Software Architectures: An Introduction

The complexity of industrial-strength computer-based systems has compelled system engineers to use higher levels of abstraction when addressing the issues of system design. It has also made the overall system organization—software architecture—an important aspect of system development. This has implied a systematic approach to the engineering and application of architectures and has led to the rapid development of architecture technology.

B. Boar has observed that a technology typically evolves from being a craft to an engineering discipline over time with the infusion of scientific theory and the need for broad application.1 As a young discipline, the early architectural efforts were characteristic of the craft stage, during which a system architecture is created from scratch, relying on the designer’s personal experience, knowledge, and intuition. With the mass market involved, the architecture discipline has now reached the economics-driven commercial stage, characterized by the introduction of architectural standards, reusable reference architectures, and domain-specific software architectures. With more maturity and the presence of increased system complexity, software architecture is likely to assume the characteristics of a professional engineering discipline, characterized by underlying theory, architectural support tools, standardized practices, licensed professionals, and a community of shared expertise.

Software architecture is principally concerned with the study of patterns of system organization, large-grained software components, their relationships, and the models of interaction between them. It addresses the overall system properties at high levels of abstraction and from multiple perspectives, such as structure, control, and data. Architectural designs also address important system characteristics, including scalability, overall performance, processing rates, and allocation of functionality to design elements.2-4

For years, system engineers have used software architectures to describe the top-level design of computer-based systems. Typically informal, these descriptions include box-and-line diagrams representing system structure and often a free-form text clarifying the meaning of the diagrams and capturing some design rationale. Even being informal, such descriptions provide a critical first view of the solution and a starting point for determining whether a system can meet its essential requirements. They help developers construct the system; they also help management organize the entire project. Unfortunately, such archi-
Architectural designs cannot be analyzed for consistency or completeness and often they are poorly interpreted throughout the lower levels of the development cycle. Furthermore, as the system evolves, architectural constraints become difficult to preserve and enforce.

Recently, software architectures have also been employed for codifying and reusing design knowledge. These types of software architecture—referred to as reference architectures—are bodies of high-level domain-specific design knowledge, references, and standards that provide generalized solutions for classes of problems. Reference architectures—which include architectural styles, domain architectures, product-line architectures, and architectural frameworks—are designed to be widely reused, to serve for long periods of time, and to support a number of development and integration efforts. They are expected to serve as bases for reuse, maintenance, planning, and evolution support. The broad scope of reference architectures, however, and the lack of suitable methodologies and tools have created problems with their description, verification, and application.

To address these problems in today’s software architecture technology, researchers and practitioners are working to transform the practical experience of designing (complex) software systems into a comprehensive, well-founded design discipline. The objectives of this work are to:

- Establish a common conceptual basis for describing software architectures;
- Provide formal semantics of architectural representations;
- Develop rigorous techniques and tools to support the definition, representation, analyses, and application of software architectures; and
- Codify design experience so it can be successfully reused.

This paper summarizes the results of a case study in which a newly developed method for software architecture engineering and representation, GARM-ASPECT, was applied in an industrial environment. This method is based on the generic architecture reference model (GARM) and the architecture description language (ASPECT) derived from it. The case study had two main objectives: (1) to define an architecture—the Directory System Architecture (DirSA)—that meets the needs of unified directory services for an overall enterprise solution of integrated business communication systems, and (2) to represent the architecture using the ASPECT architecture description language.

