The Distributed Application Architecture

Joseph Sventek
Hewlett-Packard Company
19447 Pruneridge Avenue
Cupertino, CA 95014
jss@cup.hp.com

1 Introduction

The Distributed Application Architecture (DAA) is designed to allow users of a computer network to access information, applications, and services, as well as to exchange information with others, through a single, consistent user environment. It enables the construction of new applications and services; it also provides facilities for the integration and migration of existing applications.

A complete system based on the DAA includes both components which supply services provided as part of the infrastructure and a set of conventions or policies defining how components are to interact with the provided services and each other. These conventions, in particular, enable the integration of components in an enterprise-wide context.

2 Design Philosophy

The DAA was not designed in a vacuum. Very real product requirements were applied to academic models to yield an architecture which can meet a variety of customer needs: usability, extensibility, and backwards compatibility in a distributed environment. This section describes the design philosophy behind the DAA.

2.1 Multiple Viewpoints

Systems constructed using the DAA can be viewed in multiple ways. Three of the most important are:

- The user's view - for a particular application domain, the user wants a consistent look and feel to the components that are used. This usually implies that components must conform to some rules and guidelines to enable them to work together in the common environment.
- The programmer's view - a programmer wishing to use the services of a component must understand the computational model behind the system - i.e. how interactions are initiated, interaction guarantees, how errors manifest themselves, how concurrency is represented, etc. There also may be rules and other guidelines which may affect the programmer writing components for a particular application domain.
- The administrator's view - an administrator must configure the components of a system to satisfy the administrative policy of the containing enterprise. Details of component location (both of the component's persistent state and the code necessary to animate the component) fall into the administrator's domain. The interaction between components and traditional administrative tasks (e.g. backup and restore) is also important.

2.2 Object-oriented Model

An object-oriented model is uniquely suited for a distributed environment. The encapsulation of state embodied in the model corresponds very well to the lack of shared memory in a distributed system; the operation invocation paradigm is paralleled by the ability to send and receive messages.

One cannot just blindly adopt language-based, object-oriented models, as the design centers for these models typically do not include aspects of distribution. For example, some applications need an at-most-once, request-response structure for interactions between objects (traditional object-oriented invocations), while others need a best-effort, request-only interaction style. The data types which may be conveyed in requests and responses are usually a subset of those available for use in common languages; in particular, passing a naked pointer to another object is essentially meaningless. The richer class of errors in a distributed environment forces the object model to explicitly accommodate exceptions.

Of particular importance in an object model for distributed systems is to distinguish between type inheritance and implementation inheritance; many object-oriented languages lump these two concepts together. While implementation inheritance is a perfectly reasonable way to force type inheritance for objects *in a single address space*, it is of questionable utility in a distributed environment.

2.3 Design for Extensibility

We are not clairvoyant enough to know all possible uses of DAA-based systems. As such, the architecture must be designed for extensibility.

2.3.1 Application Domains

Different application domains have different use models. For the DAA to be usable in different application domains, it should be structured into a part common to all domains (e.g. object model, communication, and location-independence) and a domain-specific part (e.g. object lifecycle, properties that each object must have, and how persistent state is stored).

The domain-specific part (policies) can be further partitioned into policies that are basic (common to many domains) and those that are truly domain-specific. Maximizing the generality of the basic policies to many application domains greatly enhances the ability to integrate objects from these different domains.

2.3.2 Existing Object Systems

Products are available today that are object-oriented; object-oriented databases are good examples. Assuming that the object models behind these products are mappable to the DAA's object model, it should be possible to easily integrate these products into a DAA-based system without massive reprogramming effort. This forces the domain-independent portion of the system (see Section 2.3.1) be structured in such a way as to permit this easy integration.

