Architecture Styles and Services: An Experiment Involving the Signal Operations Platforms-Provisioning Operations System

David G. Belanger
Yih-Farn Chen
Neal R. Fildes
Balachander Krishnamurthy
Paul H. Rank Jr.
Kiem-Phong Vo
Terry E. Walker

The process of designing vertically integrated applications is enhanced if the distinct architectures, or *architecture styles*, and relevant performance constraints and interactions can first be identified. Applications, although running in varied environments, also may require specific *architecture services*, non-operational features such as portability or fault-tolerance that might be common across several architectural styles. The application design process should be an iterative exercise of first understanding system requirements and then determining how they may be partitioned according to styles and services. An integral part of this process is to identify software components and subsystems that must be developed or can be reused from other systems. This paper describes a design-partitioning process applied to the new Signal Operations Platforms-Provisioning (SOP-P) operations system. The experiment shows that it is feasible to identify large design components confined within a few architecture styles that are common to network management and operations software.

Introduction

Developers of large-scale integrated software applications are always under pressure to increase a system's quality while reducing costs and development time. This is particularly true for the systems that provide network management and operations support (NMOS) for AT&T's World Wide Intelligent Network. As in other integrated applications, these systems are characteristically composed of subsystems that can be operations systems (OSs) themselves or generic components of other systems.

These individual systems perform a variety of different functions and have substantially different architectures. However, they typically have substantial similarities in the *types of processing* that they perform, thus providing an opportunity to take advantage of:

- Commonality of component systems to reduce the total number of such systems, and
- Commonality of the subsystems within the individual system architectures to reduce the effort to design, develop, test, and maintain them.

In the summer of 1994, an experiment was conducted to take advantage of these commonalities on a specific application, the Signal Operations Platforms-Provisioning (SOP-P) operations system. Since SOP-P replaces 12 older network management systems, this provided an opportunity to test a process for partitioning the architecture of a system in a specific large project.

Partitioning Into Styles and Services. The process applied to the SOP-P architecture has evolved from two areas of work.

Partitioning at the highest level. The *first* effort is the development of a process to take advantage of the commonalities of various system components and architectures at the highest level of design—referred to here as *architecture*—by partitioning applications into parts that have distinct requirements for *architecture styles* and *architecture services*.

An *architecture style* is a set of *operational characteristics* that are both common to

some family of software architectures and sufficient to identify that family. Architecture styles range in complexity from simple pipelines to complex on-line transaction and decision support systems (DSSs).

An *architecture service* is a *capability* that does not implement specific application features, but is essential to the application's usability. For example, operations, administration, and maintenance (OA&M) is a service required in some form by all large systems, but is not part of the feature sets of the systems. Portability, fault tolerance, and security are other services that are important aspects of software applications. These architecture services make up a substantial part of modern systems, require sophisticated techniques to implement, and often offer ideal opportunities for reusing software components at a high level.

Experience of the Architecture Review Board. The *second* work effort is the experience of the Architecture Review Board, operated by AT&T's Software Technology Center, in reviewing more than 200 AT&T projects. The information gathered during this effort indicates that most network operations systems are combinations of a few architecture styles and services.

Identifying the Styles and Services. Thus, not only can large operations systems be partitioned into a few distinct architecture styles and services, such a partitioning offers the following benefits:

- The fact that there are just a few styles and OS services narrows design and implementation decisions.
- The selection of reusable components is made easier.
- Partition translates into subsystem division and helps to divide and focus implementation.
- Up-front understanding of constraints and interactions among parts of the architecture reduces the chance of extensive modifications that often obscure original architectures.

The development of an architecture for SOP-P started with a week-long task force meeting to define a Tier 1 architecture for the system. A Tier 1 architecture is a partition of the system into subsystems and a definition of the overall interactions among the subsystems. This level of architecture should be abstract enough to apply to a family of related systems, and often leads to the high-level reuse of components and subsystems. The architecture is thus definable directly from the system requirements.