Figure 1 illustrates how the remainder of this paper is organized. The section immediately below introduces the GARM-ASPECT method, and the subsequent section presents an overview of the directory

Panel 1. Abbreviations, Acronyms, and Terms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>algorithmic simplicity</td>
</tr>
<tr>
<td>ASPECT</td>
<td>architecture specification technique</td>
</tr>
<tr>
<td>ASPF</td>
<td>avoidance of single point of failure</td>
</tr>
<tr>
<td>CDI</td>
<td>consistent directory information</td>
</tr>
<tr>
<td>DAP</td>
<td>directory access protocol</td>
</tr>
<tr>
<td>DI</td>
<td>data integrity</td>
</tr>
<tr>
<td>DirSA</td>
<td>Directory System Architecture</td>
</tr>
<tr>
<td>DISP</td>
<td>directory information shadowing protocol</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Naming Service</td>
</tr>
<tr>
<td>DOP</td>
<td>directory operational binding management</td>
</tr>
<tr>
<td>DSA</td>
<td>directory service agent</td>
</tr>
<tr>
<td>DSP</td>
<td>directory system protocol</td>
</tr>
<tr>
<td>DSQLI</td>
<td>Directory Structured Query Language interface</td>
</tr>
<tr>
<td>DUA</td>
<td>directory user agent</td>
</tr>
<tr>
<td>DUAL</td>
<td>directory user agent library component</td>
</tr>
<tr>
<td>DUI</td>
<td>directory user interface</td>
</tr>
<tr>
<td>ECS</td>
<td>enterprise communication systems</td>
</tr>
<tr>
<td>ES</td>
<td>extended services</td>
</tr>
<tr>
<td>GARM</td>
<td>generic architecture reference model</td>
</tr>
<tr>
<td>IP</td>
<td>Internet protocol</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LCAD</td>
<td>low cost of additional development efforts</td>
</tr>
<tr>
<td>LCLPM</td>
<td>low cost of legacy product management</td>
</tr>
<tr>
<td>LDAP</td>
<td>lightweight directory access protocol</td>
</tr>
<tr>
<td>OSI</td>
<td>Open System Interconnection</td>
</tr>
<tr>
<td>PF</td>
<td>performance/real-time communication protocol</td>
</tr>
<tr>
<td>RPC</td>
<td>remote procedure call</td>
</tr>
<tr>
<td>SPA</td>
<td>single point of administration</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TCP</td>
<td>transmission control protocol</td>
</tr>
</tbody>
</table>
problem domain. Directory-related standards are discussed next, followed by the architectural solution. The last section summarizes the results and presents the future outlook.

The Method: An Introduction to GARM-ASPECT

This section introduces the GARM-ASPECT method and explains how it addresses a number of important attributes of software architectures. Fundamental to the method is GARM, which guides the process of software architecture engineering. It identifies a number of architectural views and addresses their relationships. It also clarifies the distinction between reference architectures on one side and specific system architectures on the other and enables the reuse of architectural knowledge. As the underlying conceptual basis of the ASPECT language, GARM defines a set of modeling concepts, their relationships, respective semantics, and general rules for how to combine them into specifications for software architecture.

ASPECT is an architecture description language. It provides a normalized vocabulary for expressing architectural structures and establishes a framework for expressing architectural constraints as architectural rules. In addition, ASPECT supports mechanisms for extending the structural view with auxiliary nonstructural information, such as behavior or quality specifications, expressed in other languages. In this respect, ASPECT provides facilities to establish relationships between different specifications and to pull them together into an overall architectural description.

This approach allows the methodology to be flexible and easily adaptable to the specifics of a domain or a family of systems. The common structural properties are described in ASPECT, while the set of auxiliary specifications and the corresponding representation languages can be selected to fit the purpose of the architecture, the class of systems being addressed, or the domain traditions. Together, all the specifications contribute to a comprehensive architectural description.

The auxiliary, nonstructural specifications are processed by external tools, which can be integrated with an ASPECT editor. For example, the ArchE editor was integrated with the uBET tool set to evaluate behavioral specifications captured in message
sequence charts.10,11 (*ArchE* is a Web-based architecture engineering tool, developed by the Cross-Product Architecture Department of Lucent Technologies.)

ASPECT is built on a set of constructive concepts, or *architectural elements*, that form its core ontology. The architectural elements include components, contracts, interfaces, ports, roles, scenarios, and architecture. ASPECT provides representations for all these elements. **Figure 2** illustrates the element types and their relationships.

