SOFTWARE SPECIFICATION AND PROTOTYPING TECHNOLOGIES

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 is responsible for exploring new software architectures for future switching networks. He joined AT&T in 1962. He received a B.S. degree from the University of Louisville and an M.S. degree 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 does exploratory development of new network database products. He joined AT&T in 1980. He received a B.S. in computer science and mathematics from the University of Pitts-*
(continued on page 45)

It has been observed that the challenge of improving software productivity and quality is not just to "do the thing right" but to "do the *right thing* right." Doing the right thing means building products and services that meet real customer needs. *Specification* is the first step in deriving system design from an understanding of customer needs. Technology to support software specification can, therefore, provide productivity and quality benefits that accrue through the entire product life cycle. *Prototyping* is an approach to system design in which a model of the system is quickly built and iteratively refined so as to become increasingly realistic. Experience with the evolving prototype leads to an improved understanding of system functionality, dynamics, and performance so that the resulting products and services are known to match customer needs before they are built. This article describes two promising approaches—software architecture modeling and application-oriented languages—for improving software development productivity and quality through support for software specification and prototyping.

Introduction

It is essentially impossible to think through requirements and design for a large system in sufficient detail for it to be implemented without subsequent design changes. Even if the design faithfully follows the requirements, experience with the initial implementation often leads to revised requirements. Before a system performs to a level that completely satisfies customer needs, this process may have to be repeated a number of times. Very few designs for major new systems have been successful without an extensive redesign.

Software specification and prototyping technologies contribute

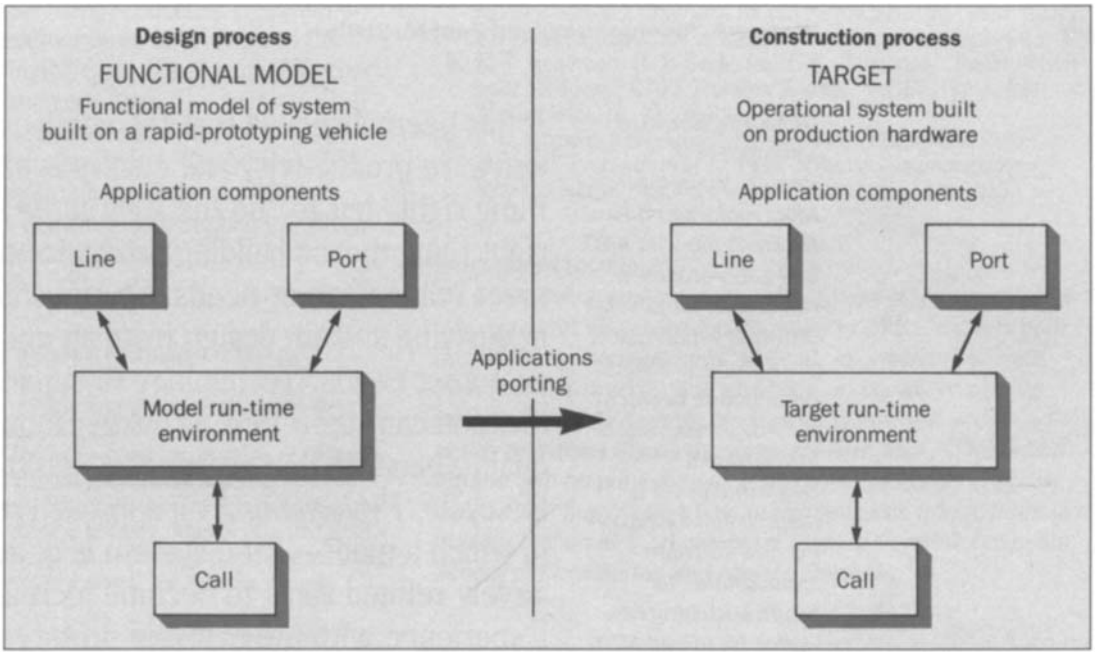


Figure 1. Model design is ported to the target without loss of structure. Components are mapped one-to-one from model to target. Component interfaces are exactly preserved, and component internal elements are reimplemented.

to higher productivity and quality by providing early assurance that a product or service will indeed meet customer needs. Modeling and early evaluation of multiple design strategies can assure cost-effective designs to meet customer needs. In addition, through portability of features and services from one product/hardware/software environment to another, consistency of feature execution across a range of products can be assured. At the same time, software can be reused to a greater extent and correctness is preserved.

Software architecture modeling, the building of a complete scale-model prototype of a product or service, is done by a small group of high-level system designers and architects, who work closely with systems engineers familiar with customer needs. The prototyping is done with a collection of hardware and software support tools that allows system designs to be implemented and evolved rapidly so that a variety of design alternatives can be evaluated. Systems engineers give feedback throughout the prototyping process. This lets system designers evolve the prototype to a design that is a highly cost-effective realization of functionality to meet customer needs. After a complete scale model has been validated, the application code is reimplemented (but *not* redesigned) for the actual target system.

