. Although `ci` did provide users with two elements that were defined as data structures, these elements did not follow the C-language standard. Moreover, `ci` was an interpreted language and was, therefore, considerably slower than compiled code.

Advantages of the incremental loader. The integration of the incremental loader provides two main advantages to `ci` users. First, users now can take advantage of all C-language features, including data structures. Second, user functions execute more quickly, because the loader functions are compiled. The incrementally linked (i.e., compiled) code is significantly faster than interpreted code, decreasing execution time by as much as 96 percent. In addition, the memory size was greatly decreased.

To take advantage of incremental loading, users create functions outside SCHEMA and, then, issue a `link function` command at the SCHEMA command prompt. Through the incremental loader, SCHEMA adds the file to the internal symbol table and updates the list of files and functions that are linked. The user can then execute linked functions simply by typing the function name at the SCHEMA command prompt.

Users can list the functions they have linked by executing the `show functions` command at the command prompt. Options permit users to select all files that are linked, all functions that are linked, functions that are linked within a particular file, or all files and functions. Linked files can be unlinked if they are no longer needed.

The incremental loading ability has become so important to SCHEMA users that some user communities are now using SCHEMA and the incremental loader as a framework over which they are integrating customized design-support systems. For example, the Carousel project uses SCHEMA to offer analog integrated-circuit designers an integrated design environment. Tools in this environment include design capture, graphics editors, simulators, netlist comparators, and layout extrac-

tors. In this environment, the internal SCHEMA database can be edited by all the tools that are incrementally linked to SCHEMA. Edits by one tool then become immediately available to all the other tools that share the database.

In addition, incrementally linked C programs can customize SCHEMA's user interface. They can add high-level functionality for several schematic entry and design entry tasks, and for directing tasks to other application programs, such as simulators, that run concurrently but do not share the internal database.

Without this incremental loading capability, the interaction between the tools would have to occur through database files. That is, when one database is changed, each of the other tools must resynchronize its own database. Clearly, this is much more time-consuming than a direct interaction.

Challenges of Integration. Integrating the loader into SCHEMA presented some uncommon challenges. For many years, SCHEMA users had been using `ci` to link their code. The change to a full C-language, incrementally linked environment had to be effected with minimum disturbance to the established user community. A substantial amount of work was done to make the conversion as transparent as possible. In general, SCHEMA users should notice little change in the way they work. Where the existing user functions are syntactically and semantically correct C-language code, the conversion is trivial.

Discrepancies between `ci` and standard C language could cause problems. For example, `ci` did not warn users when it encountered an undeclared variable; it simply assumed the variable was an integer. Code that took advantage of these differences had to be corrected.

Another feature of `ci` allowed users to specify C-language code outside the bounds of any function. This "start-up" code was executed when the file was linked. To provide the same functionality while adhering to conventional C-language syntax, users encase their start-up code in a function called `file_init`. SCHEMA has been modified to look for this function each time a file is linked and then execute the function, if present.

Quality Improvements. Users can now make use of the UNIX system tools, `lint` and `cc`, to check the syntax and semantic correctness of their code. In addition, they can use `cin`, the C-language interpreter, to debug the individual functions. These checks should result in higher quality code. In the SCHEMA user community, the conversion from `ci` files to standard C-language code uncovered many syntax and semantic errors. For example, one user's 7500-line `ci` module required over 80 syntactic and semantic corrections to conform to standard C language.

Simulation. In the AT&T simulators, MOTIS3[7] and MIDAS, the incremental loader allows users to create and debug C-language models and to create drivers and monitors for a circuit. We will describe the enhancements to MIDAS as an example.

MIDAS provides a set of function calls that permit communication between a C-language model and the simulator. For example, a model can use calls to access and report information about a simulated component, control simulation activity, or begin special actions during power-up. (These MIDAS-provided functions would be the functions that are made available to the incremental application through explicit calls to the loader, as depicted in Figure 1.) Both circuit elements and external systems (i.e., buses, memory, etc.) can be simulated through C-language models.

C-language models are functions written in the C language that describe the logical behavior of components. The models are linked with the simulator's object code to create a customized run-time simulator that calls the modeled routines as needed. Because C-language models give users a way to model circuit behavior in a familiar language, they have helped popularize model writing.

Panel 2 presents a C-language model for a simple AND gate. It shows two structures that are passed as

Panel 2. C-Language Model for an AND Gate

```
#include <stdio.h>
typedef struct {
        int     num, a, b;
} XXAND2I;

typedef struct {
        int     num, z;
} XXAND2O;

int
xxand2(in, out)
register XXAND2I *in;
XXAND2O *out;
{
        /* test inputs and outputs */
        if (in->num != (sizeof(XXAND2I) / sizeof(int)) - 1) {
                fprintf(stderr,
                        "\n\tERROR: ETYPE() - wrong # of inputs\n");
                return(-1);
        }
        if (out->num != (sizeof(XXAND2O) / sizeof(int)) - 1) {
                fprintf(stderr,
                        "\n\tERROR: ETYPE() - wrong # of outputs\n");
                return(-2);
        }
        /* AND == set the result to the AND of the two inputs */
        out->z = ((in->a) == 0 || (in->b) == 0) ? 0
                : (((in->a) == 1 && (in->b) == 1) ? 1 : 2);
        return(0);
}
```


parameters and correspond to the inputs and outputs of the gate. First, the C-language model checks the number of inputs and outputs. Next, it performs a logical AND of the inputs. Thus, it sets the output to:

- A '1' if both inputs are '1'
- A '0' if either input is '0'
- A '2' if the inputs are neither '0' nor '1'.