*Components* represent the computational entities and data of a system.6,8 Typical examples include such components as clients and servers, objects, data capsules, and filters. From a structural point of view, ASPECT distinguishes between *building blocks*, which are atomic units in the context of an architecture, and *clusters*, which are composed of building blocks and/or other clusters and contracts.

From an operational perspective, components are characterized by functions, data, observable behavior, type, and quality attributes, all defined in the component’s *interfaces*. Multiple interfaces can be associated with a component to reflect different possible functional aspects, where each provides a partial external view of a component. Interfaces comprise *ports*—access points at which interactions between a component and its environment take place. The component’s body, which “implements” the interfaces, provides the functionality and behavior observed at the component’s interfaces. In the case of a building block, the body is an external *representation*, outside the domain of ASPECT. For instance, any text, graphics, or source code representing the algorithms and data structures behind the building block’s interfaces are considered external representations. In relation to ASPECT, these representations are auxil-
iary specifications and are treated as annotations. In the cluster case, the body is a \textit{composition} represented by an internal structure—de facto, an architecture—and a set of associations between internal and external ports. The ports of the cluster, resulting from the composition, are referred to as external ports, while the ports of the components participating in the composition are considered internal ports.

\textit{Contracts} represent interactions among components.\textsuperscript{8,12} They correspond to the lines in the box-and-line diagrams used to informally represent software architectures. Examples of contracts include procedure calls, pipes, Structured Query Language (SQL) links, and communication protocols. Contracts can be primitive or composite (aggregations of components and contracts). The external view of a contract is represented by a number of \textit{roles}, which prescribe the behavior of the interaction points participating in a “contract.” These roles serve as placeholders, which are later replaced by matching ports when a system is constructed. The interaction channel established between the roles is represented by the contract’s \textit{liaison}, which bundles together two views: an operational view and a structural view. The operational view captures the interaction protocol, whereas the structural view reflects the internal structure of composite contracts.

In ASPECT, a \textit{scenario template} is used to represent static and dynamic configurations of components and contracts. Static scenarios are configuration patterns that represent system topology, whereas dynamic scenarios typically represent use cases.\textsuperscript{13}

To support the description of architecture types, ASPECT provides an \textit{architecture template} that incorporates all architecture-specific elements and their constraints into an overall specification.

\subsection*{The Problem Domain: An Introduction to the Directory Project}

The Directory Project focused on developing a coherent enterprise directory system architecture to support common directory services across multiple communication products and also to remedy problems with a number of existing directory implementations. The architecture had to introduce a standards-based directory and prescribe ways to integrate it with communication servers and applications from the domain.

Directories essentially act as repositories of network user and resource information. Consider, for instance, the Domain Naming Service (DNS), a well-known, hierarchical, replicated naming service on which the Internet is built. Although DNS is the backbone directory system of the largest data network in the world, it is not flexible enough to act as an enterprise directory. DNS is largely a service for mapping machine names to Internet protocol (IP) addresses. A fully functional directory service provided by an enterprise-capable directory system must be able to map names of arbitrary objects (such as users, machines, applications, systems, and services) to any kind of information about those objects.\textsuperscript{14} It must also allow for locating diverse types of entities, from users to systems and services.

Today’s enterprise communication systems (ECS) comprise different products, such as switches, messaging systems, multimedia communication support systems, and Internet applications (for example, voice and fax over the Internet). Most of these products use some type of directory, each supporting a different subset of directory-related functionality. There are, however, numerous problems with the existing solutions.

First, these directories cover different populations; they also have different attributes and constraints, as well as different, sometimes inconsistent, user and administration interfaces. They do not meet customer requests for unified and possibly single-entry administration across multiple products operating on their premises. This situation is further aggravated by the need to integrate “closed” products from different vendors, as well as by the increasingly quick introduction of new products that add yet another directory of their own.

