ADVANCED SOFTWARE TECHNOLOGY SUPPORTING CUSTOMER PROGRAMMABILITY

Donald W. Brown, Christopher D. Carson, Warren A. Montgomery, and Paul M. Zislis

Donald W. Brown, Christopher D. Carson, Warren A. Montgomery, *and* ***Paul M. Zislis*** *are with AT&T Bell Laboratories in Naperville, Illinois. Mr. Brown is supervisor of the Advanced Switch Architecture Group in the Advanced Switching Networks Department. He joined AT&T in 1962 and is responsible for exploring new software architectures for future switching networks. He received a B.S. from the University of Louisville and an M.S. from New York University, both in electrical engineering. Mr. Carson is supervisor of the Application Software Technology Group in the Advanced Services Technology Department. He joined AT&T in 1980 and does exploratory development of new switching products. Mr. Carson has a B.S. in computer science and mathematics from the University of Pittsburgh and an M.S. in* (continued on page 60)

Facilitating the creation of new services by an end user or network provider as required for Universal Information Services (UIS) requires new software capabilities. New software technology and design approaches will be required to support these capabilities. This article describes two promising approaches for providing customer programmability: *component-oriented software* and *application-oriented languages.* Component-oriented software packages software into units that can be independently developed, tested, installed, and maintained, providing a means of isolating customer-developed software from other software in the system. Application-oriented languages provide a high-level programming environment in which new services can be developed without extensive knowledge of the existing software. It also limits the functionality that the programmer can modify to a specific functional area. These approaches have the potential to support UIS customer programmability.

Introduction

The UIS vision of the future network will enable end users, network providers, or third parties to create and introduce enhanced services rapidly.[1] The software that provides this capability must meet the following requirements:

- The services and privacy of customers must be protected while new, enhanced services are introduced. This means that customers *not* receiving a specific service must be protected from errors in the software implementing the new service, and that the integrity of the network and constituent systems must be protected from failures of customer-developed software.
- Service creation should be possible without detailed knowledge of the network equipment or its software architecture. The key require-

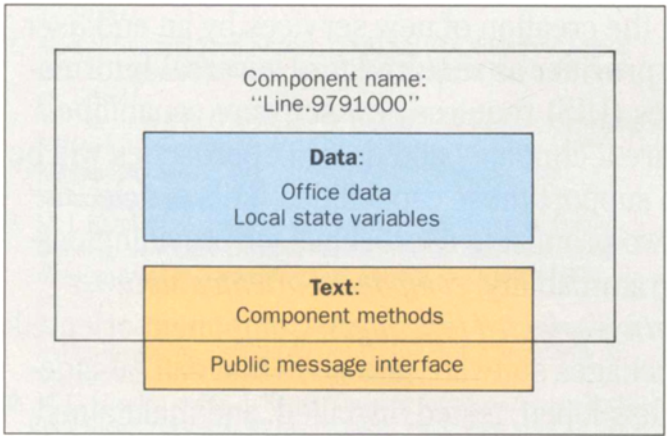

Figure 1. Software components are made up of private data and a set of public operations or methods that operate on the private data. These operations are invoked through a message interface.

ment is not the interface used for creation of new services—although this is also important—but the knowledge required by those programming the software to implement the services.

- The system architecture must accommodate continuous integration of changes made by customers and vendors and must evolve to accommodate new technology without requiring extensive revision.

Creating new, enriched services may require programming on at least two levels. At the *system* level, which provides basic capabilities from which services are built, new capabilities, such as interfaces to new types of terminal equipment, may need to be programmed. The control logic for a new service also generally affects the *application* level, which provides the specific control that exploits system capabilities to provide services.

The three basic requirements listed above apply to programming at both the system and the application levels. However, both protecting customers against program errors and programming without detailed knowledge of the system are considerably more difficult at the system level than at the application level. Our objective is to make it possible for most programming to be done at the application level, where these requirements can be more easily met, while still making it possible to program at the system level when necessary.