An *application-oriented language* (AOL) is a language for expressing specifications for features and services in a particular application domain. An AOL uses the terminology and attributes of the application domain to identify feature and service functionality. AOLs describe what the functionality is without going into the details of how the functionality should be implemented. By creating an AOL for an application, then writing application software in the AOL, substantial software life-cycle cost benefits are achieved compared to the conventional approach of coding an application directly in a high-level, procedural programming language. Moreover, an AOL for an application can be turned over to other, possibly less expert, programmers and used as a tool to simplify application programming. This latter capability makes AOLs ideal for turnkey systems in which the customer is given the flexibility to program parts of the system.

Application-oriented languages can be used with software architecture modeling. In particular, AOL design and implementation can be modeled with the rest of the system. Having modeled an AOL design and implementation, the AOL can be used to build services and features that are part of the model. The AOL definition, the AOL implementation, and the services and features implemented using the AOL can be evaluated and evolved in the modeling environment. The evolution continues until the application designers are satisfied with the capabilities of the AOL, the performance of the AOL implementation (translator and virtual machine), and the services and features implemented by the AOL programs.

Software Development Prototyping Process

The software development prototyping process is partitioned into a "front end" design phase and a distinct construction phase (see Figure 1). Each phase employs a software development environment optimized for its particular role in the process. The objective of the design process is to specify completely the software components required for the system being designed by a relatively small number of highly skilled system designers. In the construction phase, the software components specified in the design process are implemented and integrated on a production target system.

System Design Phase. System design is accomplished through software architecture modeling. For large system designs, rapid iteration of system requirements and the corresponding system architecture is accommodated as part of the front-end design process. The software development environment gives system designers the capacity to explore design alternatives and determine their effect rapidly. The specification of system requirements and system architectures is essentially a "bootstrapping" operation. Succeeding design changes are likely to be derived from a more thorough understanding of system operation gained by observing the effects of previous system modifications.

The system architecture is controlled through the front-end design process. If changes to requirements or the architecture are needed, they are made to the high-level design controlled by the front-end process and subsequently reflected in the actual system implementation. The ability to add changes incrementally to the system is essential for an orderly progression of change to the system.

Component-oriented design. To support system prototyping, the system is designed and modeled with hardware/software components that can function as interchangeable piece parts. The components can be independently developed, evolved, and provisioned within a target system. The goal is to provide for a highly modular system with fine enough module granularity so that software updates can be made through a succession of changes to independent software components. The key to providing this increased degree of independence between software components lies in the type of intercomponent interfaces supported by the *system infrastructure.*

The system infrastructure consists of the collection of underlying hardware, software development environment, and run-time (operating system) environment. The component interfaces supported by this system are defined by messages using a standard intercomponent protocol supported by the infrastructure. The meanings of messages—their semantics—are allowed to evolve in an upwardly compatible manner to accommodate changes in the functionality of the system piece parts. The system infrastructure that implements these protocols facilitates the independent evolution or customization of components without requiring changes in other components.

Component-oriented software is an extension of object-oriented software.[1-3] Advanced programming languages, such as C++,[4] can be very useful in supporting object-oriented software development. A software component is composed of private data and a set of public operations known as *methods* that operate on the private data (Figure 2). The private data contain state variables, local variables, and in some cases office data that are essential to the operation of the component. (Office data are an extensive collection of data that represent all switch and customer configuration information in a switching office.) These data are directly accessible only by methods contained in the software component, and they are the only type of data in the system. There are no global data accessible by two or more components.

