Owen H. Tallman

Project Gabriel: Automated Software Deployment in a Large Commercial Network

Digital entered into an agreement with a major French bank to develop an automated software deployment facility, i.e., to provide centralized control of software installations and upgrades for a large network of computer systems. Independently, Digital had developed a set of models designed to guide the design of solutions to this type of complex management problem. The bank project team, which had considerable experience building distributed system management applications, was able to take advantage of these models. The result was a versatile, scalable application for distributed software deployment, validation of the models, and a clearer sense of the usefulness of such models to complex application problems.

A large French bank purchased a DECnet network from Digital and was in the process of deploying the network to support all its banking operations. The network topology included approximately 3,000 OpenVMS VAX systems and about 18,000 MS-DOS PC workstations. As illustrated in Figure 1, these systems were arranged in a branch structure that roughly followed the geographical distribution of the bank branch offices and their roles in the branch hierarchy. At the bank's headquarters, an OpenVMS cluster and an Ethernet local area network (LAN) linked the mainframe data center with the rest of the banking network. The cluster was connected to the first tier of approximately 200 branch group servers. The second tier consisted of approximately 1,800 branches, each with between one and four branch servers, for a total of about 3,000 branch servers. Each branch server, in turn, provided Digital's PATHWORKS and application services to the PC workstations.

For its nationwide backbone network, the customer was using a public X.25 network, which was its only available option.[1,2] The cost for X.25 service was based on usage, so each packet of data transmitted increased the operation cost. Therefore, the need to minimize this X.25 expense was a fundamental factor in specifying requirements for virtually all software deployed in the network.

The bank's business depended on the correct, reliable, and efficient operation of the network. Consequently, network management was crucial. From the customer's viewpoint, such an undertaking meant management of systems and applications, as well as the communications infrastructure. By extrapolating its overall experience with the hardware deployment, and its initial experience with software deployment, the customer foresaw potentially unacceptable labor costs for software deployment using the available methods. The customer therefore gave high priority to improving the software deployment process.

In this paper, the term deployment (or deployment operation) represents a process that deploys a set of software components to a set of systems. A deployment is described by a deployment plan and requires

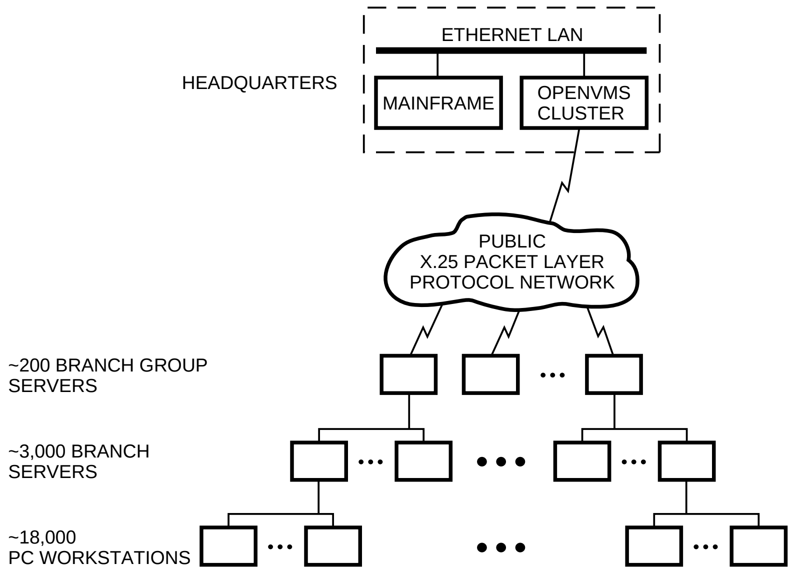

Figure 1
DECnet Network Topology in a Banking Environment

a deployment program, deployment automation software to execute the program, and an operations staff to schedule and monitor deployment program execution and, when necessary, respond to run-time problems.

The Software Deployment Problem

Ideally, the bank wanted networkwide consistency in its software, with automated, nondisruptive upgrades administered from a central point. Given the scale of the network and the number and variety of software components in use, however, this was not a realistic goal. The challenge of building a system of automated deployment tools that is capable of maintaining consistency across 3,000 widely distributed, frequently updated systems is significant in itself. Adding the problems of maintaining consistency in detailed business practices and user training in every branch greatly increases the difficulty. Actually, the business required software configurations tailored to and maintained consistently within individual business units such as branches and branch groups. Software upgrade planning and deployment activities would be essentially continuous, with numerous planning and deployment operations under way concurrently. The bank's business would not tolerate network malfunctions caused by ongoing upgrade operations or version mismatches among systems in a business unit, nor would it provide for on-site support at branches or branch groups. To implement a fully automated software deployment process would require rigorously managed, centralized planning and operational control.

The bank had already implemented a system that automated significant parts of the deployment process, using a variety of existing tools and ad hoc integration. These tools included Digital Command Language (DCL) command procedures, the Information Distribution Controller (IDC) product, which distributes files in batch mode, and a system event reporter. The process, however, was still labor intensive. The customer concluded that the only way to achieve acceptable operational costs was to increase substantially the degree and quality of automation in the process.

Customer Requirements

A solution to this software deployment problem would have to support (1) sophisticated, carefully managed planning, (2) a means of determining the current state of target systems for use in planning, (3) rigorous software certification, and (4) a highly reliable means of automating software distribution and installation. The bank's planning and certification processes were already developed, staffed, and in operation. An inventory control database for tracking system configurations was under development. However, the means to distribute and install software effectively was lacking and would have to be developed and then integrated with the other system components. The customer emphasized this need for distribution and installation automation when it first presented the problem to Digital.

All new software must be evaluated, acquired, packaged in kits that can be installed automatically, tested, and certified. Since software interdependencies may exist, multiple software components may need to be processed together to ensure proper installation and operation as a set. (In this paper, the term component refers to any software that might be distributed as a kit, e.g., a commercial layered product, an in-house application, or a patch.) Planners must determine which of the certified components to install, the branch group to install them in, and the scheduling constraints. The result is a carefully documented, uniquely named deployment plan. Deployment execution consists of performing all the steps necessary to distribute and install the software on the target group and to report the results for incorporation in the planning for the next deployment.

