CASE Integration Using ACA Services

1 Abstract

Digital uses the object-oriented software Application Control Architecture (ACA) Services to address the problems associated with data access, interapplication communication, and work flow in a distributed, multivendor CASE environment. The modeling of applications, data, and operations in ACA Services provides the foundation on which to build a CASE environment. ACA Services enables the seamless integration of CASE applications ranging from compilers to analysis and design tools. ACA Services is Digital's implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification.

2 Introduction

Based on work accomplished in many computer-aided software engineering (CASE) projects, this paper describes how Digital's object-oriented Application Control Architecture (ACA) Services can be used to construct a CASE environment. The paper begins with an overview of the types of CASE environments currently available. It describes the object-oriented technique of modeling applications, data, and operations and then proceeds to discuss design and implementation problems that might be encountered during the integration process. The paper concludes with a discussion of environment management.

3 CASE Environment Description

Today's CASE environments are required to operate in network environments that consist of geographically distributed hardware manufactured by multiple vendors. In such environments, access to data, metadata, and the functions that operate on this data must be as seamless as possible. This can be accomplished only when well-architected protocols exist for the exchange of information and control. These protocols need not be defined at the level of network packets, but rather as operations that have well-defined, platform-independent interfaces to predictable behaviors.

In addition to utilizing the various applications, environments deal with how applications are organized or grouped within a project and how work flows between applications and within the environment as a whole. These concepts are discussed later in the paper as are the different styles of integration that an application can employ.

Data integration, i.e., information sharing, is vital to any CASE environment because it reduces the amount of information users must enter. However, data integration must be accompanied by a mechanism that allows

control to pass from one application to another. This mechanism, commonly called control integration, provides a means by which the appropriate

application can be started and requested to perform an operation on a piece of information. Control integration is also used to exchange information between cooperating applications, regardless of their geographic locations. These two integration mechanisms used in tandem can solve many of the problems presented by a distributed, multivendor CASE environment.

ACA Services is Digital's implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification. ACA Services is designed to solve problems associated with application interaction and remote data access in distributed, multivendor environments such as the CASE environments just described. This support includes the remote invocation of applications and components without the need for multiple logins or the use of terminal emulators. The encapsulation features of ACA Services allow the use of applications not designed for distributed environments. ACA Services can also be configured, in a way transparent to the application, for use on a local host.

The central focus of a CASE environment is on how easily functions such as compiling, building, and diagramming can be performed. The functions available form the foundation on which the environment is constructed. Therefore, the first step in the design of a CASE environment is to determine what functions to offer. The applications currently available to support these functions may be integrated using one of two paradigms: application-oriented or data-oriented.

Application-oriented Paradigm

CASE environments that follow the application-oriented paradigm focus on standalone applications used to develop software such as editors, compilers, and version managers. Application-oriented environments normally comprise a collection of applications that support the necessary functions. In application-oriented environments, integration tends to be focused on direct communication between two different applications. In this paradigm, the requesting application knows which class of application can be used to satisfy a particular request. Environments that present an application-oriented paradigm to the user require the user to have knowledge of the applications that can be used to perform specific tasks.

As the level of task complexity increases, it becomes increasingly important to build environments that utilize a paradigm focused on the data associated with the task being done and not on the applications used to perform the task. The realization of this problem has brought about the existence of data-centered environments.

Data-oriented Paradigm

CASE environments that use a data-oriented are centered around the data

associated with the task the user is performing. To accomplish a task in such environments, operations are performed on a data object. Using the object being addressed, the operation, and preferences supplied by the user, the environment determines which application will be used to perform the requested operation. Thus, the requesting application requires no

knowledge about which application implements an operation. This paradigm is extremely useful in CASE environments because of the diversity of objects and range of applications available to perform certain operations.

The application and the data paradigms can coexist in a single CASE environment, and in fact, tightly integrated CASE environments exploit the strengths of each paradigm. A text editor can be used to illustrate this point. Typically, when the contents of a source file need to be modified, an edit operation is sent to the object representing the file. However, a debugger may also use the same editor to display source code. The operation to position the cursor on a particular line is sent directly to the text editor application, rather than to a data object such as the line. An environment with such a split focus avoids the expense and complexity of presenting a complete object-oriented interface to the environment and results in the existence of both application- and data-oriented paradigms.

Regardless of which paradigms and applications a CASE environment uses, the primary focus of the environment is on the objects and on the operations that are defined on those objects. Therefore, after determining what functions to offer, the second step in designing a CASE environment is to understand how applications, data, and operations are modeled using an object-oriented approach, in particular the one provided by ACA Services.