Panel 1. Abbreviations, Acronyms, and Terms

ACID — atomicity, consistency, isolation, and durability
DBMS — database management system
DSS — decision support system
DS —data stream, data streaming
FEP — front-end processor
NMOS — network management and operations support
OA&M — operations, administration, and maintenance
OS — operations system
SOP-P — Signal Operations Platforms-Provisioning
TP — transaction processor

This paper, which describes the process of architecture partition as applied to SOP-P, is organized as follows. The section "Distribution of Software Architecture Responsibilities" describes the general approach to architecture partitioning, while the section "Architecture Styles and Services for NMOS" describes architecture styles and services in more detail. The section "SOP-P: An Example of Style and Service Partitioning" describes the application of the architecture partitioning process to SOP-P. The last section contains our conclusions.

Distribution of Software Architecture Responsibilities

A software system is composed of components, some of which may already be available from other systems while others must be developed. Architecting is the first step in system design that partitions requirements in such a way that they can be mapped to the appropriate software components. A properly implemented architecting process should result in systems with similar requirements having similar partitionings. Such a process leverages the knowledge gained in past system design; simplifies the reuse of design, development, test, and maintenance processes; and enhances quality and productivity in constructing the software products.

Current Approaches. System architects often draw primarily from their own experiences and training to come up with an ad-hoc requirement partitioning. This is illustrated in Figure 1a where two hypothetical network

operations systems, A and B, have been partitioned in an ad-hoc manner into A1, A2, and A3, and B1 and B2, respectively. Since there are no guidelines, the result of this ad-hoc process varies with the experience and quality of the architects and, while innovative approaches sometimes evolve, experience has shown that the result is unpredictable and frequently results in radically different designs. In this case, none of the subsystems have any similarities.

In contrast, where platforms of reusable components are in use, the usual approach to system design is to constrain the requirement partitioning based on the *operational characteristics* of the available reusable components. This can lead to more consistent and stable designs of operationally similar systems.

Figure 1b illustrates this approach where systems A and B are partitioned arbitrarily into A1, A2, and A3 and B1 and B2, but because the partitioning is based on operational characteristics, A1 and B1 are similar enough to run on a common platform, and A2 and B2 also are similar. For example, they may be able to use the same database management system (DBMS) or transaction processor (TP) monitor.

This approach works well when the available platform components fit the architectural requirements of the application. However, it can falter for applications that require features from platforms that support radically different functionality. For example, a typical telecommunications provisioning system needs:

1. A high throughput flow of network-oriented data,
2. High-quality transactional interaction with a large collection of users, and
3. Data mining for decision and operations support.

Without an up-front understanding of conflicts between these requirements, even with good support from platform components the system may be designed and implemented in a way that would not meet performance requirements. In turn, this may lead to code patches that degrade the architecture and accelerate system "aging."[1]

Styles and Services as Architecture Components. Our approach to architecting systems starts with analyzing the domain of an application—that is, identifying common characteristics of applications in the same class—to determine the common architecture styles and services. Then, these styles and services are used to constrain the design partition of a given system based on *behavioral properties*. This is made clearer below.