The operations staff, i.e., those who monitor and control the network on a continuous basis, keep a repository of data that reflects the current state of software on the systems in the network. Planners use this data to plan new states for parts of the network; they store these plans in the repository also. As many as 10 planners may be developing plans simultaneously. For each plan, an application analyzes the differences between the planned state and the current state of the network and produces a deployment program.

A deployment operation may involve multiple products. This set of products must include all those necessary to satisfy the prerequisites of the other members of the set (if they are not already satisfied by products on the target system). The members of the set must be installed in the proper order. The planners determine the proper membership for any product set and create representations of those sets in the repository. They also represent the product installation order in the repository in the form of installation precedence relationships. The deployment software uses this precedence information to determine the order of installation for members of a product set.

The operations or configuration staff store the certified software kits in a library at the management center. When the kits need to be installed on a system, the deployment software compresses the kits and then copies them across the X.25 backbone to staging areas on servers. From these areas, the deployment software copies the kits to the target system or systems or, if necessary, to servers closer to the target systems and then to the target systems, where the kits are decompressed and used. By staging kit distribution in this way, each kit is copied only once over each link, which avoids wasting bandwidth. When all the target nodes have the required kits, the kits at the staging points are deleted. The copy operations must proceed concurrently whenever possible. Table 1 shows possible states and transitions for a software component kit on a target system.

Table 1
States and Transitions for a Software Component Kit on a Target System

Initial State	Action	New State
(Null)	Copy	Distributed
Distributed	Delete	(Null)

Installation is a multistep process designed to allow the synchronized change of operating software on all related systems. Once the required kit is present on the target system, the product can be installed, i.e., the files put in place and any other necessary steps taken so that the product is ready to be activated. Activation, i.e., making the new product the current operating version, is the last step. A product can also be deactivated and deinstalled. To upgrade a product requires installing the new version, deactivating the old version, and then activating the new version. If the activation is successful, the previous version can be deinstalled. Only one version of a product can be active at any given time. Table 2 shows the states and transitions for a software component on the target system.

Table 2
States and Transitions for a Software Component on a Target System

Initial State	Action	New State
(Null)	Install	Installed
Installed	Activate	Active
Active	Deactivate	Installed
Installed	Deinstall	(Null)

Table 3 shows the state transitions to be managed between the new version product kit, the new version product, and the previous version product on the target system. Note that the deployment process should minimize the time a target system must spend in step 4, when both versions of the product are installed but neither is active.

Table 3
State Transitions to Be Managed on a Target System

Step	New Version Product Kit	Old Version Product	New Version Product
1	(Null)	Active	(Null)
2	Distributed	Active	(Null)
3	Distributed	Active	Installed
4	Distributed	Installed	Installed
5	Distributed	Installed	Active
6	Distributed	(Null)	Active
7	(Null)	(Null)	Active

A planner can specify to the deployment software that an upgrade must be carried out as an atomic transaction. That is, the activation transition must either succeed or be rolled back. In a rollback, steps 3, 4, and 5 in Table 3 are reversed. Most commercial software is not packaged with installation procedures that support installation, activation, deactivation, and deinstallation steps. Therefore, the bank must package its own software and repackage software from manufacturers so that upgrades behave this way. The deployment software invokes the individual steps by executing DCL command procedures provided in each such customized kit.

The activation of all products in a deployment may be transactional, in which case all the products must activate successfully or all activations will be rolled back. The installation steps for all the products are completed first, so all the products are ready for activation at the same time. The activations are then attempted. If all succeed, the newly activated products remain as the current operating versions. If a product activation fails, it and all the preceding activations are rolled back, in reverse order of activation, and the previous versions are likewise reactivated. When the rollback completes, the deployment stops and the management center receives a status report. Once the operations staff has corrected the problem that caused the failure of the activation phase, a new deployment program may be generated. It will execute only the activation steps, not any of the preceding steps that had succeeded. That is, the new deployment program picks up where the earlier one left off.

This transactional behavior applies to all activations across all systems in a given deployment and may involve different sets of products for different systems. The transactional characteristic applies to the deployment operation, not to a product or set of products. Thus, the deployment can accommodate interdependencies among products on different systems. If an activation of any product fails in a transactional deployment, all current or completed activations will be rolled back in reverse order of activation, regardless of location. This requirement is specifically for client-server applications whose client and server components must be upgraded both simultaneously and atomically.

The deployment software must maintain the state of the deployment in stable storage so that the state can be restored and the processing continued despite transient failures of systems or networks. The software must report the state of processing to the management center at some reasonable interval and also when the deployment completes. The software then updates the repository with the status of all the individual operations in the deployment.

The deployment implementation must provide management directives to start, suspend, resume, stop, and abort the deployment, without leaving it in an inconsistent state or disrupting business operations. Suspension prohibits any new command procedure executions from starting but does not interrupt ongoing ones, thus allowing the deployment to quiesce. Suspension does not affect transactions. The resume directive restarts execution of a deployment that has been suspended. Stopping is the same as suspension except that once stopped, the deployment cannot be restarted. The abort directive stops ongoing command procedure executions by terminating their processes and thus forces the rollback of any transaction that is executing at the time the directive arrives. An aborted deployment cannot be restarted. There is also an update directive, which forces the current details of operation state to be rolled up to the management center. A show directive reports the overall state of each deployment at a particular host.

The management directives allow an external entity, e.g., a batch scheduler or an operator, to intervene in what would otherwise be a self-contained, automated operation. A batch scheduler can suspend all ongoing deployments at some time before bank branches open and resume the deployments when the branches close. It can force a deployment to stop at a predetermined time, whether or not it has completed. An operator can use the update directive to roll up the state to determine how far a remote part of a large deployment has progressed. It can also issue suspend and resume directives to subsets of the network affected by a deployment to allow for emergency manual intervention without suspending the entire deployment.

Digital's Response to the Requirements

Digital's decision to undertake the project of developing an automated software deployment facility for the bank was based on two goals. First, Digital wanted to meet the needs of an existing customer. Second, in solving the customer's problem, Digital could validate the set of network and system management models it had already developed. The following sections provide an overview of the models and details of the automated software deployment implementation.