In this paper, we discuss two technologies that could be used together to provide customer programmability for the UIS network: *component-oriented software* (COS) and *application-oriented languages* (AOLs).

COS is a design approach for software with great potential for providing programmability at the system level. With COS, software is packaged into units called components that contain program and data and communicate with other components only through high-level messages. The components can be independently developed, tested, and added incrementally to a running system. With a suitable software architecture, COS provides a means of isolating one customer from the effects of system-level changes made by another customer.

AOLs have the same potential for providing programmability—but at the application level. An AOL is a language that is specifically designed for expressing functionality in a particular application area. This makes it possible for the programmer to describe what functionality is needed without having to learn much about the rest of the software. An AOL also provides a protected interface to programmability in that the only functional area that can be modified is the one made available explicitly through the AOL. If we have AOLs for the major areas of application functionality that need to be customized, AOLs could then be translated into software running in software components and these components, combined with others providing system-level functionality not expressed in an AOL, would make up the system level.

A previous paper of ours describes these technologies in depth and considers how their use could improve software development productivity and quality.[2] The remainder of this paper briefly examines the COS and

AOL technologies and focuses on how they support UIS customer programmability. What we describe is a vision of how these technologies could be used. This vision is supported by our current work, although at present, there is not a complete implementation of the vision.

Component-Oriented Software

Component-oriented software is a technology that allows systems to be built with hardware/software components that function as interchangeable piece parts. These components can be independently developed, refined, and provisioned within a target system, and can provide a basic unit of system software that can be independently customized. COS is an extension of object-oriented software.[3,4,5]

A software component is a unit composed of private data and a set of public operations, known as methods, that operate on the private data. (See Figure 1.) The private data contain state variables and, sometimes, office data (switch and customer configuration information) that are essential to the operation of the component. These data are directly accessible only to the component and are the only type of data in the system. The public methods are operations that perform a specific function; these are invoked only through a message interface. Each component is identified by a unique system name.

The syntax of intercomponent messages (i.e., the data representation within messages) is fixed and controlled by the underlying system environment. Messages are *self-describing*, in that the receiver of a message can determine its contents from the standard syntax without prior knowledge of the structure of the message. The semantics of messages (their meanings) are allowed to evolve to accommodate changes to the functionality of the system piece parts as long as the new meaning is upwardly compatible with the old. Each message specifies the name of a target component, the name of a method to be invoked, and, possibly, arguments to that method.

When a message is sent, the run-time support environment (a special-purpose operating system) locates

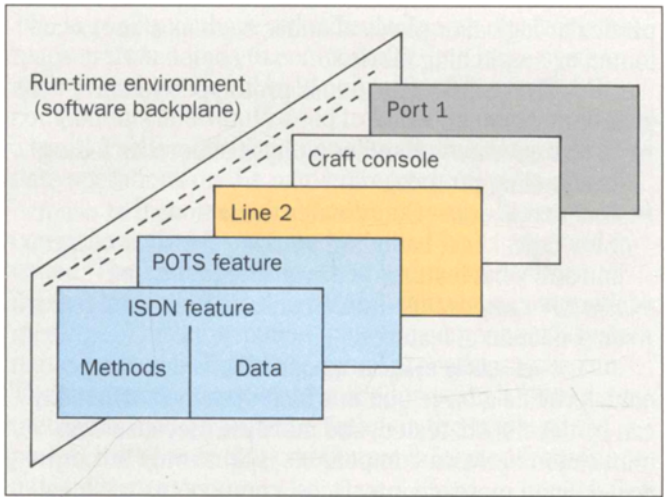

Figure 2. Software components communicate through a "software backplane" and can be customized independently without changes to other components. POTS = plain old telephone service; ISDN = Integrated Services Digital Network.

the appropriate component and method and schedules the method for execution. The run-time environment can be viewed as a type of "software backplane" (see Figure 2) that defines standards for message communication so that components can be customized independently without necessitating changes to other components.