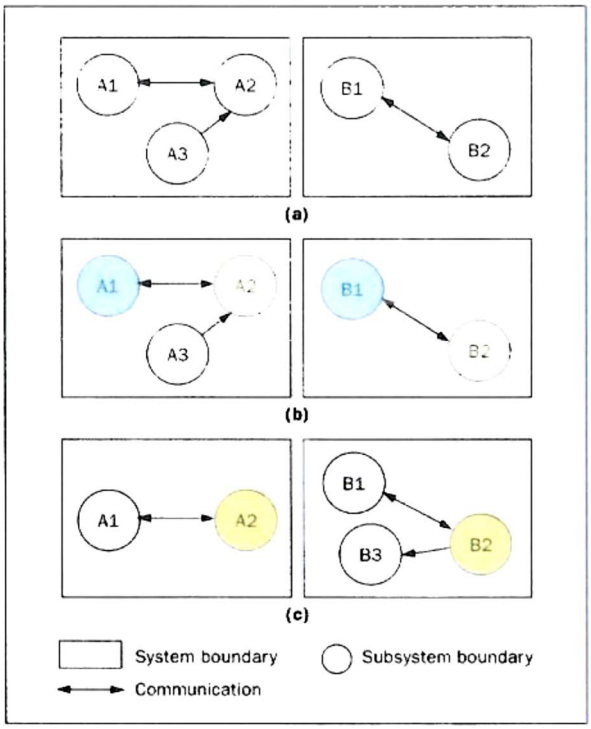

Figure 1. In (a), two hypothetical network operations systems A and B have been partitioned in an ad-hoc manner into A1, A2, and A3, and B1 and B2, respectively. The result of this ad-hoc process varies with the experience and quality of the architects. In (b), systems A and B are again partitioned arbitrarily, but because the partitioning is based on operational characteristics, A1 and B1 are similar enough to run on a common platform, as are A2 and B2. In (c), partitioning reveals that subsystems A1 and B1 express a single architectural style, even though they are in different systems, as does A2 and B2. Since the style-preserving relationships—A1 to B1 and A2 to B2—can be clearly identified, the choice of suitable platform components now largely hinges on the styles that they express.

Software components are recognized by their operational characteristics. Here we shall include *performance levels* as operational characteristics, even though they are

not strictly operational in nature. This inclusion is justified because the systems built in NMOS have strict performance requirements that often dictate what operations can or cannot be done. For example, a TP system performs operations with the following so-called ACID characteristics:[2]

- Atomicity, which means the transaction either succeeds or fails completely;
- Consistency, which means the transaction is a correct transformation of the state and its actions do not violate any of the integrity constraints associated with the state;
- Isolation, which means the transaction works as if no other transaction exists; and
- Durability, which means that once a transaction completes successfully, the changes made to the state during the transaction will survive any subsequent system failures.

Conversely, if a system always operates with these characteristics, then it must be a TP system.

Characteristics that identify a common collection of systems give rise to the notion of architecture style. More detailed descriptions on architecture styles can be found in Garland and Shaw[2] and Perry and Wolf,[3] while Gamaa et al.[4] report on the related work on design patterns that identify common data and program structures. A set of frequently asked questions and answers about domain-specific software architectures can be found in Tracz.[5]

Beyond operational features, the usability of an application also may depend on other non-operational features such as portability, fault-tolerance, and security. In recent years, much progress has been made in building software components that provide such features and that can be used transparently in large classes of applications. These components are orthogonal to architecture styles.[6] For example, the `libft`[7] software library provides facilities such as automatically saving data and process states at run time, called checkpointing, and restarting failed processes based on the saved states. This enables software fault-tolerance at a process level without any process modification. The existence of such components gives rise to the notion of architecture service.

Note that styles and services are defined by characteristics. Therefore, a particular style or service may encompass different software components that provide the same set of functions but with different performance characteristics or environmental constraints. A subsection below, "Guidelines for Choosing Styles," discusses a way of using the operational and non-operational characteristics of styles and services to identify major parts of the overall architecture of an application.

Using the same example in the last section, Figure 1c shows how design partitions are mapped to corresponding reusable components with clear subsystem boundaries. Here again are two systems, A and B, partitioned into A1 and A2 and B1, B2, and B3. Subsystems A1 and B1 express a single architectural style, even though they are in different systems, as do A2 and B2. Since the style-preserving relationships—A1 is equivalent to B1 and A2 is equivalent to B2—can be clearly identified, the choice of suitable platform components now largely hinges on the styles that they express.

For example, if A1 is TP, B1 also is TP and may use the same platform. Further, as these components share similar behavioral requirements, they may also be able to share implementation. In similar veins, A2 and B2 may be front-end processors (FEPs), while B3 is a DSS. The intuition behind this approach is quite straightforward. Since the related subsystems in A and B share similar behavioral characteristics, they can be supported by common components and the constraints on the evolution of their design also will be well-defined.

Hybrid Applications. Once an architecture has been determined to have multiple styles and is partitioned appropriately, two factors greatly influence the quality and performance of the final product: *style separation* and *style interoperability*.

Style separation. The exercise of identifying and partitioning an architecture along style boundaries may not in and of itself be enough. In instances where multiple styles compete for computing resources, separation of styles may be a necessity. Style separation can take place in two forms: *temporal separation* or *physical separation*.

Temporal separation occurs when multiple styles are separated into distinct phases of operation. This implies that no one style will be competing simultaneously with another style for computing resources. Thus, at any time, only one style of an architecture partition will be active.

