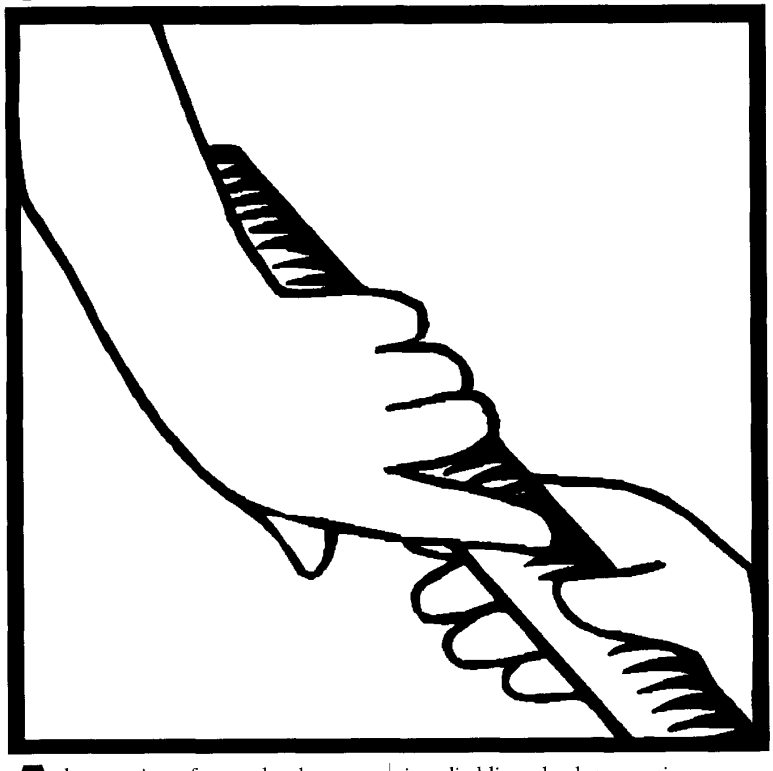
FEATURE

ON-THE-FLY PROGRAM MODIFICATION: SYSTEMS FOR DYNAMIC UPDATING

The rising cost of shutting down systems for maintenance and repair is forcing developers to look at ways to repair software as it runs. The authors briefly describe available updating systems and present a prototype.

MARK E. SEGAL
Bellcore
OPHIR FRIEDER
George Mason University



Advances in software-development technology and methods continue to help engineers build larger and more complex systems, but those systems are neither error-free nor can they satisfy every anticipated need. Despite all our advances, software must still be changed to repair bugs and add new functions — often resulting in downtime. As users grow to depend on a system, they become more and more intolerant of these interruptions.

For some companies, the cost of system shutdown can be prohibitive. Changing the software that controls an orbiting spacecraft, for example, cannot be done at all if it means disabling the life-support system. And, although not life-threatening, disabling a bank-transaction processing system may have significant economic consequences — particularly if the companies involved have a reputation for providing a highly available service. In the telecommunications domain, switching systems have a maximum downtime requirement of less than two hours within 40 years!

If the software being changed is part of a distributed system — in which many programs interact over geographically distributed networks — these problems become even more acute. Not only must you address the problems associated with downtime, but you must also coordinate the shutdown. If one computer begins running the new software while the others

0740-7459/93/0300/0053/$03.00 © IEEE

continue to run older versions, they might exchange incompatible data.

Clearly, there is a need for novel maintenance approaches that do not interrupt system operation for long periods.

One such approach is a system that updates or replaces a program version without stopping the current one. A *dynamic program-updating system* can make it easy to repair bugs or enhance running software without the cost of system shutdown.

THE MORE TRANSPARENT AN UPDATING SYSTEM, THE MORE PEOPLE WILL USE IT.

In this article, we briefly describe a number of research and production updating systems and present a prototype system developed at the University of Michigan and enhanced at Bellcore.

GENERAL CHARACTERISTICS

The techniques a system uses for dynamic updating are influenced by the application domain and the desired performance and correctness guarantees. Thus, a system for dynamically updating an information server used in a time-sharing environment, for example, would generally not be appropriate for updating a real-time process-control program. There are, however, several characteristics that all updating systems should possess, regardless of their intended use. The relative importance of these characteristics varies with the application domain.

All these characteristics work to make the system as transparent as possible to both its users and programmers and its execution environment. The more transparent an updating system, the more likely programmers and managers are to use it.

In general, all updating systems should

♦ *Preserve program correctness.* Program correctness must be preserved during the update as well as at times when no updates are in progress. Mechanisms for preserving correctness include tools for discovering program state to prevent updating at incorrect times and tools for atomically replacing program components (replacing a group of procedures without interruption). Some updating systems go further by providing policies for determining when program components can be updated. An update policy could be something as simple as updating a program module (a group of procedures and their associated data) only if it has no outstanding requests, or as complicated as updating a module if none of the data it references is being accessed by another module.

This need to preserve program correctness is essential because it lays the foundation for other characteristics such as minimizing human intervention. If policies to preserve program correctness are in place, programs can be updated correctly with little human assistance; if these policies are not in place, the burden of determining when to update a program component rests on those who control the system.

You can also extend program correctness to account for time by arguing that if the system causes a program to take significantly more time to produce results, it is affecting program correctness. Real-time systems are a good example. If updating makes a real-time program execute longer, you are violating the program-correctness criterion. Unfortunately, most updating systems do degrade program performance during an update, and anyone designing an updating system should try to minimize this effect. Similarly, designers of updating systems and the related parts of the computer's runtime system should ensure that the updating system does not degrade program performance when no updates are in progress.

♦ *Minimize human intervention.* Part of preserving program correctness during an update means ensuring that the updating components are applied in the correct order and at the right time. Even a meticulous person can perform an update improperly. There must also, of course, be ways to override default updating sequences if conditions warrant it.

♦ *Support low-level program changes.* To dynamically update a range of programs, an updating system must support a variety of low-level program changes. The simplest kind of change is to replace a module with a new one that is implemented differently. In this scenario, the module's interface (its calling conventions) remains the same, and the module retains no internal state between invocations. More complicated changes include changing the module's interface, having the module retain state between invocations, changing the state's implementation, and changing the implementations of both the interface and state variables. Changing the implementation of state variables often occurs in programs that implement data structures as abstract data types.

♦ *Support code restructuring.* Significant code restructuring can occur during maintenance — a change beyond simple module replacement. An updating system must be able to update programs when new modules are added, existing modules are deleted, and functionality is moved between modules, for example.