Second, directories are currently perceived as just a means of recovering phone numbers, addresses, and—possibly—account information. This is a serious understatement of the potential utility and power of directories. When properly designed and implemented, a directory becomes an information storage and retrieval system, outfitted with a powerful query-based interface. As such, a directory can process all sorts of information pertinent to an organization and
its business operations. As multimedia and collaborative communications solutions become more sophisticated, directories must be able to support extended functionality, such as address conversions or replication and synchronization of directory information across networks.

Third, implementation problems also constrain the use of current directories. These problems include rigid database systems, complex query languages, and lack of integration with communication tasks.

The DirSA had to address all of these problems, as well as meet the primary requirements of:

- Providing consistent directory information across multiple communication products;
- Supporting consistent mechanisms for interoperation;
- Helping applications complete “all” types of communication tasks;
- Promoting a unified and uniform user administration; and
- Allowing easy integration of the directory with the rest of the customers’ computer-based systems.

In addition to these requirements, the architectural solution also had to meet extra-functional requirements for increased overall performance, reliability, and scalability; lower development costs; and simplified serviceability, manageability, and maintainability.

**The References: Directory Standards**

In the future, more of the power of software development is expected to come from the reuse of design knowledge codified in the form of patterns, guidelines, and architectural standards. A number of established international standards, emerging standards, and implemented services are available for directories, creating a solid basis for developing an open interoperable directory system that meets the objectives described above. Two standards have played an important role in the solution addressed in our case study: the International Organization for Standardization (ISO) Directory Standard/X.500 Recommendation and the lightweight directory access protocol (LDAP).

The X.500 Directory Standard defines a number of models, operations, and protocols. The models include an information model, a name space, a functional model, a security model, and a distribution model. Operations are defined in three main areas, namely, search and read, modify, and authenticate. The protocol that prescribes the use of these operations is called the directory access protocol (DAP). Interactions between distributed directory servers are prescribed by three additional protocols: directory system protocol (DSP), directory operational binding management protocol (DOP), and directory information shadowing protocol (DISP).

LDAP is an emerging standard, widely supported by the Internet community. Instead of requiring the “heavy” ISO Open System Interconnection (OSI) protocol stack, LDAP uses transmission control protocol/IP. Otherwise, it adopts the information model and name space from the X.500 Recommendation. The LDAP functional model is a subset of the one defined by ISO X.500.

Initially, the Directory project focused on an X.500-compliant solution. Because this project began before the wide acceptance of LDAP, the examples we use in this paper are X.500/DAP-based and reflect the first version of the solution. Later in our work we also addressed an LDAP-based solution.

**The Solution: An Architectural View**

This section describes the solution to our problem, which involved developing an architectural framework, identifying and examining architectural alternatives, selecting a solution, and applying GARM-ASPECT to describe it.


The first step toward determining the solution was to define the context of the directory system architecture and its influences on the current and future product development and integration efforts. To do this, we identified an initial architectural framework for the domain of ECS, whose three primary roles are to:

- Foster common understanding and, hence, treatment of architectural issues across ECS;
- Facilitate reuse and component-based development; and
Create a basis for interoperation and sharing of services.

This framework is an evolving collection of architectural standards, specifications, rules, and agreements under the guidance and terms of which the domain-specific collection of systems and components are developed, integrated into flexible "customer offers," and maintained.

The architectural specifications, as well as the components implemented in conformance to them, constitute the core of a company's reusable assets. When reused, architectural descriptions provide verified solutions of design elements, patterns, and practices, to shorten the product development cycle and keep the target product in line with the intellectual integrity of prior efforts.

The reuse of architectural designs also leads to the reuse of implementations. The architecture serves as a blueprint for component implementation and acquisition. By defining standard interfaces, constraints, and common interaction and information models, the architecture promotes the simplified, fast, and cost-effective construction of products from commercial and proprietary reusable components. This creates a basis for a "plug-and-play" approach to software construction, which should result in systems with higher quality and lower cost. It also promises more evolvable systems—a result of being able to easily replace constituent components.