Consider, for example, a system that collects large amounts of data during the night and then puts it in

a database to be analyzed by the engineering staff the next day. Such a system would use a data stream (DS) style to collect and process the data, and then a DSS style for the engineering staff. If there is sufficient time to collect, process, and load the data prior to the start of the working day, these styles can be effectively separated.

On the other hand, if data is still being collected and loaded into the database after the first engineering requests are made, performance problems may occur due to the different styles competing for the same resources.

Physical separation is accomplished by providing separate computing resources—such as processors and memory—when multiple styles must be active simultaneously. This should be necessary only when the partitioned styles require significant resources and cannot be separated temporally.

Consider again the above example. If the data were collected and processed 24 hours a day, the DS style process would always be competing with the DSS style process. In this case, it may be necessary to apply separate computing resources to each style.

Style Interoperability. Regardless of partitioning styles along time and physical boundaries, experience has shown that style interoperability is directly related to application integrity—and thus maintainability—and unintended degradation of an architectural system. In order to maintain both, special attention should be given to how one particular style communicates with other styles. In general, the initiating style should communicate in a manner the receiving style is prepared to receive.

For example, a DS style process should communicate with a TP style process via a transaction in the form inherent for that transaction style. The DS style process should not send the TP style data in a form for bulk loading as it would to a DSS style process.

Guidelines for Choosing Styles. In general, the first step in defining an architecture is to state or determine the technical problem to be solved. Next, to begin the style identification and partitioning, key characteristics of the technical problem should be determined.

The operational and non-operational characteristics of styles and services can be used to design a set of criteria and questions that help architects to identify major parts of the overall structure of an application in terms of such styles and services. Let *S* stand for a subsystem and *c* stand for some characteristic. An investigation along the following line can be used to help define or refine subsystem boundaries:

If <*S*> exhibits/requires <*c*> then <*indicators*>.

Here, an indicator is a pair (a *style* or *service* and its *value*). The *value* part of such a pair indicates the applicability of the respective style or service with three values: *-1* for not applicable, *0* for neutral, and *1* for applicable. For example, a requirement of access restriction means that a security service is needed but it has little bearing on the requirement for reliability. Thus, if <*S*> requires <*access restrictions*> then <*security, 1*>; but if <*S*> requires <*reliability*> then <*security, 0*>.

Once the key application characteristics are known, they can be matched with the appropriate style characteristics. The general steps to complete the architecture definition are:

1. Identify the system requirements.
2. Use the developed questions to identify indicators to relevant styles and services.
3. Divide the architecture into regions to fit these styles and services.
4. Model the architecture as a set of communicating regions.
5. Identify needed components from platforms that either exist or that need to be developed.
6. Determine the feasibility of the design based on style, service, and component constraints.
7. Repeat steps 1 and 3 if appropriate to refine the process.
8. Build the system.

Note that step 6 is greatly helped by the indicators for the relevant styles and services found in step 2. For example, certain performance requirements may imply that a subsystem should not simultaneously exhibit characteristics of the two styles, TP and DSS. This does not necessarily mean that such a subsystem is not buildable, but some means may be necessary so that the operations from the two different styles do not overlap. Such means could be:

- Operation scheduling, a form of *temporal* separation as already discussed in the subsection "Style Separation," or
- The duplication of data, a form of *physical* separation.

Otherwise, certain performance requirements may not be met. In turn, this realization may require a redesign step.

Architecture Styles and Services for NMOS

This section takes a closer look at a few architectural styles and services that are of importance to NMOS.

Transaction Processing. Each transaction processing system may consist of one or more databases, with concurrent read and write transactions enacted against them. Transaction managers ensure that each transaction meets the ACID test, that is, it is atomic, isolated, consistent, and durable.

Decision Support System. A DSS may consist of one or more databases. Data typically gets into the databases by diverse means, such as bulk-loading from TP databases or from tapes, and a lack of atomicity is permitted. A DSS produces high-level and summary analyses from such data. Since the incoming data may have diverse formats and the analysis tools may require different output formats, a major part of such a system is components that transform data from one format to another.