The EMA Configuration Management Model

When Digital began discussions with the bank about automating software upgrades, in the Enterprise Management Architecture (EMA) group, Paul Kelsey was developing a comprehensive general model of configuration management for information systems. Like the influential EMA entity model that preceded it, the EMA configuration management model (CMM) defines a consistent set of concepts and terms for working in its particular problem domain.[3] The entity model broke new ground by applying what would come to be known as object-oriented concepts to the problem of managing the many types of objects found in a network. The CMM goes on to address the relationships among those objects that, in combination with the objects themselves, constitute an information system's configuration.

Configuration management concerns a broad range of activities over the lifetime of an engineered system. The larger or more complex the system to be managed, the greater the need for a configuration management discipline. The U.S. Air Force defines configuration management as "a discipline applying technical and administrative direction and surveillance to (a) identify and document the functional and physical characteristics of a configuration item, (b) control changes to those characteristics, and (c) record and report change processing and implementation status. It includes configuration identification, control, status accounting, and audits. Configuration management is thus the means through which the integrity and continuity of the design, engineering, and cost trade-off decisions made between technical performance, producibility, operability, and supportability are recorded, communicated, and controlled by program and functional managers."[4]

The CMM provides a conceptual framework for automating information system management, covering the entire scope defined in the preceding paragraph. For example, consider a disk drive. The EMA entity model provides a conceptual framework for describing the drive as an object with certain attributes (e.g., storage capacity) and operations (e.g., format) such that developers can build software that allows monitoring and control of the object by means of a management protocol. Any object in the network that presents a conforming management interface is called a managed object.

The CMM proposes a framework for describing the disk drive's role in a system configuration over the drive's lifetime. The framework covers

1. The services that the disk drive provides and the clients of these services, e.g., the logical storage volume that the drive supports
2. The services that the disk drive consumes
3. The objects that compose the drive
4. The drive's current and previous attribute values
5. The attribute values that the drive should presently have
6. Plans for future drive configurations
7. The way software should interpret and act on list items 1 through 6

The following discussion emphasizes the aspects of the CMM that influenced the design of the Project Gabriel software.

Persistent Configuration Model

In the CMM, all users and management applications deal with managed objects in an information system, whether physical or abstract, in the abstract: they manipulate their representations in a repository, and automatic mechanisms carry out the implied operations transparently. The repository maintains a persistent representation, i.e., model, of the entire information system's state; it is called the persistent configuration model (PCM). The PCM provides a common level of abstraction for all users and management applications because all management actions are taken through it. Since the model persists, the PCM can provide this abstraction in multiple temporal divisions.

Temporal Divisions

Managed objects indicate their state through attributes and through relationships with other objects. Object state is relative to the temporal division of the PCM through which the state is viewed. Each temporal division can provide a consistent view of all the objects in the network as they were at some point in the past, as they are now, or as they will be.

The historical temporal division records past system states. The present is represented in the observed and expected temporal divisions, where the observed division provides the most recent information available on actual object state, i.e., what is now. The observed division is populated by automated census services that collect current state information as directly as possible from the objects. The expected division maintains what is currently intended for the object state, i.e., what should be. This division is based on the observed division but modified as necessary to represent the state sanctioned by the system or network administrator.

The planned and committed temporal divisions represent future object states. States that may be realized at some time are planned, whereas those that will be realized are committed. The distinction permits simulating, analyzing, and evaluating future states in the planned division without implying any commitment to realize them.

Realization

Differences between object states in the expected and the committed divisions indicate changes that need to take place to realize the new committed configuration. This is the task of the realization services. The job of identifying the required changes and generating a program to realize these changes is called configuration generation (CGN). Other realization services execute the program and update the repository based on the results. A software deployment operation would be called a realization in CMM terms. The ultimate vision of the CMM is to allow the user to define the desired state of an information system and, with a single command, to realize it.

Once the planned state has been realized, automated services can maintain that state by monitoring the differences between object states in the observed and the expected divisions. These differences represent possible faults and trigger fault-handling actions.

Implementation

Digital and the bank agreed that Digital would implement the critical deployment automation part of the bank's requirements and integrate it with the bank's established processes. The focus of the discussion in this section is the engineering team's efforts to arrive at an effective, implementable system design.

System Design

The CMM provided an effective conceptual framework for thinking and talking about the system requirements and possible design choices. As one would expect from a general model, however, the CMM did not address important design and implementation issues. In particular, it did not prescribe in any detail the PCM design or how the realization services should work. The Project Gabriel engineering team, which included the CMM author, had to quickly answer the following basic questions:

- How should the team implement the PCM? Is it an object-oriented database, or will it require functionality beyond what the team can implement in such a database? What schema should the team use? How much of the PCM as described in the CMM is really necessary for this project?
- How will CGN convert the PCM state data to a deployment program? Is CGN a rule-based application or a conventional, sequential program? What will CGN require of the objects in the PCM? How will CGN communicate to the other, as-yet-undesigned realization services what needs to be done to carry out a deployment? How should the team trade off the complexity of CGN versus the complexity of the services that will execute the programs?
- What services will the team need to carry out the programs CGN generates? What form will these services take?
- How can the team minimize the complexity of the system to arrive at a design that the team can actually implement?

The last question was in many ways the most important. The team had to break down the problem into manageable pieces and at the same time devise an integrated whole. The team did not have time for a sequential process of analysis, design, and implementation and, therefore, had to find ways to start development before the design was complete. CGN presented the pivotal problem; it might ultimately be the most difficult part of the system to design, but the components on which it depended had not yet been designed. In addition, these components could not be designed effectively without some reasonable idea of how CGN would work. To efficiently use the time allotted, the team began to search for the key design abstractions while it evaluated technologies and tools.

Actions and States PCM configuration data represent multiple actual or possible states of the systems in the network. CGN would generate a deployment program based on the differences between the expected and planned states represented in the repository. This idea led to the development of a state table, which prescribed the state transitions that would have to occur to change each product on each system from its present state (as shown in the expected temporal division) to its planned future state. CGN could associate an action with each transition and program those actions. When the PCM received status from the actions taken on the target systems, the transition identifier would be included and would be used to update the PCM. This became one of the key design concepts of Project Gabriel: to model the target of a deployment operation as a collection of finite state machines.