COS is most useful in a functionally distributed system architecture. Such an architecture provides for individual system components designed to carry out specific system functions for specific entities and allows the hardware/software technologies used within a component to be customized to suit its function. Many software architectures use functional partitioning by packaging all the software for a particular functional area in a software module. In addition to this, we suggest distributing the implementation of each function to individual components, each of which handles the function for a

particular logical or physical entity, such as a line, a customer, or a switching fabric.

The components in our prototype system include:

- *Port component*—Handles a physical port, including its signaling protocol.
- *Line component*—Coordinates the actions that occur for a logical line, including tracking the state of the line and what features are available to the line.
- *Feature component*—Implements an individual feature or collection of features.

Customizing Systems with COS. Software components provide a basic unit in which system functionality can be developed, tested, and installed. Because communication between components is done only through well-defined message interfaces, components are ideal units for customizing system functionality. Customizing a system involves changing the implementation of one or more components and/or creating and installing new components. Our view of how this would be done involves two software development environments, a *modeling environment* and a *target construction environment.*

In a modeling environment, a designer manipulates a complete scale model of the system consisting of models of the individual components. Interfaces between components and behaviors of components model exactly what happens in the target system. The modeling environment provides extensive tools for manipulating the model components, allowing a designer to determine the impact of proposed modifications or additions rapidly and, thus, provides significant benefits in software quality and productivity.[2]

The construction of the actual components to be placed in the system is based on the models but uses a target construction environment. This environment provides all the tools usually found in a software development environment, plus tools for installing and testing components.

To program system components, a customer would use the following steps:

- Plan and specify changes to be made.
- Use the modeling environment to develop models of new and modified components that behave as intended. Verify that the system implements these intended behaviors.
- Once the model is correct, place the modified components in the target construction environment. Verify each component by testing its behavior against the corresponding model component.
- Load components onto a running switching system and test. New versions can be loaded dynamically as changes are made to correct problems.
- When satisfied that the new components function properly, update the system configuration to have these new components provide services to the appropriate customers.

Functional partitioning can greatly ease the testing process. For example, a change that requires a new port component to support a new form of signaling can be tested by assigning only the test line to the new port component. Other lines on the same switch would continue to use their current port components and would be unaffected by problems in the new component. (Severe errors in a component under test may cause excessive resource consumption, or may cause other components to fail. Audits done by the run-time environment and by individual components can be used to recover from such errors without extensive service disruptions.)

Which components must be modified or created to achieve a particular customization is determined by the functional partitioning of the system into components. It is best to have an architecture in which most customer changes involve modifications to a single component or a small number of components. We can try to anticipate likely customizations and choose a system partitioning that makes this easy, but no one partitioning will provide a framework in which all new feature additions will consistently map onto single components.

Implementing COS. Component-oriented software requires extensive support from the run-time environ-

ment. The run-time environment must provide:

- Support for self-describing messages
- A mechanism to locate a component from its system name
- A mechanism to invoke a component's methods
- Facilities for installing, initializing, and updating components.

The run-time environment can be structured as a *kernel* that implements intercomponent messaging and scheduling of components and a *collection of system components* that provides other operating system services such as memory allocation, timers, etc. The kernel is the only part of the system that is not structured as software components. As such, it is desirable to keep the kernel as small as possible.

Intercomponent message protocol. As new components are being tested and installed, new and old versions of the same component would be running in the same system. Component interfaces would evolve over time to include new parameters or change old ones. This means that a new version of a component would need to interact properly with older versions of other components, and vice versa. Component designers need to deal with the semantic issues that arise from this, such as what to do when an extra parameter is passed or an expected parameter is missing. The kernel, however, must provide an intercomponent message protocol that allows each component to determine what information is in a message without prior knowledge.

This is what is meant by self-describing messages. Conceptually, the message is a set of keyword/value pairs in which each value is also self-describing. The kernel must provide an efficient implementation of such a protocol to support component programming and update.