♦ *Update distributed programs.* Many programs that benefit from dynamic updating are distributed by nature. Future distributed programs will consist of collections of modules running on a variety of heterogeneous hardware. These programs will communicate and cooperate across mutually distrustful administrative domains. Administrative domains are networks of computers each controlled by a separate organization. From a security standpoint, these networks are mutually distrustful because one organization cannot make arbitrary changes to another's computers. The updating system and the programs being updated must cope with the reliability problems such a large-scale network presents.

The updating system's algorithms must scale to large distributed programs, in which there are hundreds of thousands of individual modules, and cooperate with other updating systems across administrative domains.

♦ *Not require special-purpose hardware.* They can, of course, exploit it if it is there already, but such hardware usually in-

creases system costs and can decrease portability. Adding additional hardware to a large distributed system is even more expensive.

♦ *Not constrain the language and environment.* The user must be free to choose a language and system environment. An updating system must not force programmers to write code or call operating-system primitives in a radically different manner. Doing so would prevent many large programs already in use from benefitting from an updating system. Ideally, updating systems should tolerate a variety of programming styles — or at least require a style that is already widely used — and allow substantial code reorganization between versions of the program being updated.

HARDWARE-BASED SOLUTIONS

Before looking at software-based updating systems, it is worth briefly examining hardware-based solutions, although they are costly and have a narrow application. In a system that uses hardware-based dynamic updating, an entire running program is dynamically updated with a second system identical to the one executing the program. Because the second computer system with the new version of the program is loaded while the first computer continues to execute the older version, programs can be updated with minimal downtime and maximum flexibility in restructuring.

To perform the update, you stop the first computer at a safe point in the program and simultaneously start the second. Some work in progress may be lost during the update, but you will need only a short time to complete this type of update in most cases.

The principal disadvantage of this technique is its substantial cost. It is typically used in systems that need redundant hardware anyway to provide fault-tolerance — telecommunications systems for example. However, even then, building a redundant computer system and properly connecting it with the main computer system is both difficult and expensive. Not only must the hardware be interconnected, but the software shared between the systems (such as databases) must be kept consistent. Moreover, such an approach, which requires close synchronization between systems, does not scale to a distributed environment. For these reasons, our focus in this article is on software-based systems.[1]

A slight variation of this dynamic-updating technique is in Bellcore's Service Control Point, which provides facilities for high-speed 800-number lookup and calling-card verification for local telephone companies. The SCP must always be available to service both kinds of requests. To achieve the desired performance, fault-tolerance, and availability goals, each SCP is constructed with redundant computer and communication hardware. A copy of the SCP software runs on each set of redundant hardware and the communications subsystem routes requests to available processing hardware.

In a dynamic update, the user reconfigures the SCP to reroute all requests to other programs that perform the same function, and then replaces the program(s) that must be changed. After the programs are updated, the SCP routes all requests to the newly replaced programs. The programs that handled the requests during the initial update are then replaced. If the changes involve modifications to the operating-system interface, the user must physically partition the SCP into two systems during the update. In this worst-case scenario, the update takes approximately two weeks to plan and eight hours to complete. Communications subsystem software is updated using redundant hardware, redundant intersite communication links, and rerouting algorithms similar to ones just described.

ANY PROGRAM CAN BE SO POORLY WRITTEN THAT IT CANNOT BE DYNAMICALLY UPDATED.

SOFTWARE-BASED SYSTEMS

Software-based updating systems are characterized by several structural, behavioral, and performance metrics. We distinguish systems by the granularity of the program components being updated and the way they communicate. The box on pp. 58-59 describes these metrics and gives a quick-reference comparison table of the systems we describe.

Unfortunately, we have found no single system (nor do we believe one exists) that lets you incorporate all possible changes into any program structure. In other words, any program can be so poorly written that it cannot be dynamically updated. Perhaps this is why those researching dynamic updating have concentrated on creating dynamic updating techniques for specific, well-accepted, and well-understood program structures.

Fully automatic dynamic updating, which requires no human intervention beyond starting the update, has the potential to update many program structures. But as of yet, it cannot work properly if semantic information is needed to perform any aspect of the updating. Humans are error-prone, and if a human must perform nontrivial program updates, the scope of dynamic updating will be limited by that person's capabilities. Perhaps research in program verification and understanding may reduce this limitation.

The types of software-based systems include replacement of abstract data types in programs, replacement of servers in client-server systems, updating of distributed programs that use externally specified communication topologies, and the updating of programs in procedural languages.

Replacing abstract data types. The first type of system allows replacement of abstract data types in programs. In this category are the dynamic-type-replacement approach and the DAS operating system.

Dynamic type replacement. Dynamic type replacement involves changing the implementation of user-defined abstract data types, which is described in a collection of procedures called the type manager.

When the type manager is changed, the old and new data-representation formats are different, and previous instantiations of this type will not work correctly with the new type manager. To handle this situation, programmers must build a version-tagging and data-conversion mechanism.

Robert Fabry used version numbers to tag each object of a specific type.[1] If an object is an old version, the type manager calls a routine to convert the old representation to the new representation.

Although the dynamic type-replacement approach provides a reasonable method of updating abstract data types, it fails to address the more general issue of changing code structure as well as implementation. For instance, changing the calling conventions (interface) of a type manager (other than merely augmenting them) is not supported. Also, Fabry's approach requires a capability-based addressing scheme, thus limiting its applicability to capability-based systems.

USING ARCHITECTURE BASED ON VIRTUAL MEMORY MAKES UPDATING TOO COMPLEX.

DAS operating system. Hannes Goullon and colleagues constructed an experimental operating system, the Dynamically Alterable System, for teaching and research at the Technische Universität Berlin.[2] DAS provides support for dynamic updating of application programs by letting a module be replaced with a new module that has the same interface.

Conceptually, DAS is similar to dynamic-type-replacement systems. It performs dynamic updating using "replugging," a mechanism built on DAS's address-space management system, which is, in turn, built on the addressing hardware of DEC's PDP 11/40E. When a procedure in a different module is called, DAS performs an address-space transition by placing the descriptor table entries of the new module into the address-mapping hardware, while saving the current entries on a stack. Only one code segment is kept in the process's virtual address space (to compensate for the PDP 11's small virtual-address space). The information used to keep track of each module is stored in a linked list of descriptors called a descriptor chain. Replugging is done by changing the links within the descriptor chain.