2.3.3 Multiple Different DAA Implementations

When designing an implementation of the domain-independent portion (see Section 2.3.1) of a DAA system, the designer makes several choices for implementing abstract concepts of the architecture. Two independent implementations may be constrained to make different choices. This forces the architecture to truly tease apart concepts from implementation artifacts (e.g. communication requirements instead of "use

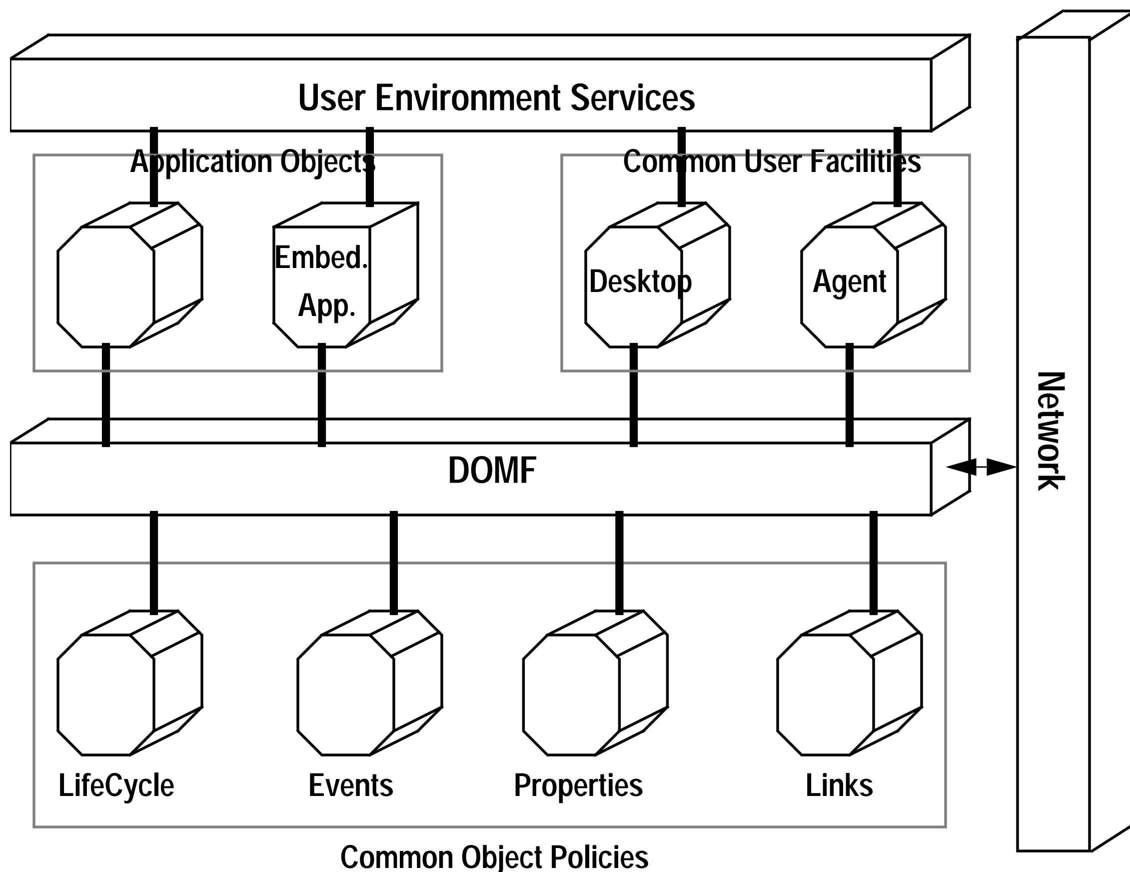

FIGURE 1. Distributed Application Architecture Model

of a particular communication protocol”).

Implementations which have made different choices may still interoperate through a variety of mechanisms. The fact that all interactions are based on a strongly-typed object model guarantees that gateways may be inserted with no loss of information.

3 Architecture

A complete system based on the DAA includes infrastructure components that supply services to objects and a set of conventions or policies defining how objects interact with the services provided and with each other. Figure 1 gives a pictorial view of the architecture.

The application architecture is organized around the concept of the DAA object, an abstract data object that uses the DAA infrastructure and subscribes to the appropriate DAA policies. Use of the term object in the remainder of the paper implies a DAA object.

3.1 DOMF

The Distributed Object Management Facility (DOMF) is the component of the DAA infrastructure that

manages the existence and execution of objects, as well as interactions between objects. The DOMF is the central layer of the architecture, providing basic functionality common to any DAA-based system. The OMG CORBA specification [2] is based on the DOMF.