CGN needed a way to program the actions so the other realization services could carry them out. The team chose to model the actions in a consistent manner for all foreseeable variations, regardless of how they are implemented or what state change they effect, as follows:

1. All actions consist of invoking a command, with some list of arguments, on some object, and within a discrete process.
2. Actions are associated with state transitions. Actions themselves have state (e.g., running) and finite duration. Actions can be started, and at some point they complete. When they complete successfully, they change the state of an object; when they fail, they do not.
3. The implementation of the command should behave such that an action's failure has no undesirable side effects, e.g., disabling a system component or causing large amounts of disk space to be occupied needlessly. This behavior cannot actually be guaranteed, however, so some failures may require human intervention to correct side effects.

In most respects, this model of command procedure execution is the same one used by both the OpenVMS batch facility and the POLYCENTER Scheduler. The principal difference is that in Project Gabriel, a user does not simply program an arbitrary sequence of actions. Rather, each action corresponds to a specific meaningful state transition of an object. When the PCM receives completion status for an action, the PCM update program can use the transition identifier to determine what state an object has attained and modify its representation in the repository accordingly.

By hiding the implementation internals behind a consistent interface in this manner, the software designed for controlling actions does not have to be concerned with those internals. This is a straightforward application of the principle of encapsulation, which separates the external aspects of an object from its internal implementation details.[5] Encapsulation allows a system designer to separate the question of how an action, such as copying a file or invoking an installation procedure, is implemented from the question of what interface the control system will use to invoke the action. This is obviously a simplification of the implementation issue, because the team had to deal with preexisting implementations, which cannot always be made to follow new rules. From a design point of view, however, the simplification is essential.

Control Distribution A deployment operation consists of multiple actions, performed in various complex sequences. The team understood intuitively that every host system would have to run software to execute the deployment program and that the management center would distribute the program to the other host systems in the network. An advanced development team working on a more scalable design for the POLYCENTER Software Distribution product had previously developed a model for this kind of distributed control. The Project Gabriel team adopted two related design ideas from its work.

The first idea is recursive program decomposition and delegation. Assume that the control system is implemented by servers called control points, whose task it is to coordinate operations. Assume also that each target system has an agent that carries out the action. Assign to each target agent a control point, and assign to each control point its own control point, such that these control relationships form a tree structure.

Assume that deployment programs are composed of nested subprograms, which, in turn, are composed of nested subprograms, and so on. Assume also that each program (or subprogram) has an attribute identifying the designated control point to which the program must be sent for processing. Such programs can be decomposed, distributed, and executed using a recursive distribution algorithm, as follows.

An operator submits a complete deployment program to its designated control point. (Submission consists of copying the program file to a well-known place on the management center host system and issuing a RUN command with the file name as an argument.) The control point breaks down the program into its component subprograms and submits the individual subprograms to their own designated control points, thereby delegating responsibility for the subprograms. The delegation ends when a subprogram has been broken down to the level of individual actions, which are delivered to the agent on the target system for execution. In the original model developed for POLYCENTER Software Distribution, program structure did not influence how operations were decomposed and delegated. Instead, a target could be a group of targets, allowing recursive delegation of subprograms according to the nesting of the groups. The Project Gabriel innovation was to use nested subprograms within the deployment program rather than nested target groups. Both approaches are built on the notion of distributing control by following a tree whose nodes are managed objects and whose edges are control relationships. This is how they were ultimately represented in the PCM.

The second idea relates to program state. The team modeled the deployment program and each of its component subprograms as finite state machines. Each subprogram goes through a definite series of transitions from ready to completed, stopped, or aborted. The state of the program as a whole reflects the state of the processing of its component subprograms, and the state of each component reflects the state of the processing of its components, and so on. At any time, an operator can issue a show directive for a control point and determine the local state of all deployment programs. Understanding the collective, distributed state of a deployment may be difficult at times, because a given control point may have outdated information about a delegated subprogram. For example, a program may be running when none of its components are running yet, when some are running, and when all have completed but notice has not yet rolled up to the root of the control tree. This latency is natural and avoidable in such a system.

The deployment software maintains program state on disk. When a component subprogram is delegated, the state is reflected at the sender by a placeholder subprogram that stands in for the one created at the receiver. The state is updated at the sender only after the receiver acknowledges receiving the subprogram and securing it in stable storage. Given this conservative approach to recording state changes, and logic that makes redundant delegations harmless, a control point server can be stopped or restarted without losing program state.

Data Distribution The team borrowed the notion of a distribution map from the IDC product mentioned in the section The Software Deployment Problem. The Project Gabriel concept is a distribution tree, which is formed in the same fashion as the control tree. Each host system is assigned a distribution point from which it gets its copies of software kits to be installed. A system that hosts a distribution point has its own assigned distribution point, and so on, for as many levels as necessary. This assignment takes the form of relationships between system objects in the PCM. CGN uses the distribution tree to determine the software distribution path for each target system.

The control and distribution trees need not be the same, and they should not be confused with one another. The control tree uniquely defines the path by which all other services, e.g., kit distribution, are managed.

SYREAL Programming Language To communicate a deployment plan to the servers that were to execute it, the team invented a simple textual representation called the system realization language (SYREAL). This language was easy for the developers and users to analyze in case problems developed and could easily be produced by programs, by DCL command procedures, or by hand. Although SYREAL is verbose (e.g., installing a few products on a dozen systems requires hundreds of lines of text), it clearly reflects the structure of the deployment operation.

PCM Implementation The development team believed that an object-oriented repository would provide the most natural mapping of the PCM abstractions onto a data model. The team used an internal tool kit called AESM, which was layered on the CDD/Repository software product. The user interface is based on DECwindows Motif software, using facilities provided by AESM.

AESM uses membership, i.e., containment, relationships to connect objects in a meaningful way. All relationships are derived by inheritance from this basic type. Thus, the PCM contains temporal divisions, which contain groups of systems, which contain software configurations, which contain specific software components with certain state attributes. A software catalog contains configurations, software components, and materials objects that describe the kits used to install these components. A plan in the PCM is an object within the planned domain that contains systems and configurations.