DAS also supports the restructuring of data stored within the modules. Restructuring is performed in conjunction with replugging.

One significant problem with DAS is its use of virtual memory to aid updating. The virtual-memory architecture — consisting of the descriptor chains and the mechanism for address-space transition — is both overly complex and inefficient. The large, sparse address spaces available on current CPUs make the architecture somewhat obsolete. To implement it with acceptable performance on the PDP 11, the DAS designers had to code portions of it in microcode and use previously unused opcodes to reference the microcode routines.

Another problem is that DAS fails to provide mechanisms for procedures whose interfaces change between versions.

Replacing servers in client-server systems. A number of dynamic updating systems have been constructed for systems that observe a client-server relationship, in which one or more servers supply services to a set of clients. The clients and servers may or may not run on physically distributed computers.

Software for this paradigm has a structure similar to that of programs for dynamic type replacement. In each case, parts of the program invoke a set of services or abstract-data-type managers.

The two approaches differ in the granularity of the objects to be updated: an abstract data type is smaller than a service. A file server provides access to files, which may be built from a number of abstract data types that manage lists of free disk blocks, access rights, open files, and so on. Thus, a service may be constructed from several abstract data types. When an abstract data type in a dynamic-type-replacement system is updated, only the representation of the type and the code that manipulates it change. When a server is updated, however, any of the code within the server may change.

Systems in this category include the Michigan Terminal System and Argus.

Michigan Terminal System. MTS, a time-sharing system for IBM-compatible mainframes, can dynamically update large system services, such as command-language interpreters and mail systems. Programs requesting services call the servers indirectly through a service manager. By modifying the service manager to call a new version of the service, you can change the implementation of a service at runtime. Programs running an old version of the service continue to execute it while new requests invoke the updated version.

This approach assumes that only the implementation of a service changes between versions, but not its interface. It also assumes that programs are explicitly written to call service managers instead of services.

MTS has several disadvantages. Because it is designed primarily for large components of an operating system, as opposed to end-user software, it does not explicitly address the problem of a new service calling an old service. If such a dependency exists, MTS deals with it by updating all dependent services simultaneously — an action that requires disabling the services to users for the duration of an update. Also, because MTS is fully reentrant, data used by these services is stored in special data areas, so you cannot change the data format between versions.

Argus. Toby Bloom describes a dynamic updating system[3] in the context of the Argus distributed programming system developed by Barbara Liskov.[4] Argus is a language based on Clu and an underlying operating system. In Argus, a program consists of a collection of servers called guardians that implement a logical set of

functions through a set of handlers. Because a guardian's internal state is not accessible to other guardians except through the use of handlers, a guardian is similar to the abstract data type described earlier except it is larger. Guardians can manage persistent data and function in a distributed environment. All communication among them is handled through a message-based communications system.

To dynamically update a program in Argus, the user replaces related collections of guardians, called subsystems. Argus's communications infrastructure and its ability to maintain consistent versions of persistent data when the system crashes enable guardians to be temporarily inaccessible during an update. Because Argus lets a guardian's persistent data be associated with different guardians over time, it can manipulate persistent data or transfer it to new versions of guardians during an update.

Much of Bloom's work addresses keeping data consistent and ensuring that an update does not make subsequent guardian interactions inconsistent. This is possible, in part, because Argus lets guardians crash and restart in a consistent way and expects clients to tolerate this occurrence.

The disadvantage of this system is that it is tightly tied to the Argus system, so adapting it to other systems could be difficult. Crash-recovery facilities are necessary for the Argus updating approach to work properly, yet few operating systems, runtime systems, or languages contain the necessary support code for crash recovery. Furthermore, Bloom acknowledges that the subsystem, which is the atomic component of updating in Argus, may be too large for some applications. If a small part of a subsystem is changed, the entire subsystem is unavailable during the update.

Updating in constrained message-passing systems. The third type of software-based updating system is designed for distributed programs that communicate using externally specified communication topologies.

Conic — developed by Jeff Magee, Jeff Kramer, Morris Sloman, and colleagues — is a system in this category.[5] Like Argus, Conic provides a language and a runtime environment for constructing distributed programs. Conic programs are constructed as a set of task modules. Each task module can have a number of entry and exit ports. Task modules communicate using these ports, which are unidirectional, typed communication channels.

One of Conic's key design goals is to totally separate module programming from configuration management. For this reason, a task module's ports do not directly refer to any other task modules. Instead, Conic provides the Configuration Manager, which lets you build a distributed program by specifying a set of task modules and their port interconnections. Module interconnections may be within the boundaries of a single physical computer or may cross a network. This separation makes it easy to reconfigure task-module interconnections because you do not have to modify the source code.

In a dynamic update, the Configuration Manager updates a task module M by removing the links to M's ports, setting up links to the new version of M's ports, and copying state information from the old version of M to the new version.

It relinks and copies state when M is in an inactive state. Because a task module cannot ascertain which other task modules it is connected to, the update is transparent to all task modules. Similarly, because the Configuration Manager cannot see the implementation of task modules, it may arbitrarily restructure them internally during an update.

Although Conic provides a powerful and versatile environment for dynamic updating, it has two potential problems. First, as with Argus, you have only one choice of language and runtime system, and Conic is not a well-known language. Second, Conic does not impose a client-server relationship on a distributed program, but it does force a module to communicate through a fixed set of ports to a fixed set of modules. This structure precludes a module from selecting where to send a request at runtime. Unless interconnections are created for all task modules that may wish to communicate, implementing programs such as mail systems and remote log-in facilities would be extremely difficult.

Updating programs in procedural languages. The third type of updating system is procedure-oriented, for programs written in procedural languages like Pascal and C. In such programs, a natural unit of replacement is the procedure or function. Because many programs are written in procedural languages, there is strong motivation for investigating dynamic updating in this environment. A distant relative of procedure-oriented updating, though not suitable for dynamic updating, is dynamic linking, which is described in the box on p. 60.

The granularity of procedure-oriented dynamic updating differs from that of other software-based updating techniques. The servers in client-server systems are typically built from many procedures. When a server is updated, all procedures constituting a server are replaced as a unit. Even if only a small portion of the server has changed, the entire server is unavailable during the update. In many cases, the granularity of type managers is also larger than an individual procedure. Dynamic type replacement emphasizes the internal restructuring of abstract data types; whereas, procedure-oriented updating emphasizes more general code restructuring.

PROCEDURE-ORIENTED UPDATING EMPHASIZES GENERAL CODE RESTRUCTURING.