4 CASE Integration in Object-oriented Terms

Describing environments using object-oriented techniques can simplify the design of an environment. Techniques such as abstraction and polymorphism can be used to describe the objects that comprise the environment, the operations that can be performed on those objects, and any relationships that exist between objects. Furthermore, using these techniques makes it possible to describe an environment as a set of classes and services for each class. ACA Services performs the role of the method dispatcher, matching an object and an operation with the function in an application that can implement that operation. To realize the benefits of this approach requires constructing models for the applications, data, and operations that will be present in the environment.

Modeling Applications and Application Relationships

Applications that are integrated into an environment can provide various functions or services to other members of the environment. The number of services an application provides depends not only on the capabilities of the application but also on the way it is modeled. These services are standalone pieces that can be plugged into a system to perform specific functions. An application can define a single operation whose sole function is to start the application; an application can export the entry points of its callable interface; or an application can define sets of operations for

each type of object it manipulates. In support of application modeling, ACA Services provides the concepts of application classes, methods, and method servers. Figure 1 illustrates the relationships among the various pieces of information used to model an application in ACA Services.[1]

CASE Integration Using ACA Services

In ACA Services, the definition of an application is divided into two pieces: interface and implementation. The interface definition is concerned with the publicly visible aspects of the application. These include class definitions for the objects that the application manipulates, a class definition for the application itself, and definitions of operations that the application supports. The operations, which represent the functions provided by the application, are modeled as messages on the application class definition. These messages define a consistent interface to various implementations of the operations. Placement of the application class definition affects the behaviors this definition inherits. This is sometimes called classification. The classification of each component of an application depends on whether a component contains a superset or a subset of the functions contained in the components of other applications in the environment.

Once the application's components have been classified, the integrator must determine how the application will make its capabilities available to the environment: as an operating system script, as a callable interface, or as an executable image. The implementation definition represents the actual implementation of the application. An application may be comprised of a number of executable files and shared libraries. Typically, only the executable file used to start the application is modeled as a method server. If the functions of the application are provided through a shared library or image, only the shared library is modeled as a method server.

The implementation of the functions or services exported to the environment are modeled as methods. Methods describe the callable interfaces or operating system scripts that implement a particular operation and are associated with only one method server.[2] During the method selection process, the messages defined for the application and the objects it manipulates are mapped onto one or more methods.

Modeling Data and Data Relationships

Data modeling is another significant aspect of creating CASE environments, especially environments that utilize a data-oriented paradigm. Identifying the data objects that the application uses is a key element in the process of integrating that application. The list of data objects should include those objects for which the application provides a service, as well as those objects on which the application makes requests. The variety and quantity of data objects can vary from application to application and depends on an application's capabilities and the paradigm utilized. To support the modeling of data objects, ACA Services uses the concept of data classes. Note that, rather than provide instance management for data objects, ACA Services provides a means to represent the data classes used by an application as metadata.

Because environments that utilize a data-oriented paradigm may contain many data classes, ACA Services organizes the data classes into an inheritance hierarchy. This hierarchy allows responsibilities, such as operations and attributes, to be inherited by other data classes. Data classes found in an

ACA Services inheritance hierarchy are related to one another through an "is-kind-of" relationship. A class that has an "is-kind-of" relationship with one or more superclasses must support all operations defined on the superclasses from which it inherits.[3] A subclass is not limited to those operations and attributes defined by a superclass but may have other operations, as well as refinements to inherited operations and attributes.

Modeling Operations

As mentioned previously, operations are modeled as messages in the CASE environment. The name of the message describes the type of operation. Some messages are data oriented, i.e., Edit, Reserve, and Copy, whereas other messages are application oriented, i.e., ExecuteCommand and TerminateServer. Messages provide a consistent abstraction of the functions provided by applications. This abstraction allows the details of how a function is implemented to be hidden from the requesting application. Since ACA Services supports more than one implementation for a single message, it also provides a means to hide various implementations.

The developer should anticipate different implementations of a message within the environment and be aware that a message may apply to a variety of classes. The developer must consider how the operation on an object might be used by various applications and in future environments.[4] In this way, adding new types of objects to an environment requires only minor changes, if any, to applications that are already integrated.

Operation Interactions. The semantics of a message dictates which particular interaction model is to be used. ACA Services can be used to construct a number of different interaction models: synchronous request, asynchronous request, and request/reply, as shown in Figure 2. The synchronous request interaction model, shown in Figure 2(a), is useful when serial operations originate from a single source. This model blocks the execution of the client application during a request. Control is returned to the client application only after the server application receives and executes the request and outputs data, if any.