The public methods are operations that perform

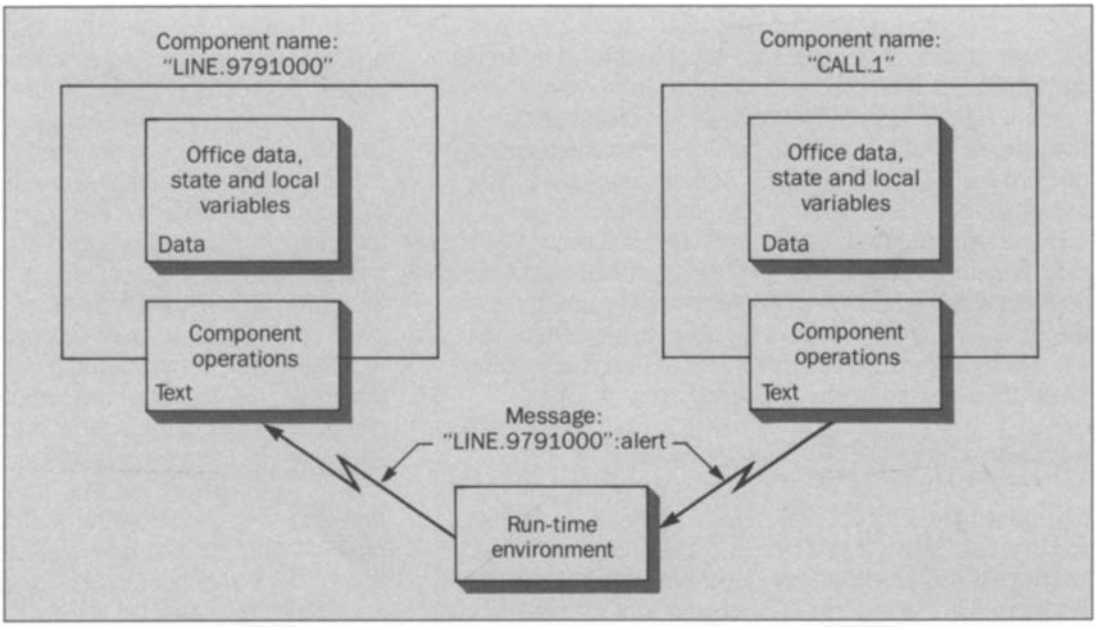


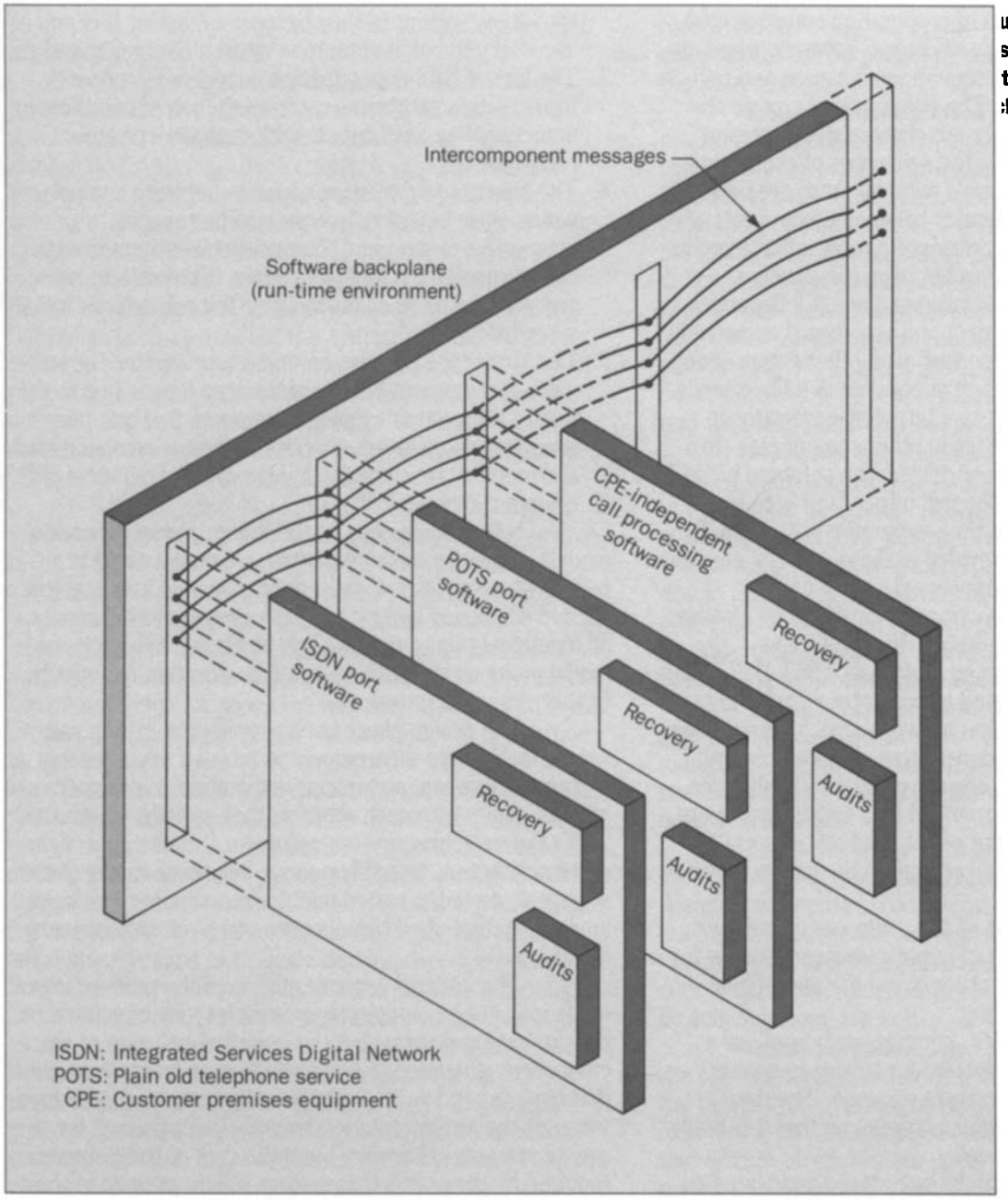
Figure 2. Structure of component-oriented software.

an application function. They are invoked solely through a message interface that specifies the name of a recipient component, the method of the component to be executed, and any parameters needed by the method. Messages are routed between components by a run-time environment (special-purpose operating system) that can locate a component by its unique system name in a distributed processing environment and can initiate execution of the method specified within the message.