Configuration Generation Processing Thus far, the paper has described the following abstractions available for CGN:

- The PCM, which contains systems and a catalog of software configurations, software components, materials, and precedence relationships—all in temporal divisions.
- Software component state table.
- Actions, which change the state of objects in the network.
- Managed objects (e.g., software components and kits) as finite state machines whose transitions result from actions.
- A control tree to partition control responsibility. This tree consists of relationships between control points and between control points and target agents.
- A distribution tree to define the path for distributing software to target systems. This tree consists of relationships between distribution points and target agents.
- Deployment programs as finite state machines whose nested structure is decomposed and distributed according to the control tree.
- Control point servers that execute deployment programs and target servers that execute actions.

Given these abstractions, the key problem of designing CGN was to determine the optimal order of traversing and analyzing an interrelated set of trees connected with a plan in the PCM. The solution had to address

- The PCM temporal divisions, to locate expected and committed states of system configurations in the plan
- The software catalog, to determine materials and precedence relationships
- The precedence relationships, to determine the processing order for the products in the plan
- The control tree, to determine how control must be distributed
- The distribution tree, to determine how software kits must be distributed

For each system, CGN must determine what products will undergo which state transitions based on the state table. The same set of abstractions made it clear what form SYREAL should take and the nature of the processing that the control point and target servers would perform.

Reducing the problem to a small number of abstractions, many of which shared a similar structure, was a major step in the process of defining an implementable system. Although the overall problem was still complex and required a nontrivial effort to solve, at least the problem was bounded and could be solved using conventional programming techniques.

Overview and Example of Deployment Processing

A user, i.e., planner, begins the deployment process by populating the repository with objects to be managed using an application that reads from the inventory database. The objects in the repository represent a software catalog, expected and planned temporal divisions, computer systems, software products, software configurations, software materials (kits), and product pick lists. By specifying the relationships between the objects, i.e., by actually drawing the relationships, the user develops a model of the network configuration. For example, a model may represent a system that has a particular software configuration and is contained in one of the temporal divisions.

In addition to allowing the user to model the network, the deployment software represents policy information by means of relationships. A software product may have precedence relationships with other software products that prescribe the installation order. Each system has a relationship that indicates its distribution point, i.e., the file service that provides staging for software distribution to that system. Each system also has a relationship that indicates its control point, i.e., the management entity that controls deployment operations for that system.

Using the graphical user interface, a planner derives new configurations from approved configurations in the repository and assigns the new configurations to systems or groups of systems. A planner can view the differences between the current and the proposed configurations and see which systems will be affected. If the observed changes are acceptable, the planner can run CGN to produce a program to realize the changes. Once the program has been generated, the planner can launch it immediately, schedule it for execution later, or just review it.

Deployment programs normally run under the control of a batch scheduler. For large-scale deployments, which can continue for days, the scheduler automatically suspends execution while branch offices are open for business, resumes execution when the branches close, and repeats the cycle until the operation has completed. Operators oversee the execution of the deployment, intervening to suspend, resume, stop, or abort the process, or to observe the program's state. Actions on individual systems that fail may suspend themselves, thus allowing an operator to intervene and correct the problem and then, if desirable, restart the operation.

Certain events, such as a deployment action failure, roll up to the central control point and trigger the execution of a user-written event script. Depending on the type of event, the script may notify an operator, make a log entry, or perform a PCM update. Normally, the last event that occurs is the completion of the program. If the PCM completed successfully, it is automatically updated. Even if a program does not run to successful completion, the operator can trigger a PCM update so that whatever changes were realized will be reflected in the PCM. A new program, generated with the same planned configuration, will include only the changes that were not completed in the previous attempt.

The remainder of this section describes the role of each major Project Gabriel component in the deployment process. The example presented was intentionally kept simple. Its assumptions are as follows:

- The repository has been populated with network information, the product catalog, etc.
- The goal is to upgrade the software configurations of a set of four branch servers, B1 through B4.
- Central control points exist at headquarters, HQ, and on two group servers, G1 and G2 (see Table 4).
- Branch servers B1 and B2 have their control point on G1; B3 and B4 have theirs on G2. HQ hosts the control points for itself and for G1 and G2.
- The branch server systems have distribution points (file servers), which in this example are on the same host systems as their respective control points. (This overlap is not required.)
- In the PCM's expected temporal division, the four systems B1, B2, B3, and B4 are governed by the same software configuration. The only layered software product is Product X version 1.1, which is in the active state.
- The planners want to have Product Y version 2.0 installed on the four systems and in the active state. They create a plan in which a new configuration, with Product Y added, governs the systems (see Table 5). They commit the plan, which invokes CGN.

Configuration Generation CGN transforms the desired future state represented in the PCM to a program that can be used to realize that state. CGN determines the difference between the configurations in the

Table 4
Designated Management Control and Distribution Points

System	Control Point	Distribution Point
HQ	HQ	HQ
G1	HQ	HQ
G2	HQ	HQ
B1	G1	G1
B2	G1	G1
B3	G2	G2
B4	G2	G2

Table 5
Expected and Committed Configurations

Temporal Division	Configuration Name	Product	Version	State
Expected	GoodConfig	Product X	1.1	Active
Committed	BetterConfig	Product X	1.1	Active
		Product Y	2.0	Active

expected and committed temporal divisions, which in the example is the addition of Product Y version 2.0 in the active state. Since the configurations differ by only one product, the question of installation order does not arise. If multiple products were involved, CGN would analyze their dependencies and arrange them in the correct installation order.

CGN uses a state table to determine the sequence of transitions that must occur to bring the software to the desired state. In the example, Product Y version 2.0 is not present on any of the target systems, so the kit must be copied to the appropriate distribution point and then copied to the target systems, after which it must be installed and activated. CGN uses the distribution tree to find the appropriate distribution points and then uses the control tree to determine which control point to use for each set of systems, for each staging copy, and for each transition. Finally, CGN generates the corresponding text in SYREAL. The program that CGN writes optimizes throughput by performing concurrent processing whenever possible.