The ECS framework identifies two major types of architectures:

- **Product line architectures**, which are parameterized architectural solutions for families of products, such as a messaging system architecture or a switching system architecture; and

- **Cross-product architectures**, which introduce integration platforms and define shared or reusable components that offer common services, such as security, transaction management, and directory.

As we discussed in earlier sections, the directory is a repository of consistent (directory) information that can be used by multiple communication servers, clients, and end users. In the context of the ECS architectural framework, it is a server component that provides common (directory) services to multiple software components, systems, and users.

### Architectural Alternatives

Based on the requirements discussed earlier and considering the issue of legacy systems, cost, and performance, we identified and examined three alternative architectural solutions:

- **Alternative 1**: Local legacy directory—global united directory.
- **Alternative 2**: No local directory—global standard directory, and
- **Alternative 3**: New local directory (a replica)—global standard directory.

**Alternative 1: Local legacy directory—global united directory.** The first architectural alternative provides a common directory, while it simultaneously protects customers' investments and promotes a smooth transition from an environment of loosely related (legacy) products to one of integrated solutions. This alternative was considered a near-term solution that can be implemented quickly to improve the current situation and then expanded and evolved over time.

The near-term solution architecture allows legacy communication servers to remain unchanged and their administration to be handled independently. Along with the directory server, this architecture introduces a new "synchronizer" component, which plays an important role in the integration strategy. It provides access to the directory and, as its name suggests, the synchronizer is responsible for keeping the overall directory-related information in a consistent state across multiple communication servers. It acts as a gateway between legacy products and the directory server, communicating with communication servers in their native protocols and with the directory by using the standard DAP, or in the new version, LDAP.

**Alternative 2: No local directory—global standard directory.** In this alternative, the directory server is the only entity that can manage directory information, as well as the only point of system administration. The communication servers and applications interact with the directory server in real time to finish their tasks. The solution we describe in this paper uses the X.500 DAP to relay clients’ requests to the directory server.
Alternative 3: New local directory (a replica)—global standard directory. The directory server in this third alternative is the only point of directory information management and administration. Communications servers contain local copies (partial customized replicas, or views) of the directory content. They interact with the directory server to refresh the local copy and to provide extended services such as communications with users from different domains.

We compared the three alternatives according to three groups of goals:

- **Customer requirement goals:**
  - Single point of administration (SPA),
  - Consistent directory information (CDI), and
  - Extended services (ES).

- **Design goals:**
  - Avoidance of single point of failure (ASPF),
  - Performance/real-time communication (PF),
  - Data integrity (DI), and
  - Algorithmic simplicity (AS).

- **Business goals:**
  - Low cost of legacy product management (LCLPM),
  - Low cost of additional development efforts (LCAD), and
  - Cost ratio percent (of change).

**Table 1** shows a brief overview of the comparison, the pros and cons of the three alternatives, and a summary of our conclusions. We see alternative 1 as a
near-term solution, favorable to legacy products and applicable to small-scale integration efforts. Alternative 3 is recommended as a long-term solution for large-scale integrated offers built from new products. Alternative 2 is similar to 3, because it assumes that the communication servers and applications will interact with the directory server by using standard DAP (or LDAP). This interaction, however, is required for every directory-related operation, which makes the architecture more suitable for low-volume, low-risk offers. The global directory server does not imply a monolithic component. The directory can be a cluster, built of multiple directory service agents (DSAs), distributed over multiple machines.

DirSA: The Selected Solution

Considering current customer needs and business goals, we have selected the near-term solution, alternative 1, as the basis for the initial architectural work. In addition, we have taken into account the evolution of legacy products, as well as new development efforts. As a result we have specified a general architecture for integrating multiple legacy, modified, and new communication servers and clients with an open real-time directory server. In this solution, the directory server is an autonomous component, compliant with the ISO X.500 directory standard. It provides common directory services in the overall architecture, characterized as a component-based, modular design.