The principal characteristic of a component that distinguishes it from other software constructs is the absence of data dependencies between system components. There are no shared compilation-time or run-time data between components. Consequently, components have no shared header files and a particular component has no knowledge of the data representation within any other component. It also has no knowledge of the data representation required to build intercomponent messages because this is a function provided by the run-time environment. Furthermore, components are constructed without explicit reference to the system names of other application components. The identities of other components required to carry out an application function are passed to the component at run time in a variety of ways. In many instances, names of associated components are obtained from the office data.

While component-oriented design eliminates all data dependencies between software components, it is not possible to eliminate semantic dependencies between components. In particular, each software component is dependent on the functionality of, and interfaces to, other software components. If the functionality of a software component is changed, then it will be necessary to examine the implementation of all components that depend on it and to revise those components, if necessary. If the interfaces to a software component are changed in a fashion that is not upward-compatible, then it will be necessary to examine the implementation of all components that interface to it and to revise those components appropriately.

The need to revise other components in a system when some component undergoes certain types of change is not a defect of component-oriented design; it is inherent



ure 3. Components software piece ts in a software kplane.

in the design of any system of cooperating software modules. With component-oriented design, however, there are fewer types of change that require examination and revision of other components. The only types of change that can affect other components are changes in component functionality and changes in the semantics of component interfaces. There are no global data, so there are no syntactic or semantic dependencies between components related to global data. The message-passing interface eliminates syntactic dependencies between components.

The high degree of intercomponent independence makes it possible to implement and provision a component on a running system independent of all other components. For a large class of changes, it is possible to edit, compile, and download a component to a target system without requiring coordinated changes in other components. In a sense, a component can be regarded as a software piece part that can be built and plugged into a "software backplane" provided by the run-time environment (Figure 3). Additional functions are provided in the system by plugging in new components or modifying existing ones.

The application functionality provided by a component is a matter of system design. In a prototype component-oriented switching system at AT&T Bell Laboratories, components are used to control peripheral entities such as lines, trunks, and the network switch fabric. Other components have been designed to provide call control, routing control, processor downloading, etc. In all cases, the scope of functionality provided by a single component is moderate, and the amount of data and text associated with it is consequently small enough to be managed by a single programmer.

The independence of the components and their limited size yield a number of important consequences for the software development process and the operational characteristics of the system:

1. The time required to edit, compile, and download a component to a target system for testing is greatly reduced, compared to current systems. The time needed to make and install a change, so that it is ready for testing, is only minutes.
2. Independent changes can be introduced and tested on the target system as they become available. It is not necessary to batch numerous system changes together.
3. The lack of data dependencies between components frees system programmers from the job of coordinating numerous low-level details with designers of other components.
4. The absence of data dependencies between components, their lack of reference to other specific component names, and their communication through a common software backplane make it possible to reuse components more easily in other related, but distinct, systems.
5. The size of the physical resource required for the software development environment is no longer tied to the size of the system's generic software. A single programmer workstation provides sufficient computational and memory resources to implement and unit-test any system component.

System prototyping. In the design phase, functional models of the system architecture are constructed to reflect the semantics of system operation. Extensive testing and debugging is done to verify that the model meets all functional requirements. The functional behavior of a model provides feedback on system operation to support further design iterations.

The design phase uses a rapid-prototyping software development environment in which a small number of systems engineers, architects, and high-level designers can interact easily with each other as they specify, design, and verify the switching system software. It is the goal of the system designers to build an exact functional model of the system down to the component level, including a complete implementation of all components and all intercomponent interfaces.

The internal implementation of the model components may not be completely satisfactory for execution on the production system, but the essential behavior of the component is defined. Compromises may be made in areas that have dependencies with the actual target system hardware, or the implementation may not be optimized for real-time usage. However, the entire set of components required by the production system will be present in the

model with the component interfaces completely defined, implemented, and tested.

The principal factor that makes this a viable approach is the use of a software development environment similar to those used in research on artificial intelligence systems.[5] Such research exploits interactive graphics, programming languages that provide for late binding of program elements, and a highly integrated set of development tools (debuggers, editors, compilers and interpreters) to build a very powerful but unstructured design environment. The goal of the environment is twofold: to remove as much of the syntactic overhead and resource administration duties as possible from the job of programming, and to minimize the time required to edit, compile, and test a program change. The combination of these attributes allows very rapid iteration of architectural changes to the system being built.

Using a network of special-purpose processors (Symbolics, Inc., Lisp machines) that support this environment, a small team of highly skilled designers can build and test functional models of large complex systems. The software components comprising the system and the intercomponent interfaces can be reworked many times until the system under design meets customer needs. The completed model provides the functionality required in the product and allows operation of the system to be verified by system users prior to its construction on a target system (the system developed during the construction phase and actually delivered to the field). If the existing system design is deficient or its operation suggests new requirements, a revised design can be produced quickly. Since the design environment supports high productivity for changing and testing the design, the design team can build alternative designs to evaluate and improve the system in an evolutionary manner. Because the product is developed by a small group of systems engineers and designers, it should exhibit a high degree of conceptual integrity. Modifications to the system design to meet customer requirements are made at the beginning of the design process, before a large amount of staff effort has been committed to the implementation of the system. When the system design has been agreed on and the model implementation has been thoroughly tested, the software components can be ported or constructed for execution on the target system.