SYREAL Program A SYREAL program has two parts: (1) object declaration and (2) the executable. The first part declares the objects to be acted upon. The control point that executes the program has no knowledge of the software products, files, kits, copy commands, etc. It knows only that objects exist that have identifiers and that undergo named state transitions as a consequence of executing commands. SYREAL provides a means of declaring objects, their identifiers, the associated transitions, and the commands that effect the transitions. Figure 2 is an example of an object declaration. The program declares the realization object that represents Product Y version 2.0. The object name is PY. Note that PY is an ad hoc, purely local naming scheme. Since there can be only one instance of any product version on a system, the name is implicitly distinguished by its locality, in the sense that it is the unique instance of productPY on system X. PY inherits the default object characteristics (not shown) and adds its own kit identifier, product name, and a definition of the ACTIVATE transition. This transition has command CMD, which is a DCL command string.

The second part of a SYREAL program is the executable. (Figure 3 shows the executable part for the deployment process example.) This part consists of at least one executable block (i.e., subprogram), which may contain any number of additional executable blocks. A block may be defined as concurrent or serial. Blocks nested within a serial block are executed in order of appearance. Blocks nested within a concurrent block are executed concurrently.

Any block may have an associated fault action expressed as one of the following commands: ON ERROR SUSPEND, ON ERROR CONTINUE, or ON ERROR ROLLBACK. A block is executed by "USING" a designated control point to control it. For example, the first executable line in Figure 3, i.e., SERIAL BLOCK USING "HQ";, declares the execution of the outermost block to be assigned to HQ. Nested USING blocks may be assigned to other control points, to the point at which the ultimate action is called for. The SYREAL program expresses this assignment by an AT block, in the sense that the action

```
OBJECT PY CHARACTERISTICS LIKE DEFAULT;
  KIT_ID "PY020";
  PRODUCT_NAME "PY, 2.0";
  TRANSITION FETCH
    CMD "$@RLZ$SCRIPTS:RLZ$FETCH";
  TRANSITION ACTIVATE
    CMD "$@RLZ$SCRIPTS:RLZ$ACTIVATE";
END CHARACTERISTICS PY;
```

Figure 2
SYREAL Program—Object Declaration

```
SERIAL BLOCK USING "HQ";
  ON ERROR SUSPEND;
  SERIAL BLOCK AT "HQ";
    PERFORM FETCH
      OBJECT PY;
  END SERIAL BLOCK AT "HQ";
  CONCURRENT BLOCK USING "HQ";
    SERIAL BLOCK USING "HQ";
      SERIAL BLOCK AT "G1";
        PERFORM COPY
          OBJECT PY
          SERVER "HQ";
      END SERIAL BLOCK AT "G1";
      CONCURRENT BLOCK USING "G1";
        SERIAL BLOCK AT "B1";
          PERFORM COPY
            OBJECT PY
            SERVER "G1";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B1";
        SERIAL BLOCK AT "B2";
          PERFORM COPY
            OBJECT PY
            SERVER "G1";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B2";
      END CONCURRENT BLOCK USING "G1";
    END SERIAL BLOCK USING "HQ";
    SERIAL BLOCK USING "HQ";
      SERIAL BLOCK AT "G2";
        PERFORM COPY
          OBJECT PY
          SERVER "HQ";
      END SERIAL BLOCK AT "G2";
      CONCURRENT BLOCK USING "G2";
        SERIAL BLOCK AT "B3";
          PERFORM COPY
            OBJECT PY
            SERVER "G2";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B3";
        SERIAL BLOCK AT "B4";
          PERFORM COPY
            OBJECT PY
            SERVER "G2";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B4";
      END CONCURRENT BLOCK USING "G2";
    END SERIAL BLOCK USING "HQ";
  END CONCURRENT BLOCK USING "HQ";
  CONCURRENT TRANSACTION USING "HQ";
    CONCURRENT BLOCK USING "G1";
      SERIAL BLOCK AT "B1";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B1";
      SERIAL BLOCK AT "B2";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B2";
    END CONCURRENT BLOCK USING "G1";
    CONCURRENT BLOCK USING "G2";
      SERIAL BLOCK AT "B3";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B3";
      SERIAL BLOCK AT "B4";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B4";
    END CONCURRENT BLOCK USING "G2";
  END CONCURRENT TRANSACTION USING "HQ";
END SERIAL BLOCK USING "HQ";
```

Figure 3
SYREAL Program—The Executable

is aimed at an individual system. An AT block may contain one or more PERFORM statements, which perform the action called for. The second executable line in Figure 3, i.e., SERIAL BLOCK AT "HQ";, calls for the fetch transition on the objectPY. This action results in execution of the command @RLZ$SCRIPTS:RLZ$FETCH on HQ to fetch the distribution kit files from the software library.

A transaction is simply a block that enforces the fault action ON ERROR ROLLBACK. Nested operations must complete successfully or all will roll back. A transaction may be serial or concurrent and may contain nested blocks that are serial or concurrent. It may not contain a nested transaction.

Deployment Processing Control point and target servers are implemented on each OpenVMS system in the network by a single server daemon called the realization server (RLZ). On receipt of the SYREAL program, the first daemon, which is onHQ, converts the program to a binary representation on disk. This data file mirrors the nesting structure of the text file but allows for storage of additional state information.

The daemon then executes the program by sending the binary version of each block that is currently eligible for execution to the block's designated control point. Each control point that receives a binary block repeats this process, until anAT block arrives at its designated control point. The control point then sends to the target system's daemon a request to perform the action. The target daemon creates a process to execute the PERFORM command, captures completion status when the process exits, and returns the status to the control point. If the perform action is successful, the control point sends the next perform request. If the perform action fails, the control point decides whether to send the next perform request, to suspend processing until an operator can intervene, or to initiate a rollback. This decision depends on the fault action in effect.

The RLZ daemon maintains processing state on disk to allow recovery from system failures, loss of network connectivity, and other transient calamities. As block processing completes, block status is rolled up to its containing block, whether local or on a remote control point. The state of the block changes to reflect the block's interpretation of the states of its nested blocks. At each level, the control point decides if, as a result of status reports, one or more additional blocks should be executed. Ultimately, the central control point at HQ will have received the status of all operations. If all the perform actions completed successfully, as determined by the fault actions specified, the deployment completes successfully. Otherwise, the deployment fails. Completion triggers execution of a PCM update script.