Data Streaming. DS consists of a component that accepts typed data from a sender, processes the data in a buffer, and then sends the results to a receiver. Certain records may be retained as a side-effect of processing the data that go through the buffer. A typical requirement for DS is very fast processing of streams of data objects.

Front-End Processing. An FEP system is needed whenever the user is to be presented with a restricted set of menus of choices specific to the application. The FEP is responsible for creating a user-friendly framework that avoids user errors, retains entered information for later use, and prevents the user from accidentally modifying data that is essential to the system's functionality.

Each of the styles is reasonably well understood by practicing architects. Some, such as TP, have been exhaustively studied and described.[8] Others, such as DS, have been less studied in the general literature but are well understood within the network operations software business. For the purposes of this paper, it is not necessary to define the styles in great detail, that is, enough to build a platform. It is only necessary to understand a style well enough to be able to ask the correct questions of the requirements, both feature and non-feature, to partition a system at a high level.

The SOP-P experiment has provided useful information on the form of these questions, and revealed many of the questions that would be useful to ask in creating a new NMOS application. Considerably more experience will be needed, however, to create a set of questions that could be used routinely in the software architecture process.

The architecture styles that have been identified need to be supported by the architecture services necessary for the non-functional requirements of a system. The technology required to provide these services has become complex and multi-leveled in its own right. As a system evolves and new levels of service become necessary, the levels should be provided by extending the services of the platform, rather than altering the application-specific code.

Vital Operations Systems Services. Several services that are vital in the domain of NMOS are provided by tools and libraries designed and built by subject matter experts. These services, which find broad applications in many architectures, are:

Communication service. This service allows clients and servers in a distributed system to send data and communicate with each other through predefined protocols.

Alarm service. Alarm service logs and manages the faults that have been detected by the system. This service may include the analysis and filtering of fault data and the configuration of the fault detection mechanisms. Some of the capabilities may be provided by the system vendor such as in fault-tolerant hardware.

Audit service. This service audits the use of the system resources. It may include monitoring and controlling user-initiated load, system memory, system files, communications channels, and computing resources. The data is frequently used in performance management functions.

Resource management service. Resource management is the administration of system resources that are not included in the other functional areas. Included in this area are hardware maintenance, system accounting, system-level configuration, backup and recovery management, software migration capability, system process management, and user login and password administration.

Security service. Security service includes the functionality for monitoring and auditing application and system activity, producing logs on unauthorized access attempts, reporting on security-related activity, and configuring access permissions on a user and application level.

Visualization service. This service provides the ability to graphically view the structure or the behavior of a system, by interacting through a pictorial representation. This capability is useful in understanding, debugging, and controlling application systems.

SOP-P: An Example of Style and Service Partitioning

The SOP-P is an NMOS that supports provisioning for the AT&T signaling network. SOP-P is a multiphased development with the first phase currently in production. The SOP-P's primary role is to generate update messages that are sent to the network elements to modify the network signaling plan. The originator of these messages, from a SOP-P point of view, is either an upstream system or a network administrator.

Another system requirement is to gather data from the network and either provide the data to upstream systems or store the data for future analysis. There also is the need for general services such as auditing all transactions, restricting users to only those actions they are authorized to carry out, and triggering alarms on fault conditions.

Before discussing the styles within SOP-P, a word is in order about the platform tools being used to construct it. SOP-P has used object-oriented technology from analysis to implementation and is using the AT&T product LibOS, a package supporting portable object modeling for event-driven systems. The means of communicating between objects is a set of routines called LibE. These routines provide a structured messaging facility which, for each object, triggers an appropriate message, based on message type. The user interface is developed using standard tools such as Visual C++ and Visual Basic with a LibE library provided to interface with the host servers.

SOP-P styles. The requirements of SOP-P indicate that its primary style is DS. In fact, a look at the SOP-P architecture in Figure 2 shows how the data moves through SOP-P towards the network elements and also how data is extracted from the network elements. The data is either transferred to other systems or ends up residing in a local database for analysis. A number of common characteristics of a DS style also are exhibited.