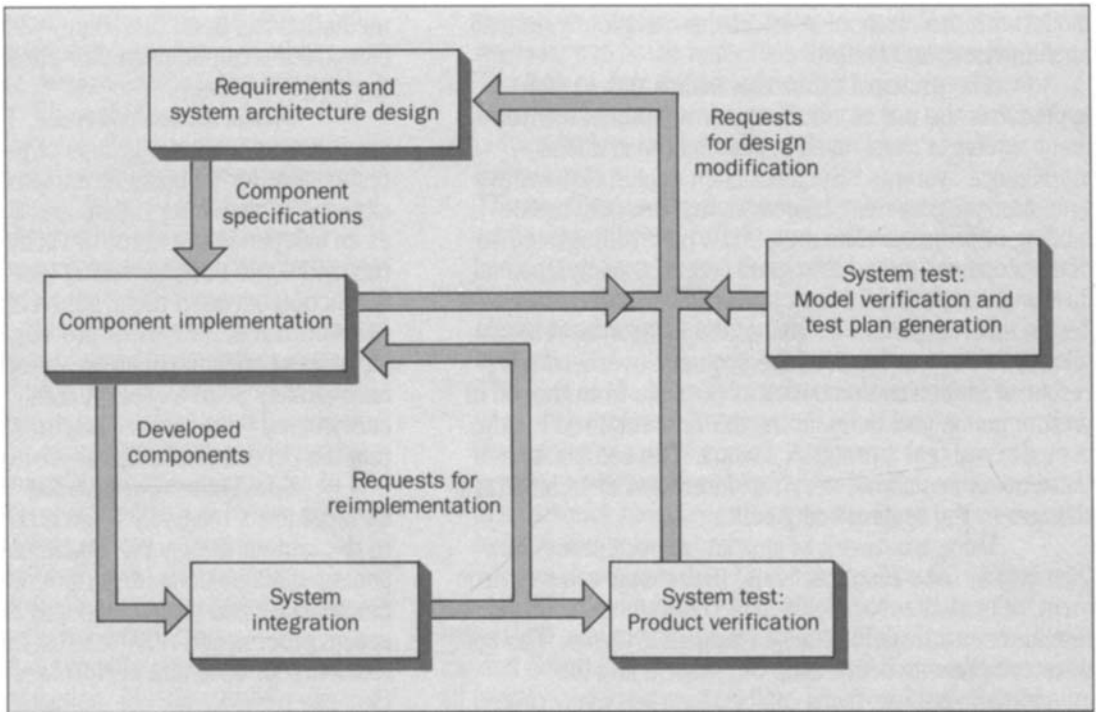
System Construction Phase. The construction process involves three distinct groups of personnel. One group is responsible for component implementation, one is responsible for system integration, and a system test group acts as an independent auditor to verify the operation of both the model and target systems (see Figure 4). The primary distinction between the system construction phase and the conventional development process is that, in the former, component specifications are complete and consistent and have already been tested. A highly parallel system can be constructed with far less synchronization and communication than in the conventional process.

Component Implementation. The component implementors are a relatively large group of people, in contrast to the system designers. The implementors take the component specifications (descriptions of component functionality and intercomponent interfaces) from the design process and reimplement the component internal elements for optimum performance on the target system. Because components are syntactically independent and the interface specifications have been completed in the design process, a large staff of implementors can work efficiently in parallel on component construction. The amount of communication required among implementors is dramatically reduced compared to that required with conventional design techniques.

The software development environment used for the construction process is more conventional. It is designed to impose structure on the implementation process and is optimized to provide support for project management, software version management, and source code control that allows development of a target system to be scheduled and tracked.

System Integration. The integration team develops the plan for staging component construction, building up the target system through component integration, and monitoring its performance. New or modified components that disrupt target system operation are removed or restored to their earlier version until their implementation

Figure 4. System construction process.



flaws are corrected. At any point in time, the components integrated in the target constitute a version of the system generic. When the functionality and performance of the system meet system requirements, the set of new and modified components can be transmitted to the field for a generic software release.

System test. The staff assigned to test systems consists of a limited number of highly experienced individuals who are skilled in evaluating the operational characteristics of large real-time systems. They evaluate a system's real-time performance, availability characteristics, and adherence to the application requirements. The system test team also develops plans for system testing, plans the staging of components for delivery for system testing, and manages the target vehicle configuration.

It may be possible to repair components that fail system test by reworking their internal implementation. If this does not resolve the problem, it is reported to the system architects for correction by a system redesign. The control of component specifications and their interfaces is always maintained by the architects.

Application-Oriented Languages

The description of the software development prototyping process has focused primarily on the software architecture modeling activity, the system construction activity, and how the two activities are related. This section concentrates on an application-level view of system functionality. Application-oriented language technology is an approach to specification and implementation of application-level services and features that can provide benefits in both architectural modeling and system construction activities. AOL implementations can also benefit from using component-oriented design, which is an essential part of the software development prototyping process.