Several research and commercial systems use procedure-oriented dynamic updating, including the DMERT operating system, Dymos, and PODUS.

DMERT operating system. AT&T's 3B20D processor, a part of the Number 5 Electronic Switching System, runs the Duplex Multiple Environment Real-Time operating system. The Field Update Subsys-

SELECTING A SUITABLE UPDATING SYSTEM

After deciding that your application domain requires dynamic program updating, the next logical step is determining which system would best fill your needs.

A number of sources of information about specific dynamic updating systems are available. The *Bell System Technical Journal* describes the internal workings, including updating issues, of all AT&T's electronic switching systems. Companies such as Stratus and Tandem sell computer systems that use redundant hardware to provide fault-tolerance and high availability.

Papers describing dynamic program updating systems research may be found in the proceedings of the IEEE Conference on Software Maintenance, the IEEE International Conference on Distributed Computer Systems, and the IEE International Workshop on Configurable Distributed Systems, for example.

Table A is a list of the updating techniques and systems we have described and their associated features. It and the references at the end of the article should help point you in the right direction.

The table rows consist of the metrics or attributes we used to evaluate the systems. Entries and their explanations are (rows like System Status are omitted because they are self-explanatory):

- *Type of updating system.* The system types correspond to the classifications used in the main article. Examples are procedure, client-server, module, abstract data type, and hardware-based.
- *System-support requirements.* This is the support that the development and underlying runtime environment must provide to make the updating

TABLE A
COMPARISON OF DYNAMIC UPDATING SYSTEMS

Feature	Argus	Conic	DAS	DMERT
Creator	MIT	Imperial College	Technische Universität, Berlin	AT&T
Type of updating system	Client-server	Module-based	Procedure-based	Procedure-based
Status	Research project	Research project/ product	Research project	Commercial product
Hardware requirements	None	None	PDP-11/40E	None
System-support requirements	Argus runtime environment	Conic runtime environment	DAS operating system	DMERT operating system
Language requirements	Argus	Conic	Procedural	C
Distributed?	Yes	Yes	No	No
Distributed interprocess communication	Remote procedure calls	Message passing	—	—
Granularity	Whole server	Task module	Procedure	Procedure
Changes supported	Guardian (service) implementation	Task-module implementation	Procedure implementation	Procedure implementation
Degree of human intervention	Unclear	Moderate	Unclear	Moderate
Update time	Short to moderate	Short	Unclear (presumed short)	Short to moderate
Main limitations	Works only with Argus	Special-purpose language; interprocess communication requires explicit intermodule links	No support for complex changes; requires PDP-11	No support for complex changes

**Segmented virtual memory speeds up.*

†*Segmented virtual memory improves performance.*

system work properly. The less support required, the easier it is to adapt an updating system to a particular platform. Examples are redundant hardware, capability-based addressing, and choice of language and operating system.

♦ *Distributed interprocess communication.* If the updating system can update distributed programs, the type of interprocess commuication used affects the kinds of programs that can be updated. Examples include no communication, remote procedure call, and message passing.

♦ *Granularity.* Granularity is the unit of what the updating system replaces. The finer the granularity (smaller the unit), the more easily and quickly the system can update programs with small, localized changes. Units include procedure, module, abstract data type, and whole program.

♦ *Changes supported.* The types of changes supported depend on the system's granularity. Systems that support fine-grained program changes let you change a procedure or an abstract data type, including its local data. These types of updating systems are best for making small, localized program changes.

Updating systems that support course-grained changes update large pieces of programs; they are better for making sweeping changes to major program components. The extreme example of course-grained change is replacing an entire program as a single unit with new code; this lets you do arbitrary code restructuring. The types of changes supported depend on the system's granularity. Fine-grained systems support changing the implementation of a procedure or an abstract data type, including its local data; coarse-grained systems are better at changing aprocedure's or an abstract data type's specification.

Dymos	Dynamic linking	Dynamic type replacement	MTS	PODUS	SCP
Insup Lee	Multiple	Robert Fabry	University of Michigan	Mark Segal and Ophir Frieder	Bellcore
Procedure-based	Procedure-based	Abstract-data-type-based	Client-server	Procedure-based	Hardware-based
Research project	Commercial product	Research project	Research project/product	Research project	Commercial product
None	None	Capability-based addressing	IBM 370-compatible mainframe	None*	Redundant CPUs
Dymos runtime environment	Implemented in operating system	Capability-based operating system	MTS operating system	None†	VMS operating system
StarMod	None	Procedural	None	Procedural	None
No (but multithreaded)	—	No	No	Yes	No
—	—	—	—	Remote procedure call	—
Procedure	Procedure; library	Abstract data type	Whole program	Procedure	Whole program
Procedure specification and implementation	Procedure implementation	Abstract-data-type implementation	Arbitrary code restructuring	Procedure implementation	Arbitrary code restructuring
Moderate	Low	Unclear	Moderate	Low to moderate	Extremely high
Moderate	Short	Unclear (presumed short)	Short	Moderate	Long
Requires fully integrated system	No support for complex changes; no way to preserve correctness	Can't change code structure; needs capability-based addressing	Can't map state/calling sequences across changes	Requires top-down structured programs	Needs fully redundant hardware

WHY NOT DYNAMIC-LINKING SYSTEMS?

A somewhat distant relative to procedure-oriented systems is the concept of dynamic linking, or late binding. Available in many commercial operating systems, it lets you link references to an external procedure (usually part of the operating system or a library) to the external procedure itself when the program is run or when the external procedure is first referenced during the run.

Programs can thus invoke the most recent version of operating-system or library procedures without having to relink each time a new version of the library is made available. In this regard, it is similar to dynamic program updating since replacing a dynamically linked library essentially updates part of a program.

Examples of operating systems that provide such a capability are Multics, HP/Apollo's Aegis, Sun Microsystems' SunOS 4.0, and Microsoft's OS/2.

Dynamic linking does not provide true dynamic updating capabilities because it does not let you change references to an external procedure during a run after they have been established. Even if an external reference is resolved every time the external procedure is invoked, dynamic linking would not be an updating system because it still lacks correctness-preserving mechanisms. For example, a library could be replaced while a program is executing a procedure inside the library, which could lead to undefined program behavior.

Dynamic linking systems also do not have mechanisms for mapping internal state across library versions.

tem of DMERT supports the updating of the C functions that make up the switching software running on the 3B20D.[6] The Field Update Subsystem is used primarily to install emergency patches into switching software.