Locating components. To deliver messages to components, the kernel must locate a component from its system name. The system name does not dictate the processor on which a component runs; in fact, the location of a particular component may change over time. The kernel must have a distributed name directory that maps system names to components. In a large system, each processor would hold only a small fraction of all components and would need to know the location of only a small fraction of all components. Both lookup and update operations on the name directory must be efficient.

Method invocation. When a message is sent to a component, the correct method must be located and invoked. The name of the method is determined by a method-name field that is part of every intercomponent message. This information, plus the component's system name, is used to locate the actual procedure to be run. The determination of what procedure to call for a given method is made from information in the target component itself, once that component is located from the name directory.

Because a component determines what specific procedure to call for a given method name, it is possible for a single method to have different implementations according to the type of component. For example, it is possible for all the components in a system to have a *recovery* method with universal semantics but with different implementations based upon the component type. If *port* components are provided in a switching application to implement the signaling protocol for subscriber terminals, they might also have a common set of methods (*report origination, alert terminal, request digits*, etc.) with different implementations for different types of lines.

Prerequisites for a reusable software component design include:

- The selection at run-time of which procedure to invoke based on the method name and the type of the component.
- A design practice of building sets of components with universally understood method semantics.

These allow program designers to characterize component piece parts in terms of their method semantics rather than in terms of their specific implementations. All component classes that implement a given set of methods are considered "plug compatible" from the

perspective of components that interact with them.

Component installation and initialization. When a new component or a new version of an old component is installed, space for its text and data must be allocated and the text and initial data must be loaded into memory. This provides a prototype that can be used to create new components or update existing components. When a new component is created, it is registered in the name directory and an initialization method is invoked. Typically, components that represent lasting entities, such as lines, ports, or features, will be created through a system administration interface. This interface is a collection of components whose purpose is to create other components and supply them with the appropriate office data. Existing components can be replaced by similar procedures. Because all references to a component go through the name directory, updating the name directory is all that needs to be done to make the new component available.

COS provides an effective infrastructure for building systems from interchangeable parts. By partitioning the system architecture into components, we can address some of the issues highlighted in the introduction. The task of building or modifying the application functionality within a particular component would remain difficult, however, without the second technology we are proposing: application-oriented languages.

Application-Oriented Languages

An application-oriented language is a computer programming language that makes it easy to write software for a specific application. AOL software is easy to write because the AOL allows the software to be expressed in ways that are natural to the application programmer. For example, the interface to a spreadsheet program is an AOL well-suited to the application of financial analysis. A spreadsheet program allows financial analysis software to be expressed in familiar and natural concepts: rows and columns of numbers and arithmetic operations on those numbers. It is easy to write financial analysis software using a spreadsheet—that is, it is far easier than doing the same calculations in a high-level computer programming language such as C or Fortran. It is so easy to use a spreadsheet that many spreadsheet users don't even think they are programming a computer—but they are. The difference between laying out a spreadsheet and writing a financial modeling program in a conventional language is primarily the degree of difficulty.

We have studied the application of AOLs to customer programmability of switching systems for the last three years.[6] The focus of our work has been the discovery of the natural forms of expression—what we call the design paradigms—for various applications in switching-system software so that we can build AOLs around these design paradigms. We have concluded that one AOL cannot provide the broad spectrum of functionality that is needed for the UIS vision of the future network; instead we foresee many AOLs, each providing access to specific switching functions in much the same way that existing switching systems are built from many different subsystems. AOLs would program the functionality of software components that implement the system. Thus, the partitioning of the system into components and the design of AOLs must be considered together. Below are some examples of applications expressed in the terminology of today's network that we have considered for customer programmability:

- *Call processing.* Call processing software handles the high-level call functions from the time the receiver is picked up until the time it is placed back in the cradle. It is call-processing software, for example, that decides when to give dial tone. It is also call-processing software that handles custom calling features, such as call waiting and three-way calling.
- *Feature interaction.* Feature interaction software decides when one feature, such as call hold, takes precedence over another feature, such as three-way calling.
- *Digit analysis.* Digit analysis software interprets the digits that the end user dials. Digit analysis software is responsible for deciding when a complete phone num-

ber has been dialed, and is responsible for interpreting any special codes that the customer may dial.