Examples from Existing Systems. The languages described in this section predate our work on AOLs. A key contribution of our work was to recognize the characteris-

tics and benefits that these languages share and to generalize the concepts so that they may be more broadly applied. The examples described in this section demonstrate the power of AOL technology to enhance productivity.

Many tools used for text processing are essentially AOL translators. The *pic* language is a good AOL for describing pictures because it embodies concepts that are obvious and natural when pictures are being created. *Pic* suffers from one flaw that is common to many AOLs: *pic* is a textual language, and its expressive power is limited to what can be specified in text. In *pic*, a box is represented by the word "box" rather than a more natural sketch or graphical representation of a box. This defect is insignificant for a simple picture of a box, but becomes more significant for highly interconnected figures that are composed of many types of objects.

Many people now create complex *pic* pictures through a graphical front end known as *cip*. *Cip* is a graphical AOL for creating pictures. With *cip*, a box is represented as a box and a complex picture is represented as a complex picture. *Cip* embodies picture creation so well that you tend to forget that you are working in an AOL and begin to feel that you are actually drawing a picture when you use it.

Other text processing AOLs include the input language for the *grap* system, which is used to draw graphs in documents, and the input language for the *eqn* preprocessor, which is used to typeset equations. For example, the *eqn* input

```
a + b over 2c = 1
```

generates this equation:

$$a+\frac{b}{2c}=1$$

Common AOLs from outside the field of text processing are spreadsheet packages. These packages are designed to do financial analysis, although they have been used in a large variety of other applications. They embody the application paradigm of the accounting ledger sheet. Spreadsheet packages have been used to create sophisticated application software, applications that might never have been developed had it not been for the availability of the spreadsheet interface.

Common to these examples of AOLs is that the language processors do not compile into executable computer code as conventional programming languages do. The text processing systems produce code that drives printing devices. Spreadsheet systems are more like interpreted computer languages, but still are not programmable in the same sense that a general-purpose programming language is programmable. Spreadsheet packages typically lack generalized looping constructs, for example. Nevertheless, each of these examples represents a language that is sufficient (and extremely well-matched) for the application in question. Each example represents a complete application-oriented language.

Implementing AOLs. Specifications written in an AOL can be realized as executable code by creating two support programs:

- An *AOL translator* is a program in the software development environment that translates AOL programs into code in a lower-level language such as assembly code, C language, or another AOL.
- A *virtual machine* (VM) is a collection of software that supports AOL program execution and acts as the interface between the translated AOL program and the runtime environment in the target system.

AOL-based development consists of the following activities:

- Language design
- AOL translator construction
- Virtual machine construction
- Application programming.

These activities are best carried out in parallel, using rapid prototyping to converge on an easy-to-use language. Language design is the heart of the process, but it is difficult to get a sense of the completeness of an AOL without going through the mechanics of creating the AOL translator, the VM, and an application. There may be cases where the VM or the final application is not fully specified during the rapid prototyping phase, but it is generally possible to come up with close substitutes that can be used to

help refine the language.

Translator construction has a deserved reputation as a difficult and time-consuming process. However, tools such as Stage[6] take the sting out of translator development and make it possible to iterate the design of a translator to follow the evolving design of the language.

After the AOL has been designed and the AOL translator and VM have been built, the application programmer writes and tests AOL programs in one of two types of development environments. The first environment, shown in Figure 5, is an interpretive environment; the VM interprets some intermediate code that is created by the AOL translator. The second environment is a compiled environment, which is neither as flexible nor as easy to use as the interpreted environment, but is often chosen for reasons of execution efficiency in the final product. The compiled environment is shown in Figure 6. In the compiled environment, the virtual machine is incorporated into the final executable code either by the AOL translator or by the linker that binds separate sections of object code.

The choice between the interpreted or the compiled form of AOLs is entirely a matter of judgment. It frequently makes good sense to build both environments, using the interpreted environment as an application prototype debugging utility and the compiled environment as the production vehicle. When AOLs are being designed and evaluated as part of software architecture modeling (see the "Software Development Prototyping Process" section), it is very likely that an interpretive implementation would be used in the modeling environment while a compiled implementation would be used in the target environment.

AOL Design Principles. An AOL should incorporate the concepts that an experienced application designer uses when thinking about an application. By closely reflecting the design paradigm used by application experts, an AOL makes the expression of an application program as natural as possible.

The figure-drawing language *pic* is an example of this principle. *Pic* embodies common, well-understood ideas about figure drawing: objects such as boxes, circles, ellipses, lines, and arrows, and well-known directions such as north, south, up, and down. Figure 7 illustrates a simple

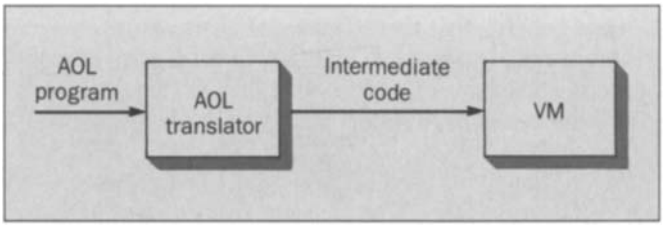


Figure 5. Interpretive AOL environment.

pic program and the figure it produces.