3.2 Common Object Policies

These components are the object-management policies and supporting services. Some policies will have broad applicability, while others are more narrowly defined.

3.3 User Environment Services

These services provide an abstract view of the windowing environment. All objects that interact with a user do so through this element.

3.4 Common User Facilities

These facilities are user-accessible objects available on (most) DAA systems. These objects (such as desktops, folders, etc.) are customizable components of the architecture.

An Agent is the general term for services (objects) that can be constructed to monitor and control other objects and the interactions between them. The Agent automates repetitive tasks, automates (and facilitates compliance with) standard processes, and creates new solutions by integrating and combining existing objects.

3.5 Application Objects

These are objects that are used to accomplish specific end-user functional tasks. Application objects may have a specific focus (e.g. an accounting or CAD application), or they may be more generalized (e.g. word processing or spreadsheet application). Applications can be implemented to take advantage of any level of distributed support provided by the DAA. Existing applications can be embedded in the system.

4 DOMF

The DOMF supports the computational model described in Section 4.1. It is also architecturally divided into two layers:

- Communication Services - provides a uniform mechanism for delivering requests and responses
- Object Management Services - provides object adapters and an object location service

4.1 Computational Model

A DAA system provides services to clients. A *client* of a service is any entity capable of requesting a service.

4.1.1 Objects

A DAA system consists of entities known as objects. An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

4.1.2 Requests

Clients request services by issuing requests. A *request* is an event - i.e. something that occurs at a particular time. The information associated with a request consists of an operation, a target object, zero or more (actual) parameters, and an optional request context.

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests. Request forms are language-specific.

A *value* is anything that may be a legitimate (actual) parameter in a request. A value may identify an object or an abstract data type. A value that identifies an object is called an *object name*.

An *object reference* is an object name that reliably denotes a particular object. Specifically, an object reference will identify the same object each time the reference is used in a request. An object may be denoted by multiple, distinct object references.

A request may have parameters that are used to pass data to the target object; it may also have a request context which provides additional information about the request.

A request causes a service to be performed on behalf of the client. One outcome of performing a service is returning the defined results, if any, to the client.

If an abnormal condition occurs during the performance of a request, an exception is returned. The exception may carry additional, exception-specific, return parameters.

The request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A request may also return a single *result value*, in addition to any output parameters. A result value, if defined, is simply a distinguished output parameter.

4.1.3 Object Creation and Destruction

Objects can be created and destroyed. From a client's point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

4.1.4 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over values. A value *satisfies* a type if the predicate is true for that value. A value that satisfies a type is called a *member of the type*.

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

The *extension* of a type is the set of values that satisfy the type at any particular time.

An *object type* is a type whose members are objects (literally, values that identify objects). In other words, an object type is satisfied only by (values that identify) objects.

4.1.5 Interfaces

An *interface* is a description of a set of possible operations that a client may request of an object. An object *satisfies* an interface if it can be specified as the target object in each potential request described by the interface.

An *interface type* is a type that is satisfied by any object (literally, any value that identifies an object) that satisfies a particular interface.

Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The *principal interface* is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

4.1.6 Operations

An *operation* is an identifiable entity that denotes a service that can be requested.

An operation is identified by an *operation identifier.* An operation is not a value.

An operation has a signature that describes the legitimate values of request parameters and returned results. In particular, a *signature* consists of:

- a specification of the parameters required in requests for that operation
- a specification of the result of the operation
- a specification of the exceptions that may be raised by a request for the operation and the types of the parameters that accompany them
- a specification of additional contextual information that may affect the request
- an indication of the execution semantics the client should expect from a request for the operation

Two styles of execution semantics are defined by the object model:

- at-most-once: if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at most once; and
- best-effort: a best-effort operation is a request-only operation - i.e. it cannot return any results and the requester never synchronizes with the completion, if any, of the request.

4.2 Communication Services

The Communication Services layer enables requests to be delivered among objects. It is responsible for the conversion of data representation required when the requestor and performer are on different kinds of computers. It must also provide the basic security mechanisms upon which the user-perceived security models are constructed.

The Communication Services isolate the rest of the DOMF and the objects from the effects of different network topologies, protocols, and software.