**Figure 3.** Implementation view of the Directory System Architecture.
The directory system is decomposed into a collection of components, each of which is allocated a particular responsibility in the system. The components, drawn as boxes in the architectural diagram, are combined into a configuration via contracts, drawn as double-headed arrows.

The core of our system is the directory data server, which provides support to a number of clients—administrators, browser-based applications, new or modified communication systems (for example, call control engines, messaging servers, and clients), and auxiliary products such as the synchronizer.

The interactions among the components represent a separate view of the solution. As shown in Figure 3, the system contains different interaction types—such as “SQL,” “remote procedure call (RPC),” and “DAP”—which imply design and implementation constraints.

We viewed system scalability and extensibility as specific architectural challenges. Ideally, the architecture should remain stable despite a significant increase in scale. For example, an important early consideration is the point at which such a system would have to evolve from a single directory data server (DirSOFT) to a cluster of such servers, as shown in Figure 3. It was equally important to make an early determination of the cost of building in the cluster flexibility and of the long-term benefits of such an architectural solution.

The directory as a cluster component that can be implemented as a single or a distributed directory. Based on analysis and evaluation of commercial directory products, our initial decision was to focus on a single directory server implementation.

**Applying GARM-ASPECT**

We used both GARM and ASPECT to engineer and represent the architectural solution. GARM provided guidelines for the process of abstracting architectural properties and identifying architectural components and contracts. We used ASPECT to write the DirSA specification, part of which is included later in this paper.

The process of engineering the software architecture, as implied by the model, is one of refinement, in which more general architectural definitions are specialized and instantiated into specific system architectures. Figure 4 presents the most general view of DirSA architecture. The elements at the first level of refinement are specializations from the generic element types of the language; therefore, the parent type attribute in their headers is defined as “generic” in Figure 5.

The directory data server—DirDataServer—is specified as a cluster component. It can be implemented as a configuration of possibly distributed data servers, DirSOFT, compliant with the DSA specification of the ISO X.500 standard. The body of the directory server is
Figure 5.
Elements of an architectural description of DirSA (top part).
a composition—DirCluster. Its internal structure is an architecture—Scal DirArch (see Figure 5). This architecture is composed of encapsulated data servers—DirSOFT building blocks—interoperating according to the rules of three protocols: DOP, DISP, and DSP.15 The information tree and the replication-shadowing mechanisms are allocated to these encapsulated data servers according to the rules of the X.500 standard. Directory services are provided at the directory data server’s ports, comprising its interfaces: a DAP-based directory user interface (DUI) and an SQL interface (DSQLI).

As illustrated in Figure 3, the directory data server provides services to a set of directory clients. These clients interoperate with the directory data server using the DAP. A specialized class of clients also has an SQL-based interface to the directory data. The DirClient component is a description of a generic client. The AdminDirClient component is a specialization that stems from the DirClient component. Although it is not shown in our example, each specific directory client is derived from these component types. In theory a DirClient could be represented as a refinement of an abstract directory user agent (DUA).15 However, we have taken a more practical approach and have represented the implementation view, that is, a cluster of a domain-specific component (such as a personal DIR or a messaging server) and a DUA library component (DUAL). This decision was influenced most by the availability and popularity of off-the-shelf DUA libraries.

The contracts introduced in Figures 4 and 5 represent the three types of interaction defined in the architecture:

- The DAP contract, which represents the directory access protocol;
- The SQL contract, which defines an SQL-based interaction with the directory data; and
- The RPC contract, which defines an RPC-based interaction.

The “ownership of data” and the mechanisms for “change propagation” were significant issues in the data aspect of the system. We used the rules and scenarios8 of ASPECT to reflect not only these two issues, but also the global control structures in the architectural specification. For performance reasons, we have provided SQL-based (non-DAP) interfaces to the directory data, which comply to the general security rules defined in the architecture.