- *Routing.* Routing software decides how to interpret a dialed number, and where in the network a call to a dialed number should be routed.

People are often startled to see such a long list of applications, each of which will turn into a custom AOL. They fear that it will be difficult to learn all these individual programming languages. Our experience shows that each AOL is easy to learn once the application is understood, and that it is easier to learn and work with a collection of AOLs than it is to learn each of the applications and a general-purpose language to program the applications.

Service Creation Using AOLs. AOLs could be used to provide a programming language interface to customer programmability, but, to fully exploit their power, a complete *service creation environment* (SCE) is required. The SCE is a system containing all the software tools that the customer would need to program functions in the switching network. The SCE would be built on a computer system, separate from the computers that control the network and dedicated to the function of new service creation. The service creation environment is analogous in function to the conventional software development environment. Tools in the service creation environment include:

- AOL translators.
- Special-purpose software testing tools, customized for the AOLs that the SCE supports.
- AOL editors. For some AOLs, a general-purpose text editor would be sufficient as a tool to enter and update AOL programs. For other AOLs, particularly those intended for end users, a special-purpose editor customized for the AOL would be more appropriate.
- Version management tools that allow consistent sets of files containing AOL programs to be maintained.
- Deployment tools that move completed AOL programs into the network and move programs into a restricted part of the network for testing. Deployment tools must be capable of "backing out" a new AOL program and replacing it with an old version if the new program proves to be malfunctioning.

These are some of the tools that might be available on an SCE. Some of these functions are facilitated by the use of COS underlying the AOLs. For example, installing new programs and backing out changes may be done by installing or backing out the underlying components. Not all users would have access to all tools. End users, for example, might have access to only a subset of AOLs and a subset of testing and deployment tools.

The process the customer uses when programming depends on the extent of the new service or the change to existing service. For simple changes, such as changing the routing of calls to an end-user's regional processing center, the customer would:

- Plan the changes
- Access the SCE
- Use the version management system to create a new version of the existing routing AOL program
- Make changes to the new version of the program using the appropriate AOL editor
- Test the program using SCE-based AOL testing tools
- Translate the program into a final form and deploy the program into the network
- Run a final test of the service in the network.

For larger or more complex service creation efforts, the basic steps on the SCE remain the same. In addition, however, programs in multiple AOLs would be created or changed and several iterations of the edit-and-test cycle would be necessary. A significant test effort would be planned for the service in the network. This test effort would involve temporary, dedicated network facilities.

Highly complex service creation efforts might require programming in areas not covered by AOLs and, thus, would require programming or modifying software components in a general-purpose language. This would place additional requirements on the SCE for directly invoking the procedures to test, install, or back out software components.

Building AOLs. Two pieces of software must be

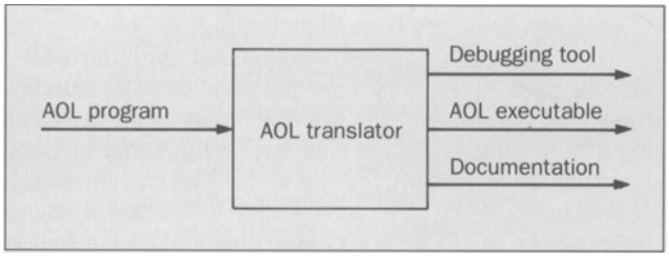

Figure 3. An AOL translation results in custom debugging tools and special program documentation as well as a translated, executable program.

created to turn an AOL from a concept into a useful, functional programming tool. The first is an AOL translator, which reads the human-readable AOL program and translates that program into a form that is convenient to execute. There is a strong similarity in function between an AOL translator and a compiler for a typical programming language. One major difference between an AOL translator and a compiler is the effort necessary to create them. Compilers are viewed as being difficult to write. Software-development tools introduced recently make it straightforward to build sophisticated AOL translators with a modest investment of effort.[7] As a result, translators for multiple AOLs in a switching system can be provided at modest development expense.