- Data comes from one or more sources and is delivered to one or more destinations.
- The distribution of data can be scheduled.
- The input data may require expansion or contraction based on mapping information and may result in more than one piece of data going to multiple destinations.
- The output data format may depend on the type and software version of the network element receiving it.
- The protocol used to talk with external devices may differ from network element to network element.

The net result is that data continues to move across the architecture—at times being combined with other data, at times being modified in content, and at times being held up for an appropriate time before moving on—but it ultimately makes it to its final destination. In the SOP-P implementation, LibOS repositories are used to hold these data items with their associated transport methods as they move across a machine boundary. LibOS also contains the objects that manipulate these data items.

While DS is the primary style used in SOP-P, the other styles also are used. The FEP style is used for the user interface, and access into the system is provided by menus and forms with strict rules controlling data entry. When a user inputs information into the system, it might be a data item to be sent to one or more network elements. In this case the first step, the flow of data from the user interface, uses the FEP style before it moves to the second step, the DS style.

On the other hand, if the user input is used to modify one of the LibOS repositories whose objects are used to manipulate data items, then this becomes a TP style. And finally, if the data entered by the user is interpreted as a command to analyze data in a repository, then this is using the DSS style. A particular example of this is when the user would like to have a particular view of the data expressed in a graphical user interface representation.

While all four styles are used, they are clearly differentiated and care has been taken to minimize any potentially adverse results when these styles interact, as described in a previous subsection ("Style Interoperability").

SOP-P services. A number of services required by SOP-P are used by all parts of the architecture, independent of the style being used. The basic communication services are provided by the LibE library. However, three other services were developed as LibOS repositories with associated libraries containing functions to be used by the application clients and servers.

The first of these is the *audit server*, a LibOS reposi-

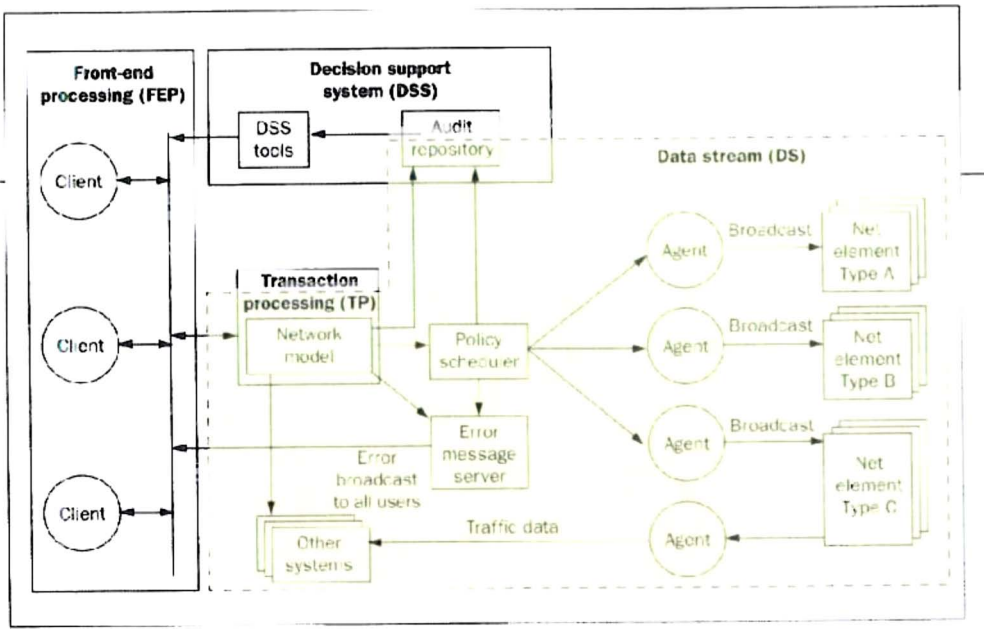

Figure 2. The SOP-P architecture is shown with style partitioning that yields four distinct styles—front-end processing (FEP), decision support system (DSS), transaction processing (TP), and data stream (DS). The data moves through SOP-P towards the network elements and also is extracted from the network elements. The data is either transferred to other systems or ends up residing in a local database for analysis.

tory for objects that record user actions such as connection attempts and object creations. The objects are collected from all servers and stored for a specific time interval, and performance and network change history reports can be generated using the objects. Additionally, a mechanism is provided for archiving older objects to off-line storage and then purging them from the repository.