PCM Update The overall status of a Project Gabriel realization is an interpretation of the results of many individual operations, some governed by fault actions different from those of the others. Because CGN dynamically generates the block structure of a realization program, the structure has no direct counterpart in the PCM. Therefore, only the results of individual perform actions are of interest for updating thePCM. The update program examines the completion status of each perform action completed on each object on each target system. The program updates the corresponding objects in the PCM based on the results of the last action completed on each object.

Note that since object and transition definitions are specific to a particular SYREAL program, realization servers are not limited to the object classes that Project Gabriel's CGN and PCM update handle. Applications can be written to perform other kinds of operations with new object classes, transitions, etc.

Realization Block Diagram Figure 4 illustrates the complete processing that the RLZ servers carry out in response to the example SYREAL program in the case where no faults occur. Events flow from left to right. The outermost block contains all the events of interest except PCM update, which is implicit in every SYREAL program and carried out automatically by the RLZ server at the root of a deployment operation.

The first action to be executed within the outermost block is fetching PY from the library to staging storage on HQ, under the control of HQ. Subsequently, HQ controls concurrent operations to copy PY from HQ to both G1 and G2. When the copy action is completed on either G1 or G2, HQ transfers the next block to the respective control point to perform the copy and install actions on its two targets. For instance, the concurrent block using G1 executes the copy action to B1 and then the install action on B1, while the same sequence executes on B2. Processing of these concurrent sequences synchronizes on G1 when both complete. At that time, the status of the entire concurrent block using G1 rolls up to HQ, where processing will again synchronize with the concurrent block using G2.

HQ also executes the concurrent transaction. This execution flows similarly to the preceding concurrent block execution except that since no action needs to be taken on G1 or G2 before proceeding to act on B1, B2, B3, and B4, the serial blocks at G1 and G2 are unnecessary.

Fault Handling In the deployment example, the fault action represented by the command ON ERROR SUSPEND governs the steps prior to the transaction. This means that, if an action fails, no dependent action

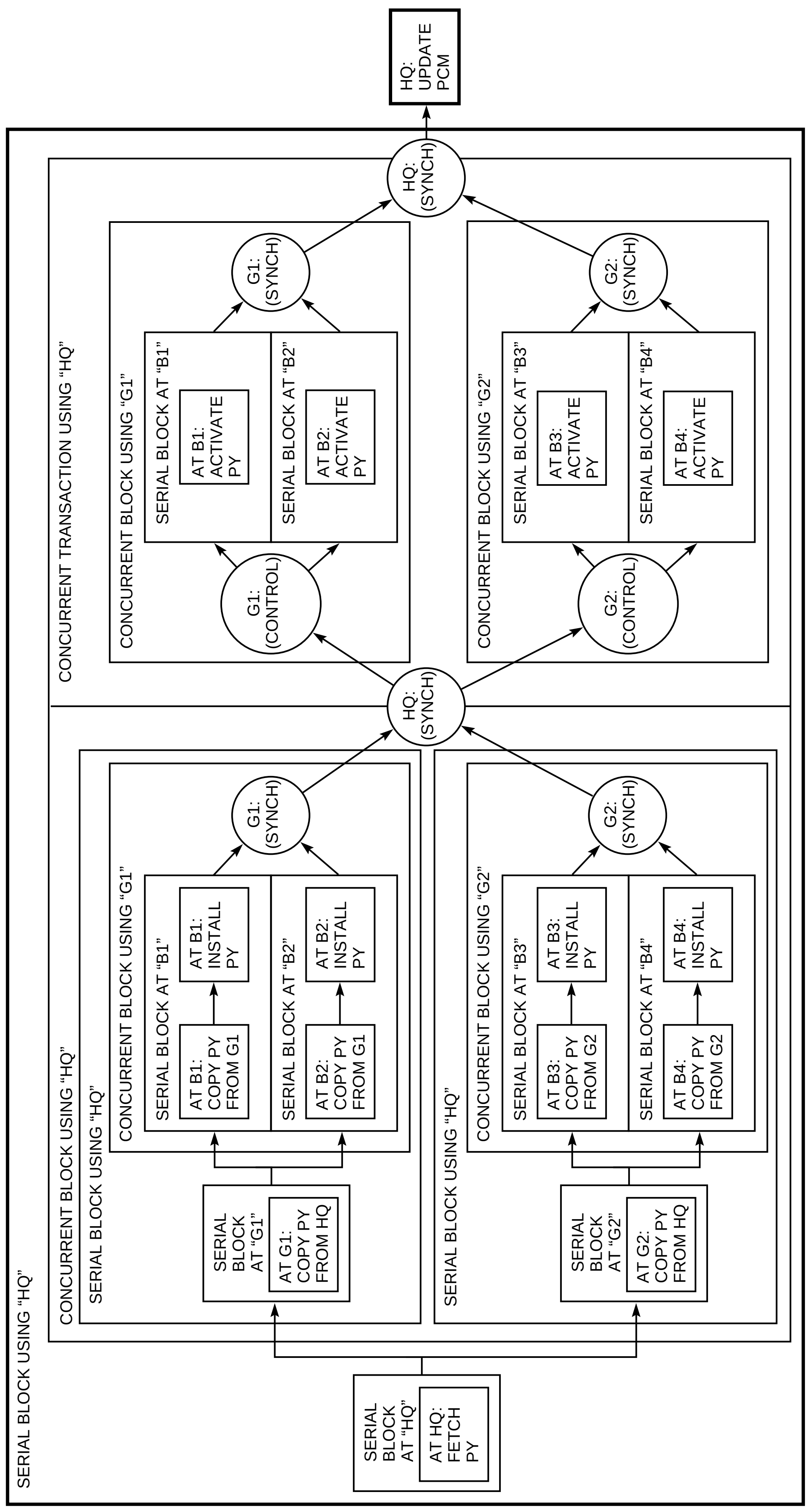

Figure 4
Realization Block Diagram

will be performed. Instead, an event message will be sent up the control tree to HQ. An operator can detect this condition (either as a result of the event message or during a periodic status check), intervene to correct the problem, and restart the action that failed. For example, if the copy action of PY to B1 from G1 fails, the first serial block at B1 will be suspended and the action to install PY on B1 will not be performed. (The install action follows the copy action in a serial block because it is dependent upon successful completion of the copy action.) The blocks in the first part of the deployment, i.e., the serial block at B2 and the concurrent block using G2, continue to execute, however. No processing will go beyond the first HQ synchronization point until the fault is corrected and the serial block at B1 completes. If the problem cannot be corrected, the deployment can be stopped and replanned, perhaps excluding the node that failed.