The DMERT operating system supports dynamic updating in the following manner. Each DMERT process contains a transfer vector that provides a level of indirection between a function call and the actual address of the function in memory. By changing the address for a particular function in the transfer vector, all future references to that function are routed to the new version of the function. The Field Update Subsystem provides automated mechanisms that cause this change to take place (and to back it out if need be) as well as update the disk-based program images and logs of the programs running on the 3B20D.

The DMERT operating system assumes that the interfaces of the functions do not change between versions, and does not provide a mechanism for moving static state information between versions of functions. Despite these limitations, it is an example of a software system that can dynamically update programs in an application domain that is relatively intolerant of system downtime.

Dynamic Modification System. The Dynamic Modification System, developed by Insup Lee,[7] provides a fully integrated environment for software development and program updating for programs written in StarMod — a concurrent language similar to Modula. The Dymos environment contains a command interpreter, a source-code management system, a StarMod compiler, a file editor, and a runtime environment. In Dymos, a program is updated by replacing individual procedures. Because StarMod supports the concepts of modules and abstract data types directly, you can update a program at the module level. (Although languages like Pascal and C do not support these concepts directly, you can implement modules and abstract data types in them.)

Dymos also provides mechanisms that let you change a procedure's interface between versions as well as implement static data local to a procedure.

Because Dymos is integrated, the source code, object code, and compilation artifacts (like parse trees and symbol tables) of the program being executed or updated are available at all times. Dymos uses this program information to aid in software development and dynamic updating.

To update a program in Dymos, you must explicitly inform the system which procedures to update and under what set of conditions. The conditions for updating a procedure consist of waiting for a specified set of procedures or modules to become idle (not executed by any process).

For example, the Dymos command

```
update A when X, Y, Z idle
```

tells the updating system to update procedure *A* when procedures *X*, *Y*, and *Z* are idle. Dymos uses this command, the compilation artifacts, and the state of the running program to determine when it is safe to update procedure *A*.

Although Dymos's general architecture is similar to that of several procedure-oriented updating systems, it fails to provide some of the general characteristics described earlier.

- *It is language specific.* StarMod is not widely used, and its syntax was modified to support dynamic program updating. Altering a language's syntax to support dynamic updating (or anything else) invariably leads to portability problems.
- *It is fully integrated.* Dymos's full integration causes two problems. First, the tools available under the host operating system must either be modified (assuming the source code is available) to manipulate the compilation artifacts that Dymos maintains, or simply not be used. This becomes a problem if, for example, the Dymos editor has a different user interface than the editor the user is accustomed to.

Second, Dymos implicitly assumes that source code is available — a possibly invalid assumption when the software provider and the user are different. For example, a company providing proprietary software to a customer will, in general, not want to give the customer the source code. If the software must be dynamically updated and the source code is not available, you cannot use Dymos.

♦ *Each procedure must be updated explicitly.* Although the language constructs supplied for updating are quite expressive, an inexperienced user could update the wrong procedure at the wrong time by typing an incorrect update command. In some cases, you can even deadlock Dymos.

♦ *No support for distributed programs.* Because Dymos supports only multithreaded programs, it may not be as effective for updating programs written in other styles, such as single-threaded or distributed. For example, whenever a procedure is invoked, a locking protocol must be executed to control access to the procedure during an update. A performance penalty is paid for executing this protocol regardless of whether an update is in progress. If Dymos did not support multithreaded programs, this protocol could be made much simpler or even eliminated. More important, the locking protocol does not scale to a distributed system.

PODUS. In the Procedure-Oriented Dynamic Updating System, developed at the University of Michigan and later enhanced at Bellcore,[8,9] a program is updated by loading the new version of the program and replacing each old procedure with its corresponding new procedure during execution. Updating a procedure involves changing the binding from its current version to the new version. When all procedures have been replaced by their corresponding new versions, the program update is complete.

As a program executes, users request that newer versions be loaded into other sections of memory. The loads are performed without affecting the execution of the current version. Once a new version has been loaded, the user initiates the update by invoking an update command. The updating system interrupts the program and examines the current state of its runtime stack. Using this information and the list of all procedures that each procedure can call (generated by the compiler), the updating system calculates when each procedure may be updated.

PODUS imposes two requirements on the structure of the programs to be updated. First, programs must be written in a top-down manner. If the overall structure is top down, the high-level logic is specified at the top levels of the call graph, and the program's implementation is specified at the lower levels. Typically, the lower level implementation of program logic changes more often than the higher level, so there is generally less work during an update because less code has changed. The lower level procedures become inactive sooner, causing the update to complete sooner. Second, data accessed by several different procedures (such as global variables) is accessed through abstract data types. Thus, procedures are updated only when they are inactive. Badly written programs, such as those that consist of one large main procedure, cannot be updated using PODUS because the procedure would be active until it exits. By definition, dynamic updating systems update a *running* program, so there would never be a time to update such a program.

WE HAVE USED THE PODUS PROTOTYPE TO EVALUATE PROGRAMS OF VARYING COMPLEXITY.

Using the binding architecture, interprocedures, and mapper procedures, PODUS lets you update a program while preserving the program's interface and internal state. An interprocedure, shown in Figure 1, is a user-specified routine that converts the procedure's interface. This ensures that procedures are not called with the wrong number or type of parameters. If an old procedure attempts to invoke a new procedure, the corresponding interprocedure is automatically invoked. Interprocedures invoke only procedures that are in their new updated version and exit to procedures that have yet to be updated. Mapper procedures, or mprocedures, perform a conceptually similar task by converting the static data used by a procedure into a format suitable for the new version of the procedure.

PODUS is suitable for distributed environments because it uses remote procedure calls to preserve procedure-call semantics across computer boundaries. Distributed PODUS[10] is nearly identical to the centralized system except that semantically dependent procedures must reside at the same site. This exception lets site administrators initiate updates without worrying about other sites. Using a monitor-like control structure for remote-procedure-call server-side code, and system-generated stubs for client-side code, PODUS successfully updates geographically distributed programs.

PROTOTYPE SYSTEM

To evaluate the potential of the PODUS approach, we developed a prototype implementation and several sample programs that would be candidate scenarios for dynamic updating. The prototype helps us experiment with dynamic updating techniques and provides timing tools to evaluate the PODUS updating algorithms. It runs in a Berkeley Unix-compatible environment (such as SunOS or Ultrix).