The Computational Model (Section 4.1) makes particular demands upon the Communication Services. The protocols used must support the at-most-once and best-effort delivery of requests. They must also be able to signal an exception if an error condition occurs during the delivery of a request or a response.

Typical remote procedure call (RPC) protocols provide appropriate bases [1] for the Communication Services. Besides handling the delivery of requests and responses, such protocols provide for the marshalling of parameters to support interactions between objects constructed on differing machine architectures. They also support an exception model upon which to construct the DAA exception model.

4.3 Object Management Services

4.3.1 Object Location Service

In order for the Communications Services to deliver a request to a specified object, the computer on which the target object is located must be identified. When a DAA object issues a request, the target object is specified by an object reference. Given an object reference to any object, the Object Location Service can find that object anywhere on the network, even a network spanning a large, multinational corporation.

In the current implementation of the Object Location Service, a network is divided up into regions. While there is no hard rule governing its size, a region might typically be a building or a small campus. Every computer must be a member of one and only one region. Each region has an Object Region Expert (ORE), which is used to locate objects within the region. There is also a Meta Object Region Expert (MORE), which is used to locate regions and computers within each region. While there is conceptually only one MORE per network, and one ORE per region, implementations are likely to replicate them to improve performance and availability.

This Object Location Service has been optimized for locality of reference. This means that locating an object should depend upon as little of the rest of the network functioning as possible. For example, it should be possible for one object to locate another in the same computer without the network being functional, and to locate an object in the same region, requiring only that the region and its ORE be functional.

4.3.2 Object Adapters

An object adapter is the primary means for an object implementation to access DOMF services. An object adapter exports a public interface to the object implementation. It is built on a private interface to the communication services.

Object adapters are responsible for the following functions:

- generation and interpretation of object references
- method invocation
- security of interactions
- object and implementation activation and deactivation
- mapping object references to the corresponding object implementations
- registration of implementations

These functions are performed using the Communication Services and any additional components necessary. Often, an object adapter will maintain its own state to accomplish its tasks. It may be possible for a particular object adapter to delegate one or more of its responsibilities to the Communication Services upon which it is constructed.

The architecture defines the Basic Object Adapter (BOA), which can be used for most objects with con-

ventional implementations, generally separate programs. It allows there to be a program started per method, a separate program per object, or a shared program for many instances of many types of objects. It provides a small amount of persistent storage for each object, which can be used as a name or identifier for other storage. If the implementation is not active when a request arrives for the object, the BOA will start it. The BOA guarantees that an object can determine the principal of the object making a request so that it can apply discretionary access control

5 Policies

The DOMF provides a sufficient set of services to permit objects to interact via requests. As such, it forms a distributed backplane through which objects can interact.

While such a backplane is sufficient, it hardly is likely to lead to integrated solutions. It is essential that standard policies be defined which constrain the behavior of consenting objects to permit integration of objects into solutions unforeseen by their original implementors. It is these policies which permit the DAA to enable enterprise-wide integration.

5.1 General Description

All of the policies defined by the DAA are with respect to the behavior of pairs of consenting objects. Each policy consists of one or more interfaces and/or a description of the protocols to which objects interacting via the policy must subscribe.

The functionality inherent in some policies is specific enough to permit the specification of interfaces that each party must support. In addition to supporting the interfaces, there is a description of the sequences of requests through these interfaces that are legal in the course of interactions via the policy; the responsibilities of each party to the interaction, in addition to these request sequences, must also be specified.

Other policies are less specific. In fact, some policies can only give general guidelines for the objects' behavior.

The policies are defined based upon the use model for particular application domains. Substantial enterprise-wide integration is enabled if the policies are carefully segregated into a set of basic policies (those applicable to a large number of application domains) and sets of application-domain-specific policies.

5.2 Relationship to Services

By carefully restricting the policy specifications to the behavior of consenting objects, the architecture gives the widest possible latitude to the respective object implementors. Of particular importance is the ability for other objects supporting the specified interfaces to be interposed between the interacting parties.

This form of indirection permits value-added services to be provided in support of a particular policy. The use of such services is transparent to the other party of the interaction, in keeping with the encapsulation basis of the object-oriented model.