AOLs should hide as much as possible of the implementation details from the application designer. This allows the application designer to focus more on the application and less on coding. To maximize portability and reusability, the AOL programmer should be shielded from the following implementation details: selection of underlying implementation language, interfaces to underlying operating system, and messaging mechanisms.

Pic provides us with an example of this principle. *Pic* ultimately drives some device that can draw figures on paper, but details of how that device works are mercifully hidden from the user. Moreover, even though the *pic* translator generates output specifically for the *troff* text processing environment, it is, in principle, possible to build a new *pic* translator that generates output for other text processing environments, because nothing about the *pic* language presupposes an implementation. *Pic* does allow escapes to the underlying *troff* implementation language, in quoted strings. This is done to simplify the language.

Virtual Machine Principles. The VM for an AOL can be thought of as emulating a machine that executes AOL programs. It implements the application design paradigm and the primitive operations for the AOL. It also interfaces the application to the system's underlying hardware and software environment. The VM handles all implementation details that were hidden from the AOL programmer.

The VM for an AOL may be an interpreter for the AOL, but need not be. For example, *pic* does not have an interpreter. Instead, details of implementing picture drawing functions and interfacing to the underlying *troff* environment are embedded in the *pic* translator.

It is common practice to embed a VM in an AOL

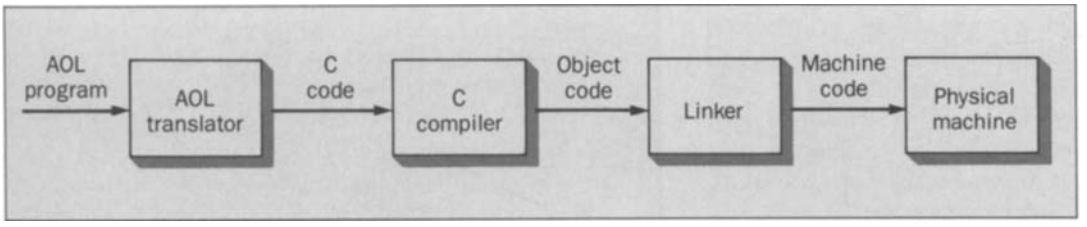


Figure 6. Compiled AOL environment.

translator. In such cases, it is tempting to drop the distinction between VM and AOL translator altogether. However, identification of the VM is important for software reuse; an AOL program can be reused without change in any environment where the VM is reimplemented. In consequence, major investments in application design and implementation can be preserved when it becomes necessary to move to a new hardware/software target environment. As illustrated in Figure 8, reimplementing the VM in a new target system environment allows a straightforward move of the application programs.

Applicability of AOLs. AOLs are not suitable for all applications. Much of our work has been directed at discovering the domain of problems for which AOLs are suitable. Our strategy has been to identify appropriate applications within switching systems to achieve improvements in switch software development productivity and quality by introducing AOLs. An early prototype[7] implemented call processing primitives for a finite-state-machine-based design paradigm. We were able to introduce new and customized call processing features into the running prototype in about 10 minutes, including specification, translation, downloading, and activation.

AOLs are suited to applications in which a customary procedural programming solution would be highly repetitive. These are often applications that are programmed with table-driven approaches. It can be extremely cumbersome to fill in the driver table for a table-driven program because of the restricted syntax for table initialization. This type of code can be extremely repetitive and error-prone. An AOL can help by providing a problem-oriented syntax that makes the tables more readable and maintainable. The AOL translator would automatically generate the repetitive code for filling in the table entries.

Table-driven approaches tend to tie the specification (the table) to a specific implementation (the driver). In the AOL approach, the specification of the problem, as AOL code, can be separated completely from the implementation of the solution in the VM. For one AOL, we built an AOL translator that supported three different but interchangeable VM implementations. There were three C language implementations: a version that was conventional code, a version that was table-driven code, and a version that was augmented with special-purpose debugging instructions. The developers simply chose the implementation that looked best for the task at hand. This approach allowed the application designers to defer some key implementation decisions.

The AOL concept is an ideal way to provide sophisticated programming/user interfaces to a turnkey application. The *pic* example is a classic case: the AOL was created precisely so that people who understand the application (those who create figures) could get figures without understanding the labyrinthine details of the underlying *troff* text processing system.

Benefits

Software architecture modeling and application-oriented languages provide significant support for system specification. Both technologies use rapid prototyping and an iterative design approach to produce working models of system features and services quickly. By reviewing these early models with systems engineers, and potentially with customers, we can establish a high degree of confidence that the product system will meet real customer needs before extensive resources are committed to development. As a result, these technologies provide major benefits in the areas of quality and productivity, helping to ensure that the *right thing* is being built.