The second is the *alarm server*, a LibOS repository for alarm objects. An application server needs only to create an alarm object; populate its data members, that is, structure fields, with information about the alarm; and "install" it in its disk storage space. The object is automatically migrated to the alarm server where it may be saved or analyzed and determined to be a duplicate. If the object indicates the condition has cleared, the master object in the server will be discarded.

The third is the *security server*. SOP-P uses an object-oriented permission mechanism, in which each user is assigned a specific role and SOP-P maintains, for each role, a list of the classes a user is allowed to interact with, as well as the operations or methods of each class. For example, a network element administrator may be allowed to make changes in network elements, but not to the network model database. The security server is the official repository for the objects that are used to control access to SOP-P. For example, it holds an object for each user, role, and server in SOP-P, as well as the class or method pairs appropriate for each role.

Each of the other servers in SOP-P maintains a synchronized copy of the security objects they need via a distribution object mechanism. In the event that communications with the security server is lost, the various servers would continue to function with their copies until they are reconnected to the security server. When the security administrator adds or modifies one or more of the security objects in the server, new copies or deletion notices are broadcast to the other servers.

Using the SOP-P. A user logs onto the system using the FEP's login screen and, based on the role of the individual logging in, the security server assigns the set of objects with which the user is allowed to interact. One way of applying this assignment is to present to the user's screen only those operations assigned to that user. If, for example, this user is not authorized to add a network element, the screen would not display that function. For the rest of this example, we will assume that the tasks discussed are included in the user's role.

If the user wants to change a table entry in a network element, this would be within the TP style. If the user wants to analyze the audit log to determine the frequency of occurrence of a certain type of action and then correlate these actions with the user's logged on, this would be in the DSS style. But the most frequent user activity would be associated with data movement to and from the network elements.

For example, if the user would like to add a link between two network elements at a specified time, the new information required to accomplish this would be entered using the appropriate forms, such as an FEP. The appropriate objects would be launched from the user client and start it's DS journey, and the appropriate audit object would be sent to the audit server.

To accomplish the task, a number of messages will be required to be sent to multiple network elements. This message expansion occurs in the policy scheduler. There are usually sequencing rules associated with the order in which the data messages are to be sent to the network elements. To accomplish this, the policy scheduler has a scheduling function embedded within it. At the appropriate times it sends the data messages to the distributor servers, which provide the correct format and protocol to allow them to be sent to the appropriate network element. The audit server would be notified when the data is transmitted to the network element. If the connection to the network element fails, an alarm object is sent to the alarm server.

Conclusions

We have described the role of architecture styles and services in building large integrated applications. The paper is based on a large study done internally as well as our direct participation in the partitioning of the SOP-P architecture. Although the work is concentrated on a single domain—that of NMOS systems—the techniques for partitioning the architecture into architectural styles and services can be generally applied to other applications. Architectural styles can span multiple domains but their usefulness and applicability will vary according to the domain. For example, a TP style is not likely to be very useful in a domain where there is no interest in concurrent updates of durable records.

There is little in the literature in terms of experiments or case studies about applying architectural styles and services to building industrial-strength software. In building large-scale software, a certain amount of architectural degradation can be expected to take place. The approach we offer is intended to control the drift by constraining the choices for initial system partition and, within the components of the partition, constraining the architectural elements that can be used.

The evidence so far is promising. It confirms a number of hypotheses, including:

- Intelligent partitioning into architectural styles and services can be done directly from requirements.
- Such a partition leads to better understanding of constraints on components of the system, and to an architecture that remains stable.
- The nature and specificity of the questions asked during the partitioning process has a substantial impact on the system requirements.
- Though we found several systems which had evolved to this type of partition, it was not the first proposal for an architecture. In fact, for SOP-P, the partitioning process produced a different partition than what the architects originally envisioned.
- This approach does support the effective use of platforms.

On the other hand, this approach raised some questions for further investigation, such as:

- How long into the system's life cycle will the partition's impact last?
- Is there a natural way to determine inter-component interactions based on the style and service partition?
- To what extent can an algorithmic process for partitioning be developed?