The enabling of services in such an environment is important; it essentially establishes a third-party marketplace for objects which can satisfy certain performance or other requirements.

5.3 Particular Policies

The policies defined by DAA support the implementation of a distributed application environment over a variety of computer architectures and host operating systems. The use model for DAA is based upon the familiar desktop metaphor; it provides support for compound objects and other features of a desktop system.

5.3.1 Basic Policies

5.3.1.1 Events

An *event* is an occurrence within an object that may be of interest to other objects; typically, an event occurs when an object experiences a change of state that is externally visible. When an event occurs, the object generating the event notifies other objects which have expressed interest in that event and which may take some action based upon that event occurrence.

The basic architecture supports two forms of interaction between consenting objects:

- blocking request/response operation invocations by one object (the requester) of another object (the performer)
- non-blocking, one-way operation invocations by the requester of the performer

Both of these interaction styles are very synchronous in nature - i.e. the requester invokes an operation in the performer when the service is desired.

Another common style of interaction between consenting objects consists of registration of interest by the requester in occurrences in the performer; when such an occurrence takes place, the expectation is that the performer will notify the interested party of the occurrence. This is a more asynchronous style of interaction.

An object designer is free to support this asynchronous style in an interface-dependent way; since typed object references can be passed as arguments to operation invocations, a particular interface can define one or more operations which take an object reference as an argument. When an appropriate occurrence takes place in the performing object, it can invoke an operation in the interface passed to it by the requester; this assumes that the event generator has *a priori* knowledge of the operation to invoke in the event observer's interface.

It is expected that the need for this asynchronous style will be pervasive enough that significant leverage can be achieved by defining a basic policy for events; higher level policies constructed upon the basic policy may also be appropriate to meet additional needs of programmers.

5.3.1.2 Links

Any object may hold an object reference to another object. Links are value-added object references that may be held persistently. Links enable various levels of referential integrity to be maintained in the system.

Three styles of links are supported:

- one-way: this is basically one object holds an object reference to another; the target of the link does not know of the link, and neither party has any responsibilities with regard to the link (i.e. the holder can drop it at any time)
- two-way, with notify: both the holder and the target know about this link; the holder is required to notify the target if it drops the link, and the target is required to notify the holder if it is committing suicide
- two-way, with negotiation: both the holder and the target know about this link; the holder must negotiate permission from the target to drop the link, and the target must negotiate permission from the holder to commit suicide

5.3.1.3 Properties

The object model supports the notion of attributes; attributes represent portions of the object's state which are made accessible to clients via accessor operations.

More generally, there are properties associated with an object. Properties are (name, value) pairs representing aspects of an object's state. Some properties are mandatory, while others are optional. Optional properties may be created or destroyed.

Each object must support a standard interface permitting the creation and deletion of optional properties. This interface also permits all properties to be read and/or written. Attributes in the object model, given that they are part of an object's type, constitute the mandatory properties. The value associated with a mandatory property may be read/written through the standard property interface or using the accessor operation pair defined by the object model.

5.3.1.4 Negotiation

A key aspect of distributed object models is that objects can be introduced at run-time, often by end users, with the intent that they work together for some purpose. From the objects' point of view, they must enter into a relationship that involves some interaction protocol. Each object plays a particular role in carrying out the protocol, and thus in supporting the relationship.

Since objects are generally designed independently and only connected at run-time, there must be a standard way to verify that each object can carry out its role in the protocol. Negotiation is the process by which an object can determine at run-time whether another given object conforms to a specified type - i.e. whether it can play a particular role in a proposed relationship.

5.3.1.5 Trading

An important issue in distributed systems is the ease with which applications can be constructed from, and interwork with, existing components. At any time, it may be important for a component to know what services are available in a system. The types of systems needed may be chosen at design time, but the choice of a particular instance will invariably occur at run-time.

Negotiation permits the determination of the appropriateness of a particular object for a particular role. Trading facilitates the determination of appropriate objects available in the system for a particular role. Obviously, the most important parameter in this determination is the type of each object, since each object's type is directly related to the roles it can play. Additionally, there may be associated information which can be used (properties) about the objects to make a particular choice from among a group of

objects that satisfy the type constraints.