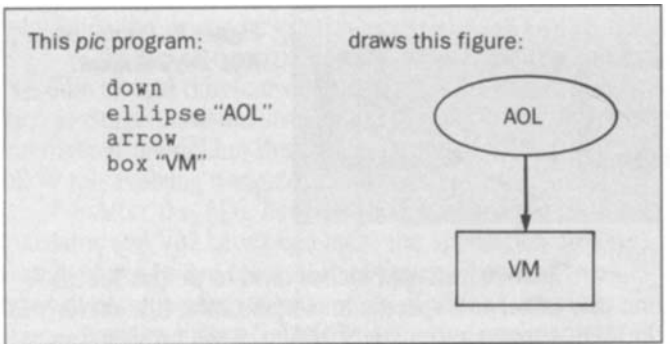


Figure 7. *Pic* program and resulting figure.

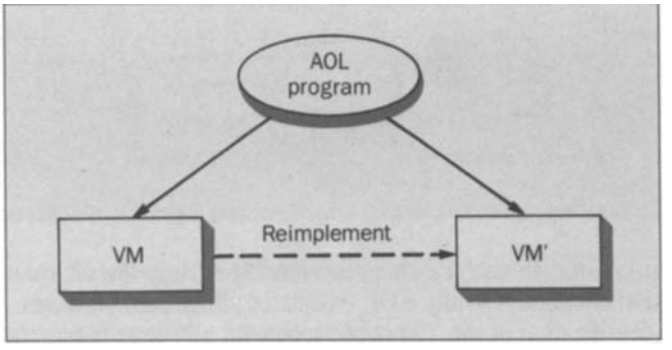


Figure 8. AOL portability and reuse.

Software architecture modeling uses powerful, interactive workstations and a software development environment, adapted from artificial intelligence applications, that supports rapid creation, modification, and validation of designs. Software architecture modeling also uses component-oriented software design to ensure that system software consists of highly independent, loosely coupled, interchangeable piece parts.

Component-oriented design itself provides numerous benefits in the production target system. The primary benefit is that changes in a component can be accomplished and introduced into the system (model or target) rapidly without significant changes in other system components. The elimination of syntactic dependencies between components results in less information being available to support error detection at compilation time. However, extensive testing and debugging capabilities supported in the design environment compensate for this lack.

Application-oriented languages support direct implementation or modification of application features and services by application designers who do not have to be programming experts. By identifying the design paradigm that is natural for a particular application and embedding it in a virtual machine, designers are able to define services and features with an application-oriented language that is, in effect, a specification language for that particular application. By reimplementing the virtual machine in a new product/hardware/software environment, application features and services can be ported readily to new environments. This ensures complete compatibility for the same service operating in several different products. It also ensures a high degree of productivity and quality because software reuse is maximized and existing test suites, applied in the original product environment, can be reapplied in the new product environment to validate that performance and functionality have been preserved.

Application-oriented languages can make very effective use of component-oriented design, one of the technologies that supports software architecture modeling. While actual applications are built by using the application-oriented language, the underlying virtual machine that executes the language is built by using component-oriented design. This approach makes it easier to reimplement the virtual machine for new products and environments, because its constituent components can be reimplemented in parallel without having to be redesigned. This approach also makes it possible to use application-oriented languages as an integral part of software architecture modeling. The virtual machine can be modeled with the rest of the system. Services and features can be built and validated in the modeling environment and then ported to the desired target system as soon as the virtual machine has been reimplemented for the target system.

References

1. G. Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986.
2. J. O. Coplien, "The Object Paradigm as a Future Life Cycle

Method," *Proceedings of NCF/86*, National Communications Forum, Chicago, November 1986, pp. 1110-1115.
3. B. Stroustrup, "What Is 'Object-Oriented Programming,' " *Proceedings of the European Conference on Object-Oriented Programming*, Paris, June 1987.
4. J. O. Coplien, S. C. Dewhurst, and A. R. Koenig, "C++: Evolving toward a More Powerful Language," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 19-32.
5. B. A. Sheil, "Power Tools for Programmers," *Interactive Programming Environments*, Barstow et al., eds., McGraw-Hill, New York, 1984.
6. 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.
7. T. L. Hansen et al., "A Nonprocedural Language for Telecommunication Call Processing Applications," *Proceedings, IEEE Workshop on Languages for Automation*, Singapore, August 1986.

Biographies (continued)

burgh and an M.S. in computer sciences from Purdue University. Mr. Montgomery is supervisor of the Software Infrastructure Group in the Advanced Switching Networks Department. He explores new software technologies for building switching systems. He joined AT&T in 1978. He received an A.B. in mathematics and engineering from Dartmouth College and M.S., Engineer's, and Ph.D. degrees from the Massachusetts Institute of Technology, all in electrical engineering and computer science. Mr. Zislis is head of the Advanced Software Technology Department, which explores advanced software architecture, development environments, process modeling, specifications support, and expert systems. He joined AT&T in 1969. He received a B.S. degree in mathematics and computer science from the University of Illinois, an S.M. degree in information sciences from the University of Chicago, and a Ph.D. degree in computer sciences from Purdue University.

(Manuscript received April 26, 1988)