Figure 3 shows that the AOL translator can also produce custom debugging tools and special program documentation as well as a translated, executable program. We have found that the modest additional investment in the AOL translator to produce this additional output pays handsome dividends.

The second piece of software that must be created is known as a *virtual machine*, or VM. A VM is a collection of software that executes the translated AOL program. The term "virtual machine" was chosen because the VM software executes the translated AOL program as if a custom, application-oriented machine had been built to execute the application. The VM for a particular AOL will, in general, consist of one or more software components that interact with other software components in the system to execute AOL programs. Much as a compiler produces code for a physical machine to run, an AOL translator builds code for a virtual machine to run. (See Figure 4.)

One significant benefit that comes from using AOLs is that the application program in the AOL will run in any environment in which the VM can be implemented. Figure 5 shows that if the VM is reprogrammed in a new environment, all existing application software in the AOL can be ported to the new environment without change. For example, a VM specially instrumented for testing can be created in the service creation environment and a VM designed for operational performance can be created for the network switching product. In this way, AOL code can be tested directly on the SCE and deployed in the network only for final testing and installation. The same AOL software can also run in a number of different switching vehicles, provided that each switching vehicle contains a VM for the AOL.

Benefits of COS and AOLs

In introducing this paper, we cited three requirements for the customization capabilities needed for the UIS network. The approaches that we have presented, component-oriented software and application-oriented languages, address these requirements directly.

Protecting Services and Privacy. At the system level, COS segments the software architecture into self-contained software components. Because each component can access others only through well-defined public interfaces, it is possible to build components defensively, so that failure of a newly introduced component will not disrupt normal operation of other components. By defining separate components for each logical or physical entity that we wish to control independently, we can isolate the users from failures in the services that they are not using.

At the application level, AOLs provide focused, protected interfaces to particular areas of system

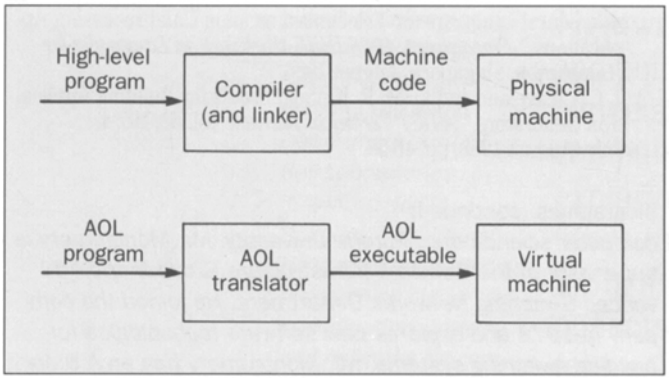

Figure 4. An AOL translator builds code for a virtual machine to run in the same way that a compiler produces code for a physical machine.

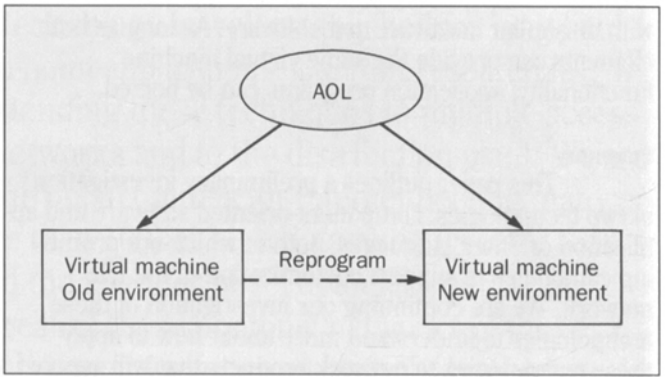

Figure 5. Porting AOLs between systems.

functionality. The functionality to be customized can be limited by the interface provided through an AOL to prevent an error in an AOL program from disrupting services to other customers.