If one of the actions in the concurrent transaction fails, no additional actions within the transaction will be started and any that completed, including the failed one, will be rolled back. Each transition may have an associated ROLLBACK command. The rollback of an action consists of executing its ROLLBACK command. (This command is not shown in the SYREAL sample.) In this case, the ROLLBACK command deactivates PY. If the transaction has more activations, any that start before the failure is detected are rolled back in the reverse order of execution. The RLZ server effectively runs the transaction in reverse, from the point at which the failure was detected, executing the ROLLBACK command for each action that had completed. To accomplish this, each control point that detects a failure within a transaction or receives a rollback request from one of its subordinate control points initiates a rollback in all the parts of the transaction under its control. At the same time, the control point sends a rollback request to its control point. This process continues until the rollback request reaches the control point that controls the outermost block of the transaction.

A Note about Testing

Consider the challenge of testing a deployment system designed to operate over hundreds or thousands of systems. The PCM and CGN components are centralized, so load testing and boundary testing are relatively straightforward. Executing deployment operations is an inherently distributed process, however, with one RLZ server per host. The team designed the RLZ server to isolate all its data, e.g., network object name, log files, deployment program state data, and command procedures, based on the name given the server process. This design enabled the team to run as many copies of the server on a single system as the system's resources allowed—one VAXstation 4000 system was able to run more than 250 simultaneous servers—and to execute dummy command procedures. Such a design allowed the team to test elaborate simulated deployments and forced it to design the server to deal with a number of unusual resource shortages.

Project Gabriel's performance data indicated that the overhead of the RLZ server was relatively insignificant when compared with that of the actions performed by means of command procedures. This data supported the team's belief that the system would be scalable: A target system that has the resources to support relatively resource-intensive actions like software installations can support one RLZ server to automate the installations.

Conclusions

This paper does not cover topics such as the complex rules regarding the suspension/resumption and restart of operations, lost server time-outs, and interim status updates. Also, the PCM data is considerably more complex than the discussion indicates, as is the asynchronous processing implemented in the RLZ server and the logic of CGN.

A great deal of detail has been omitted in order to focus on the usefulness of a particular collection of abstractions in solving a difficult problem. The entity model and the configuration management model helped to define, partition, and communicate about the problem. The distribution model from the POLYCENTER Software Distribution advanced development work provided essential ideas that the other models did not. These intellectual assets were instrumental in fulfilling the customer's requirements. In "What Good are Models, and What Models are Good?" Fred B. Schneider asserts: "Distributed systems are hard to design because we lack intuition for them."[6] By formulating and analyzing an abstract model, we can develop such intuition, but it is a slow process. It is easy to underestimate both its difficulty and its value.

The model of distributed process control developed for Project Gabriel has proven useful and versatile. It could be profitably applied to the design of a process control service for distributed object technology, such as the Object Management Group's Common Object Request Broker Architecture (CORBA).[7] In such a design, instead of executing a command procedure to perform an action, a process control daemon would invoke a CORBA request on an object. Programs become nested collections of requests with associated state. Improving distributed object and object-oriented database technology should make possible fuller realization of the PCM and a more powerful CGN. The work accomplished in Project Gabriel just scratched the surface.

Summary

By applying relatively well-developed conceptual models for network and system management, Project Gabriel successfully implemented automated software deployment in a large commercial network. The result is a scalable, distributed system management application that can be used to solve a variety of complex distributed system management problems.

Acknowledgments

The following individuals contributed directly to the design and implementation of the Project Gabriel software: Cecile Beyh, Hugo Diaz, Gill Haik, Paul Kelsey, Michel Lazarescu, John Lundgren, Pat Madden, Mary Maling, Keith Noddle, Laura Spink, Son Voba, David Wihl, and Michael Young.

References

1. *Information Technology—Data Communications—X.25 Packet Layer Protocol for Data Terminal Equipment,* ISO/IEC 8208:1990 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

2. *Interface between Data Terminal Equipment and Data Circuit-terminating Equipment for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuits—Data Communication Networks: Services and Facilities, Interfaces,* Recommendation X.25-89 (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], 1989).

3. M. Sylor, "Managing DECnet Phase V: The Entity Model," *IEEE Networks* (March 1988): 30–36.

4. *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs,* MIL-STD-48A (Washington, D.C.: Department of the United States Air Force, June 4, 1985).

5. J. Rumbaugh, et al., *Object-Oriented Modeling and Design* (Englewood Cliffs, N.J.: Prentice-Hall International, 1991): 457.

6. F. Schneider, "What Good are Models and What Models are Good?" *Distributed Systems,* 2d ed., S. Mullender, ed. (New York:ACM Press, 1993): 17–26.

7. *Common Object Request Broker Architecture Specification,* draft 29, revision 1.2 (Framingham, Mass.: Object Management Group, Document No. 93-12-43, December 1993).

Biography

Owen H. Tallman
Currently employed by NetLinks Technology, Inc., of Nashua, New Hampshire, Owen Tallman worked at Digital Equipment Corporation from 1983 through 1994. As a principal software engineer in the Networked Systems Management Engineering group, he led Project Gabriel. He was a management information architect in the Enterprise Management Architecture group and helped develop the POLYCENTER Software Distribution product (formerly known as the Remote System Manager [RSM] product). Owen holds a B.A. in computer science from Regents College in Albany, New York, and is coauthor of two pending patents on RSM technology. He is a member of ACM.