Clearly, the current process requires a certain amount of art in its application. Our next steps are to investigate other system projects and to make the process easier for any architect to execute.

Acknowledgements

We thank all the architects who participated in the field study. We also thank Dave Thomas for suggesting the FEP style during the study; Jackie Burke for supplying us with SOP-P project data; and Nick Landsberg for his very useful study of the various operations systems architectures based on the work of the Software Technology Center's Architectural Review Board.

References

1. M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, San Diego, California, 1985.
2. David Garlan and Mary Shaw, "An introduction to software architecture," in V. Ambriola and G. Tortoara, editors, *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, World Scientific Publishing Co., River Edge, New Jersey, 1993, pp. 1-39.
3. D. E. Perry and A. L. Wolf, "Foundations for the Study of Software

Architecture," *Software Engineering Notes*, Vol. 17, No. 4, October 1992, pp. 40-52.
4. Erich Gamaa, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Massachusetts, 1995.
5. William Tracz, "Domain specific software architecture: Frequently asked questions," *Software Engineering Notes*, Vol. 19, No. 2, April 1994, pp. 52-56.
6. Balachander Krishnamurthy, ed., *Practical Reusable UNIX Software*, John Wiley & Sons, New York, New York, 1995.
7. Yennun Huang and Chandra Kintala, "A software fault tolerance platform," *Practical Reusable UNIX Software*, edited by Balachander Krishnamurthy, John Wiley & Sons, New York, 1995.
8. Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, California, 1993.

(Manuscript approved January 1996)

David G. Belanger *is head of the Communication Information Systems Research Department and the Information Engineering Laboratory at AT&T Bell Laboratories in Murray Hill, New Jersey. His department is responsible for research and the transfer of technology and concepts that create and improve AT&T systems that move, manipulate, analyze, and display information. Mr. Belanger joined the company in 1979. He has a B.S. degree in mathematics from Union College in Schenectady, New York, and M.S. and Ph.D. degrees in mathematics from Case-Western Reserve University in Cleveland, Ohio.*

Yih-Farn Chen *is a member of technical staff in the Communication Information Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include data modeling, reverse engineering, and software repository. He joined the company in 1987. He has a B.S.E.E. degree from the National Taiwan University in Taipei, an M.S. degree in computer science from the University of Wisconsin-Madison, and a Ph.D. degree in computer science from the University of California-Berkeley.*

Veal R. Fildes *is a distinguished member of technical staff in the Operations Technology Center of AT&T Network Services Division in Cincinnati, Ohio. He is a member of the architecture team for signaling provisioning for the SOP-P project. Mr. Fildes joined the company in 1977. He has a B.S.E.E. degree from Cornell University in Ithica, New York, and an M.S.E.E. degree from Carnegie-Mellon University in Pittsburgh, Pennsylvania.*

Balachander Krishnamurthy *is a member of technical staff in the Software Engineering Research Department of AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include UNIX tools, event action systems, software reuse, and software architectures. Mr. Krishnamurthy joined the company in 1988. He has a B.S. degree in mathematics from the University of Madras in India, an M.S. degree in mathematics and operations research from the College of William and Mary in Williamsburg, Virginia, and M.S. and Ph.D. degrees in computer science from Purdue University in West Lafayette, Indiana.*

Paul H. Rank Jr. *is a technical manager in the Operations Technology Center of AT&T Network Services Division in Cincinnati, Ohio. His group is responsible for developing the SOP-P project. Mr. Rank joined the company in 1967. He has B.S. and M.S. degrees in mathematics from Stevens Institute of Technology in Hoboken, New Jersey.*

Kiem-Phong Vo *is an AT&T Bell Laboratories Fellow and distinguished member of technical staff in the Communication Information Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include graph theory and discrete algorithms and their application in reusable software. He joined the company in 1981. He has M.A. and Ph.D. degrees in mathematics from the University of California at San Diego.*

Terry E. Walker *is a distinguished member of technical staff in the Operations Technology Center of AT&T Network Services Division in Cincinnati, Ohio. He is an architectural consultant for operations support systems. Mr. Walker joined the company in 1984. He has a B.S. degree in computer science from Eastern Kentucky University-Richmond.*