We have used the prototype to dynamically update several sample programs of varying complexity.[9,10] For one applica-

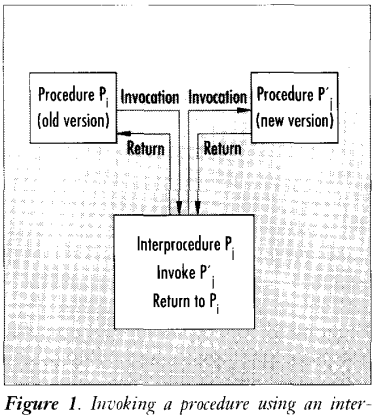


Figure 1. *Invoking a procedure using an interprocedure.*

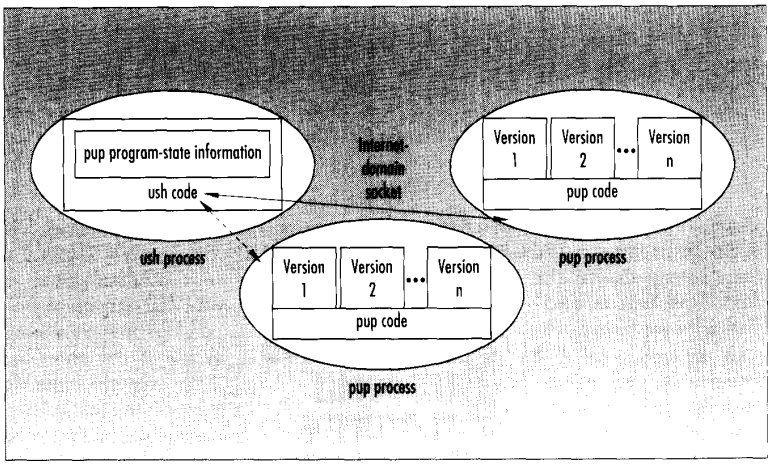


Figure 2. *Structure of a prototype system based on PODUS, which has two components: the program-update processor, called pup, and the updating-system shell, called ush. The dashed arrow indicates a potential interaction. Ush communicates with only one process at a time. No connection is set up until it needs to comunicate with another pup process.*

tion — the updating of a hypothetical Internet packet router — we did a timing analysis.[9] The analysis revealed that throughout the actual update, there was relatively little system degradation; the router continued processing at 90-percent efficiency.

Components. The prototype has two main components, as Figure 2 shows. The updating-system shell provides commands for loading, running, and dynamically updating programs, as well as facilities for managing multiple programs and online help. Users interact with PODUS by sending commands to the updating-system shell.

The program-update processor controls a separate Unix process in which programs are run. The processor manages the loading of programs and provides the virtual-memory primitives to perform dynamic updating. It accepts commands from the updating-system shell and processes them, taking no action on its own other than to notify the shell when operations complete.

The user program runs inside the program-update processor's address space. For each version of a program loaded into the processor, the program's code, static data, and procedure name/address mappings are saved. Interprocedures and mapper procedures needed to update from version i to version $i+1$ are also stored in the processor's version area. Overall program-state information (like procedure-call graphs and the state of an update) are stored in the updating-system shell.

Under normal circumstances, the user program cannot detect the presence of the program-update processor or the updating-system shell. Because all processor-shell communication is done through Internet interprocess communication, the shell and the processor need not reside on the same machine. In fact, in one updating application, the shell ran at Bellcore while the processor ran at the University of Michigan. Multiple program-update processors (for different programs) can be run on the same physical machine; a PODUS port mapper lets programs register and unregister themselves with the machine.

Updating. The user may initiate an update at any point throughout execution. Once an update request has been generated, the updating system determines the set of active procedures, S. A procedure, P_i, is active if it is on the runtime stack or its new version can directly or indirectly invoke any other procedure, P_j, that is active. Thus, by definition, the main procedure is always active.

Having determined S, the updating system replaces the old version of P_i with its corresponding new version, P_i', whenever P_i is not an element of S. Once a procedure is updated, it is marked as new and is not considered further.

PODUS identifies active procedures by examining the state of the program's runtime stack and procedure-call graph. It looks at the calling relationships between procedures — syntactic dependencies — and uses that information in determining when the user can update a procedure.

However, syntactic dependencies alone are not always sufficient for it to make this determination. Sometimes a group of procedures will interact in subtle ways that cannot be ascertained from the program's syntax. For example, a group of procedures that control different parameters for a robot may not call each other, but still depend on each other's actions. We call these relationships semantic dependencies.

Thus, PODUS must look at both syntactic and semantic dependencies in determining when an update is possible. It can find syntactic dependencies automatically because the procedure-call graph can be calculated from the language's syntax and program's source code. However, it cannot detect semantic dependencies automatically, so the programmer must specify them before the program is updated. PODUS atomically updates semantically dependent procedures. Such updates occur only when all the semantically dependent procedures are simultaneously inactive.

Both syntactic and semantic dependencies have formal definitions[9] that guarantee an old version of a procedure P_i invokes the most recent version of P_j, namely P_j or P_j', depending on whether an update of P_j has already occurred. However, a new version of P_i, P_i', invokes only the new version of a procedure P_j, namely P_j'.

Because the set of active procedures changes throughout the program's execution, S must continuously be reevaluated. To reduce the overhead incurred by repeatedly recomputing S, you must be able to tell when a procedure becomes a candidate for updating. Quite simply, an inactive old procedure cannot become active because it would have been converted to a new version when it first became inactive. Thus, recomputing S is limited to when an active procedure becomes inactive. Old active procedures become inactive precisely when the runtime stack contains fewer elements than there were during the most recent procedure update. If no procedures have been updated, old active procedures become inactive when the runtime stack contains fewer elements than when the update was initiated.

P_i comprises two components: the procedure specification, P_i.spec, and the pro-

cedure implementation, P_i.imp. P_i.spec abstractly corresponds to the specification of P_i's behavior, while P_i.imp corresponds to the behavior's implementation.

P_i.imp is represented by an address corresponding to where the code for the procedure resides. Conceptually, PODUS keeps track of the specification/implementation binding in a data structure called a binding table. The binding of P_i.spec to P_i.imp is a mapping between P_i's abstract behavior and a specific implementation of that behavior. In this context, P_i's interface (calling sequences and return value) is part of P_i.imp.

Changing the interface. PODUS views an update as the change of binding of P_i.spec from P_i.imp to P_i'.imp. If the interface to a procedure does not change as part of the update, the rebinding of P_i.spec to P_i'.imp poses few problems.