5.3.2 Desktop-specific Policies

The desktop-specific policies are those that are specifically designed to facilitate the desktop metaphor. Many of these policies are also expected to be applicable in other application domains, but have not been proven so.

5.3.2.1 Containment

To permit a user to manage objects, each desktop object must satisfy the containment policy. This policy forces each object to know about its parent (there can be only one) and all of its children. This containment relationship forms a hierarchy. Obviously, there is some primeval object that has no parent; all others are descendants of it.

Support for the containment policy affects many other desktop policies. For example, if an object is to be moved, it must reestablish its parent relationship with the new destination. When coupled with the notion of colocation, the containment metaphor establishes a notion of object location, especially of their persistent state, for the user.

5.3.2.2 Object Lifecycle

There are certain lifecycle operations that an object must support to participate in the desktop. These include:

- creation - this is actually a requirement on the factory objects for a particular type of object, as an object cannot be asked to create itself;
- delete - when asked to delete itself, an object must terminate its relationship with the system;
- copy - the object must copy its state into another environment where a clone of it will be available; and
- move - the object must move its state into another environment; typically, this is done via an atomic copy, rename, delete operation.

5.3.2.3 Compound Object Lifecycle

Compound objects result from a coupling of the containment relationship between objects and forced colocation. Application of lifecycle operations to a compound object require the compound operation to be propogated along the containment/colocation relationships.

Due to the multiple objects involved in such compound object lifecycle operations, the system must support the ability to coordinate the actions of the objects involved. These coordinated actions are similar to transactions, with the exception that the user may be consulted in the middle of the coordinated action to determine the appropriate action to take if the entire compound operation cannot be successfully performed (e.g. an object is told to delete itself; it is the target of a two-way with negotiation link from another object, and the other object indicates that it does not agree to the object's demise.)

5.3.2.4 Data Interchange

While data may be passed between consenting objects as arguments and results to requests, there are often occasions when two objects would like to enter into a more stream-like relationship to ship larger amounts of information.

There are many standards for data interchange formats defined in the industry [3], [4]. The data interchange policy must permit the two objects to negotiate about the format of the data to be interchanged and what data is to be interchanged (e.g. a range of spreadsheet cells, paragraphs of type Foo in a document). The data supplier provides an event which the consumer can register interest in; notifications of that event are delivered whenever the data has changed, enabling the consumer to retrieve the associated information.

5.3.2.5 Presentation-Semantic Split

All user-visible objects consist logically of two parts:

- a semantic portion, which understands what particular services the object provides to other objects and
- a presentation portion, which provides the user interface, permitting the user to interact with the object.

By actually implementing the user-visible object as two DAA objects along this logical partition, we have been able to give the user extremely effective use of distributed objects. Such a structure permits many users to have simultaneous views of the same object; it permits each of these views to be of a different sort (e.g. one user sees a table of numbers, another sees a chart of the same information). It also permits the presentation object to be in a different network location (usually closer to the user) from the semantic object, thereby improving the immediacy of response by the object to user gestures.

There is obviously a private interface between the semantic object and its presentations to permit the presentation objects to retrieve the information which must be displayed to the user. This protocol typically uses events to permit the semantic object to notify its presentation objects of changes to the data behind their presentations.

6 Conclusions

From HP's experience with various prototypes, it is clear that the DAA DOMF/Policy split significantly enables application integration. The further decomposition of the policies into basic and application-domain-specific enhances enterprise-wide integration. HP is working closely with the Object Management Group and standards organizations, both *de jure* and *de facto*, to further the adoption of these concepts.

References

[1] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato and G. Wyant, *Network Computing Architecture*, Prentice Hall, Englewood Cliffs, NJ, 07632, ISBN 0-13-611674-4, 1990.

[2] *The Common Object Request Broker: Architecture and Specification*, Document Number 91.12.1, Object Management Group, 492 Old Connecticut Path, Framingham, MA, 01701, 1991.

[3] *Office Document Architecture and Interchange Format*, ISO/IS 8613/1-6, ISO/TC97/SC18.

[4] Postscript Language Reference Manual, Adobe Systems Incorporated, Addison-Wesley, Menlo Park, CA, December 1985.