Minimizing Specialized Knowledge. Both COS and AOLs allow programmers to customize services without a specialized knowledge of that area. In a COS, the functionality that needs to be customized for any particular purpose generally resides in a few components. A service provider need only know the internal structure of the components that must be modified, plus the public interfaces of other components. If areas that will require frequent customization are known in advance, the architecture can be tuned so that most of the desired customizations involve very few components. Furthermore, the standard message syntax eliminates a common source of difficulties in programming complex systems (i.e., the need to learn the syntax of interfaces to components).

At the application level, AOLs provide a highly focused view of the application functionality. As long as the desired customization falls clearly in one of the areas covered by an AOL, the language allows a customization with minimal knowledge of the rest of the system. Application designers can work at maximum efficiency by focusing on their application and not on the rest of the system.

Supporting System Evolution. COS allows a system to evolve to exploit new technology or include new applications. Components can be moved between processors to exploit new processor and memory technology, with minimal impact on components. Partitioning a functional area into individual components serving individual entities allows hardware entities to be upgraded individually with minimal impact on the rest of the system and allows old and new hardware and software components to coexist in the same system with minimal software complexity. When software is packaged in components, customized software can be ported between network elements in response to changing technology or needs.

AOLs provide a clean level of separation between application functionality and system hardware and software. They provide a clean definition of the application interface that must be preserved when modifying system functionality, so that independent evolution of customized applications and vendor hardware and software can take place. AOLs also provide the potential for porting application functionality between network elements

with dissimilar hardware and software. As long as both elements can provide the same virtual machine functionality, application programs can be ported.

Summary

This paper outlines a preliminary investigation of two technologies: component-oriented software and application-oriented languages, both of which are promising candidates to support customization of the UIS network. We are continuing our investigation of these technologies to understand more about how to apply these technologies to network products that will support the UIS vision of the future network.

References

1. A. Heiber, "An Overview of Universal Information Services: Concepts and Technologies of Future Networks," *AT&T Technical Journal*, Vol. 68, No. 2, March/April 1989, pp. 5-13.
2. D. W. Brown, C. D. Carson, W. A. Montgomery, and P. M. Zislis, "Software Specification and Prototyping Technologies," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 33-45.
3. G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211-221.
4. J. O. Coplien, "The Object Paradigm as a Future System Life Cycle Method," *Proceedings of NCF/86*, National Communication Forum, Chicago, November 1986, pp. 1110-1115.
5. B. Stroustrup, "What is Object-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming*, Paris, June 1987.
6. T. L. Hansen, W. E. Hyatt, W. H. Leung, and P. M. Zislis, "A Non-procedural Language for Telecommunication Call Processing Applications," *Proceedings, 1986 IEEE Workshop on Languages for Automation*, Singapore, August 1986.
7. J. C. Cleaveland and C. M. R. Kintala, "Tools for Building Application Generators," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 46-58.

Biographies (continued)

computer science from Purdue University. Mr. Montgomery is supervisor of the Software Infrastructure Group in the Advanced Switching Networks Department. He joined the company in 1978 and explores new software technologies for building switching systems. Mr. Montgomery has an A.B. in mathematics and engineering from Dartmouth College and an M.S. and a Ph.D. in computer science from the Massachusetts Institute of Technology. Mr. Zislis is head of the Advanced Software Technology Department, which explores advanced software architecture, software development environments, process modeling, specifications support, and expert systems. He joined the company in 1969 and has a B.S. in mathematics and computer science from the University of Illinois, an S.M. in information sciences from the University of Chicago, and a Ph.D. in computer science from Purdue University.

(Manuscript received January 23, 1989)