If an update does require changing the interface, however, there must be a way to convert the old interface to the new interface as part of the procedure invocation and the new interface to the old interface as part of the return statement. Consider, for example, a sort routine that initially sorts integers and is later replaced by a sort routine that sorts real numbers. A routine is needed that converts the data format from integer to real numbers on procedure invocation and back to integer values upon the return. Likewise, data included in or missing from the new invocation must be maintained or generated to ensure program correctness.

In the sort example just described, a possible interprocedure for the sort-procedure invocation is logically defined as

```
interprocedure sort (data : array of integer);
var
  real_data : array of real;
{
    real_data := convert_to_real (data);
    sort (real_data);
    data := convert_to_integer (real_data);
{
```

Address space. PODUS's underlying architectural model consists of a very large, sparse virtual-address space. Sparse address spaces are not required; instead, they are used as an abstraction of the runtime environment. As Figure 3 shows, each machine address consists of n bits and the following components:

- a machine ID denoting at which machine the program segment resides;
- a version ID denoting which version of the program the address references (multiple versions can reside simultaneously);
- a set of type bits to signify whether the procedure the address references is a normal procedure, an interprocedure, or a mapper procedure;
- a procedure ID that specifies which procedure the address references;
- a displacement, which represents the offset into the code of the procedure denoted by the procedure ID; and
- other miscellaneous bits used by the operating system for various related functions such as process IDs or protection information.

By incorporating a version ID into the virtual address, the large address space is partitioned into a number of version spaces. Within each version space, the object code for a specific version of a program and its static data are stored, along with a binding table. Only the procedures in that version space use the binding table.

When a procedure is updated, the

VIRTUAL MEMORY AND SPARSE VIRTUAL-ADDRESS SPACES

Conventional virtual-memory systems give you the illusion that the computer has more physical memory for your program than it really does. By moving parts of a program from disk to memory and back, the virtual-memory system lets you run programs that are larger than the physical memory. Virtual memory frees you from having to manage overlays (make sure the correct parts are in memory at a given time).

As the cost of memory continues to decrease, the importance of running large programs in computers with small physical memories is also diminishing. In spite of this trend, virtual memory is still an important part of contemporary operating systems. Besides providing a mechanism for managing small physical-address spaces, it is used in multitasking operating systems to prevent the memory used by one process from being accessed or modified by another.

Also, if the architecture can manipulate large enough addresses, the information used by the operating system to protect a process's address space can be encoded into the address itself. Using this technique, you can encode attributes such as location information (useful in distributed systems), owner information, and type information into an address.

If the addresses are made even larger, you can address file data as if it resides in primary memory. This style of file access is called a single-level store. A single-level store was first provided in the Multics system and has been since used in the IBM System/38 and the Apollo Domain system. By combining a single-level store with the addressing information, you can build a system that uses a large, sparse virtual-address space.

The address space is sparse because most of the addresses do not actually contain data. Instead, only the parts of the address space that contain useful information are brought into physical memory as is done in conventional virtual-memory systems. Similarly, programs in this type of system do not generate addresses that contain hundreds of bits; the large addresses are generated from a combination of normal machine addresses and values stored in registers. The HP Precision Architecture and IBM PC RTs use variations of this style of addressing.

Most of today's architectures do not directly provide mechanisms for manipulating sparse address spaces. Sparse address spaces can be emulated using existing machine registers (instead of registers explicitly designed for this purpose) to manage parts of the address space. In PODUS, for example, we use the registers normally associated with segmented virtual-memory systems to manage the version and procedure IDs associated with the large addresses. By performing this mapping, we can provide a good conceptual design as well as an efficient implementation on existing architectures.

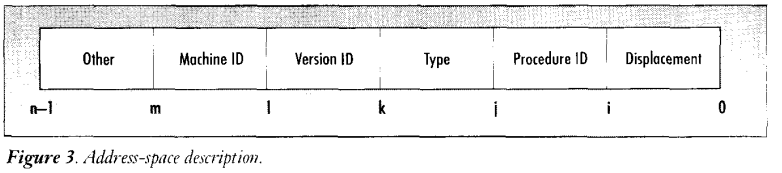


Figure 3. Address-space description.

binding table in the old version space is set to point to the interprocedure for the procedure, and the binding table in the new version space is set to point to the new version of the procedure. Active old procedures that invoke the procedure just updated actually invoke the interprocedure, while updated procedures will invoke the new version of the procedure.

Unfortunately, most current CPU architectures do not provide support for this kind of addressing. To overcome this deficiency, we map the components of these large addresses onto the existing registers of a conventional segmented virtual-memory system, as described in the box on p. 63. Performing this mapping (described in detail elsewhere[9]) lets PODUS use a sound architectural model and still be realizable on existing hardware.

PODUS ALSO SUPPORTS MULTIPLE VERSIONS OF A RUNNING PROGRAM.

Comparison. Like some other updating systems, PODUS provides mechanisms and policies for preserving program correctness during an update. But it goes beyond many of these systems by providing an algorithm for replacing the procedures of the program being updated in an appropriate order. This algorithm, along with methods to manually override it, constitutes PODUS's updating policy, which together with updating mechanisms, lets you update programs quickly with minimal human intervention.

PODUS also minimizes performance degradation. By building an updating system on top of a well-known (and well-studied) foundation — segmented virtual memory — it can exploit a range of existing hardware, software, and theory. Although not a requirement, a good implementation of PODUS using commercial virtual-memory hardware could substantially improve our prototype system's performance.

Unfortunately, we cannot compare PODUS's performance to that of other software-based updating systems because, to our knowledge, no performance data on these systems has been published. As a result, there is very little information to determine how fast our system is in comparison to other updating systems, or how fast it should be. We have published performance data on the effects of updating a small program using a prototype implementation of PODUS.[9]

This preliminary data showed that PODUS did not significantly degrade the performance of the program being updated. More meaningful data could be obtained by comparing how quickly production-grade implementations of PODUS and like systems updated similar large, complex programs. Programs must have a single thread of control. Data shared between procedures (including file data) must be accessed through abstract data types. As long as these constraints are obeyed, PODUS lets you update a procedure's code, its interface to other procedures, and the implementation of its internal data structures. Besides making localized changes, these building blocks can be used to restructure the program being updated. Many earlier systems perform only a subset of these changes.