We used the ASPECT templates to represent the types of components, interactions, and common structural patterns of the target architecture. They defined element types that can be refined and extended to produce new definitions (see Figure 5). They can also be instantiated into specific elements and system architectures. As we discussed above, DirClient is a component type. From it, we derived multiple clients (see Figure 3) and extended them with additional interfaces, strictly observing the subtyping discipline of ASPECT. For instance, the administrative directory client, AdminDirClient, was derived from the DirClient element type and extended with an SQL-based interface.

**Summary and Outlook**

A “good” architecture is a product of both methodology and good architectural design. Methodology defines the units of architectural abstraction and the system of specification. A good methodology provides for important architectural properties like modularity and consistency, but it does not guarantee them. Good use of the methodology—that is, good design and process—is required as well.

We have used this case study and the example of the DirSA prototype architecture to show that the GARM-ASPECT approach can be used to guide the process of software architecture engineering and to provide rigorous architectural specifications of practical architectural solutions.

Following the steps of the process implied by the methodology4,18 we have defined the scope of the architecture and the levels of abstraction at which architectural issues are to be addressed. We have also identified several alternative solutions to the “directory problem.” We then evaluated the alternatives and specified the selected solution. (This paper does not describe all steps of the process in detail.)

Applying GARM-ASPECT, we have underlined a number of important methodological aspects of abstract description of software architectures. We have separated interaction from computation as component and contract types, described different interaction
types as different contracts, specified all component interfaces, and defined patterns of control and ownership of data.

The methodology enabled us to take a systematic approach to architecture definition and specification that helped us to gain a better insight into the problem. It also helped us to identify and evaluate several alternative architectural solutions and to produce precise architectural specifications, and, hence, better documentation.

The formalization of component, interface, and contract types has permitted us to define a more general solution and to analyze its properties. The constraints and alternatives we have identified and described have laid the groundwork for developing a more successful solution.

Acknowledgments

We gratefully acknowledge Don Stewart and David Sokoler of Lucent Technologies, who lent their confident leadership and support to the directory architecture engineering and specification effort. The technical contributions of Saswata Bhattacharya, Piyush Lumba, Rahim Choudhary, and Ninad Mehta, also of Lucent, were valuable in specifying the directory architecture. To the many colleagues who helped us with their insights and comments, we extend our thanks. Special appreciation goes to Thomas Marlowe, affiliated with the Department of Mathematics and Computer Science of Seton Hall University, in South Orange, New Jersey, for his creative suggestions. The authors also recognize the significant contributions made to this work by Wilhelm Rossak, currently affiliated with Institut für Informatik of Friedrich Schiller University in Jena, Germany.

References


(Manuscript approved September 1998)
VASSILKA D. KIROVA is a member of technical staff in the Cross-Product Architecture and Systems Engineering Department in the Business Communications Systems Group at Lucent’s Middletown, New Jersey, location. Her interests include software architectures, architecture representation, requirements specification, software process, and systems integration. Ms. Kirova received B.S. and M.S. degrees from the Technical University of Sofia in Bulgaria and Saint Petersburg Electrotechnical University in Russia, both in computer science and engineering. In addition, she is a visiting professor in the Department of Computer and Information Science at the New Jersey Institute of Technology in Newark, where she is also a Ph.D. candidate in computer science.

HOWARD G. KRADJEL works as a systems engineer in the Computer Telephony Integration (CTI) Platforms and Applications Department in the Business Communications Systems Group at Lucent’s Middletown, New Jersey, location. His interests include software development methodology, system architecture specification, code generation, and automated system construction. Mr. Kradjel holds a B.S. from the Pennsylvania State University in University Park and an M.S. from Lehigh University in Bethlehem, Pennsylvania, both in industrial engineering.◆