C-language models have been used with the simulators for some time, despite a variety of problems. For example, the models had to be linked directly to the simulator's object code, which produced a new executable file for each set of models linked. Thus, to simulate two versions of the circuit, the developer built two complete versions of the simulator. This was a tremendous

waste of disk space, because each designer had a copy of the simulator for each set of models that needed to be simulated. Corrections to a model required the recompilation of the bad section of code, plus a relink of the entire simulator. This practice directly affects designer productivity and, therefore, design schedules.

The incremental loader directly addresses these productivity issues. Rather than linking the simulator and models to create a new simulator, the user incrementally links the model into the simulator at run time. The simulator's executable file remains unchanged. If the model has an error, the designer simply relinks the corrected model on top of the old model, replacing the old model with the corrected one. The alternative would be to exit the tool, remove the bad executable code, relink the whole simulator, and restart the simulator.

The development of new C-language models illustrates another aspect of the benefits of incremental loading. Currently, `MIDAS` links to the standard, C-language model library. When a new model is added, the model is incrementally linked into `MIDAS` and tested. During the debugging process, the new model can be altered and relinked without affecting previously linked C-language models. Otherwise, each change to a C-language model would require a recompilation and relink of all C-language models, a time-consuming task.

Users can also write functions that serve as drivers and monitors for the circuit, link those functions to the program, and observe the result. The driver function generates the input vectors for the circuit, while the monitor function validates or records the outputs of the circuit. A simple example is a driver function that provides the clock to drive a counter.

Conclusion

Incremental loading allows users to customize a CAD tool to suit their needs. It increases their productivity by helping them to customize the interface, extend the command set, and write application programs. Simulator users can take advantage of the incremental loader to write and debug C-language models quickly. Loading and executing functions dynamically improves user efficiency and, therefore, affects the time to market for designs.

Acknowledgments

We thank Healfdene Goguen and John Puttress for the development and continued support of the `cin` incremental loader. We are indebted to Helen Angelina, Don Christman, and Lincoln Fajardo for the conception and prototype of the replacement of `ci` with the incremental loader.

We'd like to extend our appreciation to Lee Deschler, Jon Eiseman, Judy Miller, and Suthat Narakornpichit for being among the first to integrate the loader into their tools. They provided valuable feedback, which led to improvements in the loader interface and the addition of important features.

We especially thank Carl Seaquist for his energy, time, and interest, which made this project possible.

Finally, we thank Lisa Kowalski, Basant Chawla, John Tauke, and James Coplien for useful comments on previous versions of this paper.

References

1. W. A. Burling, B. J. B. Lax, L. A. O'Neill, and T. P. Pennino, "Product Design and Introduction Support Systems," *AT&T Technical Journal*, Vol. 66, No. 5, September/October 1987, pp. 21-38.
2. J. J. Puttress and H. H. Goguen, "Incremental Loading of Subroutines at Runtime," AT&T Bell Laboratories, Murray Hill, New Jersey, April 9, 1986.
3. T. J. Kowalski, Y. M. Huang, and H. V. Diamantidis, "An Interpretive Environment for Operations Support Systems," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 42-51.
4. *UNIX Time-Sharing System Programmer's Manual, Research Version*, Eighth Edition, Vol. 1, M. D. McIlroy (ed.), AT&T Bell Laboratories, Murray Hill, New Jersey, February 1985.
5. A. K. Bose, B. R. Chawla, and H. K. Gummel, "A VLSI Design System," *Proceedings of the 1983 IEEE International Symposium on Circuits and Systems*, Vol. 2, Newport Beach, California, May 2 through 4, 1983, IEEE, New York, May 1983, pp. 734-737.
6. J. Bierbauer, J. A. Eiseman, F. A. Fazal, and J. J. Kulikowski, "System Simulation with MIDAS," *AT&T Technical Journal*, Vol. 70,

No. 1, January/February 1991, pp. 36-51.
7. R. Beale, R. Chadha, C.-F. Chen, A. Prosser, and K.-M. Tham, "Design Methodology and Simulation Tool for Mixed Analog-Digital Integrated Circuits," *1990 IEEE International Symposium on Circuits and Systems*, Vol. 2, New Orleans, Louisiana, May 1 through 3, 1990, IEEE, New York, 1990, pp. 1351-1355.

Biographies (continued)

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 an M.S.E.E. and Ph.D. in electrical engineering from Carnegie-Mellon University (Pittsburgh, Pennsylvania). Mr. Smull is currently responsible for SCHEMA, *AT&T's design capture tool. His interests include programming environments, human factors, and computer graphics. Mr. Smull joined the company in 1982. He has a B.S. in computer science from Pennsylvania State University (University Park) and an M.S.E. from the University of Texas (El Paso).*

(Manuscript received October 17, 1990)