Unlike many earlier updating systems, PODUS does not force you to use specific development tools to obtain the benefits of dynamic updating. It does, however, require you to make appropriate modifications to compiler code generators and linkers to correctly interface them with the updating system. Although this work is not trivial for the system programmer, the modifications will be transparent to the application programmer. To some extent, PODUS's design dictates the class of languages and programming style, but we believe its requirements are reasonable. They encompass a range of languages and specify top-down programming, a style that has been advocated for some time.

We also believe that the language and programming style scale to distributed environments — many of which must run continuously and would benefit significantly from the use of a dynamic updating system.

PODUS updates distributed programs written using a subset of the remote-procedure-call paradigm. It goes beyond most updating systems by supporting multiple versions of a running program. This support is essential in a geographically distributed environment because updates take time to propagate across unreliable networks. We envision future distributed systems being shared by multiple, administratively distinct organizations, each of which uses its own favorite hardware and software. If these systems are to interact (and be updatable), a dynamic program-updating system must be able to operate under these circumstances. By design, PODUS accommodates heterogeneous distributed hardware, software, and administrative domains.

Finally, unlike some updating systems that require redundant hardware to provide dynamic updating facilities, PODUS needs no special hardware to work properly. Virtual-memory hardware can be used to improve PODUS's performance, but it is not essential.

All the software-based dynamic updating systems described require some kind of indirection between the program modules that invoke each other. If indirection cannot be incorporated into a language or its underlying runtime system, dynamic updating cannot be done. As we have seen, indirection is not sufficient for dynamic updating. A dynamic updating system must also provide techniques for preserving the correctness of a program being updated.

Much work remains to be done before dynamic updating systems can become an integral part of today's computing infrastructure. Some tasks that will require further study are

♦ *Develop tools for testing support code.* Although there are tools to help programmers make intelligent decisions about code modification, thus reducing the chance of human error, there are no tools for constructing and testing the support

code of dynamic updating systems (the interprocedures and mapper procedures). Testing support code under conditions similar to the actual update is particularly difficult.

♦ *Increase the number of languages and styles that can be updated.* Future updating systems should support multilingual programs and should let you update programs in different styles. Most of the techniques we described update programs by procedure or module, for example. Future updating systems might update programs written in declarative, functional, or object-oriented languages.

♦ *Support updating of multithreaded programs.* Because many future programs are likely to be distributed or parallel, there must be support for multithreaded programs as well as programs that do not communicate using remote procedure calls. Some systems, such as Conic, can update programs that use a limited form of message passing, but future systems should extend this idea to support programs that use less restrictive communication patterns.

♦ *Obtain comparative performance data.* No such data for updating systems has been published, although dynamic updating systems have been around since the 1970s. The lack of data is probably because either the updating system's performance was hard to characterize—and consequently the data was hard to collect—or they were built as research prototypes and never applied to real problems. We particularly need experiments that evaluate the effects the computational model and the granularity of the updatable component have on the performance of the updating system. This information will illustrate the limitations of current systems and help identify areas for further research. Performance information will also aid practitioners who are trying to select an updating system for their domains.

♦ *Support hard real-time programs.* These programs can really benefit from dynamic program updating because they must produce correct results at a specific time. By definition, they should not be interrupted to install new versions (or for any other reason). Unfortunately, no existing updating system supports the updating of these programs, probably because they are especially difficult to maintain.

At Bellcore, we are investigating the feasibility of using PODUS to dynamically update several large Bellcore products. We are also expanding PODUS's updating algorithms to work with distributed systems that do not communicate using remote procedure calls — specifically the control software for a multimedia communications system. ♦

ACKNOWLEDGMENTS

We thank Peter Bates, Al Davis, Gita Gopal, Carlyn Lowery, Lillian Ruston, John Unger, and the anonymous *IEEE Software* referees for their many helpful suggestions. Their knowledge of some of the systems we describe helped us improve the accuracy and clarity of this article. Their stylistic comments also dramatically improved the editorial quality.

Bellcore's policy is to avoid any statements of comparative analysis or evaluation of products or vendors. Any mention of products or vendors in this article is for scientific accuracy and precision or for illustration and should not be construed as commentary. The inclusion or omission of a product or vendor should not be interpreted as indicating a position or opinion of either the authors or Bellcore.

REFERENCES

1. R. Fabry, "How to Design A System in Which Modules Can Be Changed on the Fly," *Proc. Int'l Conf. Software Eng.*, IEEE-CS Press, Los Alamitos, Calif., 1976, pp. 470-476.
2. H. Goullon, R. Isle, and K. Löhr, "Dynamic Restructuring in an Experimental Operating System," *IEEE Trans. Software Eng.*, July 1978, pp. 298-307.
3. T. Bloom, *Dynamic Module Replacement in a Distributed Programming System*, doctoral dissertation, MIT Press, Cambridge, Mass., 1983.
4. B. Liskov, "Distributed Programming in Argus," *Comm. ACM*, Mar. 1988, pp. 300-312.
5. J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. Software Eng.*, June 1989, pp. 663-675.
6. R. Yacobellis et al., "The 3B20D Processor and DMERT Operating System: Field Administration Subsystem," *Bell Systems Technical J.*, Jan. 1983, pp. 323-339.
7. I. Lee, *Dynos: A Dynamic Modification System*, doctoral dissertation, University of Wisconsin, Madison, 1983.
8. M. Segal and O. Frieder, "Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime," *J. Software Maintenance: Research and Practice*, Sept. 1989, pp. 54-79.
9. O. Frieder and M. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *J. Systems and Software*, Feb. 1991, pp. 111-128.
10. M. Segal and O. Frieder, "Dynamically Updating Distributed Software: Supporting Change in Uncertain and Mistrustful Environments," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif, 1989, pp. 254-261.

Mark E. Segal is a member of the technical staff in Bellcore's Network Systems Research Department. His research interests include operating systems, distributed systems, software engineering and maintenance, computer networks and software structures for large-scale multimedia communications systems.

Segal received a BS, an MS, and a PhD in computer and communications sciences from the University of Michigan, Ann Arbor. He is a member of the IEEE and ACM.

Ophir Frieder is an associate professor of computer science at George Mason University. His research interests include parallel and distributed architectures, database systems, and operating systems. He is also a staff consultant for the Federal Bureau of Investigations and the Institute for Defense Analysis.

Frieder received a BSc in computer and communications science and an MSc and a PhD in computer science and engineering, all from the University of Michigan. He is a member of the IEEE Computer Society and Phi Beta Kappa.

Address questions about this article to Segal at Bellcore, MRE 2A275, 445 South St., Morristown, NJ 07962-1910; Internet ms@thumper.bellcore.com.