The asynchronous request interaction model, shown in Figure 2(b), is useful in situations where the client can process other work until the server application completes the request. This model is especially beneficial when the requested operation takes a considerable amount of time to complete or if the server is busy with other requests. Execution of the client application is blocked only for the amount of time required to deliver the request. Client execution resumes once the request has been delivered. Upon completing the processing of the request, the server application notifies the client application of the completion and returns any output data.

The request/reply interaction model, shown in Figure 2(c), is most

appropriate for requests whose implementations cannot perform the operations required to obtain the necessary output data. Gateway and message-forwarding applications are examples of applications for which this type of interaction model is best suited. In this model, the message that represents the request cannot have any output arguments and must

pass an application handle to itself. The server application uses the application handle to return any output information to the requester by sending a message that represents the reply. In a request/reply model, a single reply message should be defined for returning information, thus reducing the number of messages an application must support.

Message Arguments. A message argument for passing the object being manipulated need not be defined. ACA Services automatically passes the object to which the message was sent to the method. Each method routine can access the object through a structure containing context information for the current invocation.

The arguments of a message should not be designed around a specific instance of an application, nor should they imply how an object is physically stored. To help meet these design criteria, all references to an object should be passed as instance handles. In this way, the application that receives the instance reference can use it directly for subsequent operations on that object. In addition, when defining the message arguments, developers should consider other applications that could be instances of a particular class and possibly used as replacements.

However, all instances of an application do not have the same set of capabilities. To support the various capabilities, the developer may have to define additional arguments to represent bit masks and flags. An argument list or an item list can be used to pass information about different data types or quantities. The message design should not require implementation-specific information for proper application operation; this design implies that reasonable defaults accommodate any unspecified information. In cases where proper operation of an application requires implementation-specific information, the most suitable design is to use the context object as a place to store the default values. With such a design, the application no longer needs to use hard-coded default values and can be customized for the environment.

5 Integration Frameworks

A number of issues must be resolved in the construction of a CASE environment before the first line of code can be written. Many of these issues center around the modeling of objects in the environment. As discussed in the previous section, abstraction is used to hide much of the actual implementation of the operations on objects from the requesting application. However, additional context may be required for further operations. If the application is using an application-oriented paradigm, most operations are directed to an application class that provides the service. In cases where a data-oriented paradigm is used, the application typically directs operations to the data class of which the object is an instance.

Besides the application and data objects found in the environment, the designer must also take into consideration the other components of the CASE environment itself. Figure 3 shows the major components of a CASE environment: activities, applications, application and data interfaces, work flow management, and handle management. Each component represents a particular aspect of the overall environment. The components are introduced in this section and described in detail elsewhere in the paper, as indicated.

Activities provide the basic work structure for a particular task within an environment. Each activity comprises one or more applications and a number of data objects, forming a single composite object. Applications within an activity operate through the application interfaces. The section Application Integration describes the principles of an activity and includes a discussion of the sharing of applications within and among other activities.

Application interfaces, illustrated in Figure 3 as arrows connecting the various applications, form the primitives by which integration is accomplished. Some of the more general concepts for application interfaces were discussed in the section Modeling Operations; these concepts are described in detail in the section Styles of Application Interfacing.

Finally, the section Environment Management addresses how to manage the flow of work within the environment. This section describes the management of instance and application handles, the use of storage classes as a means to provide data transformations, and the management of events within the environment. To better understand each of these topics requires the following basic information about various aspects of the environment.

Adding New Implementations

Updates to the environment may include adding new application classes, data classes that the new application supports, method definitions for the application, and possibly a method server definition. As described earlier in the paper, ACA Services uses data and application classes to represent the different classifications of data and application objects found in an environment. Storage classes represent the classifications of storage and how objects are referenced in the environment. Each class, i.e., data, application, and storage, contains a list of messages that represent the operations that can be performed on the class.

Digital's CASE environment, COHESION, was designed to present a data-oriented perspective to the user. An initial level of integration was achieved by utilizing this same data-oriented approach to application integration. Implementation of a data-oriented approach required that method maps for messages on data classes contain an indirect reference

to an abstract application class.[5] Figure 4 illustrates this concept by showing two different messages: the Edit message, which uses an indirect method reference, and the Browse message, which uses a direct method reference. An indirect method reference has two parts separated by the character '@': first, the name of the message to be sent; and second,

the name of the class on which to send the message. Although not commonly done, an indirect method reference allows the original message to be mapped to another message on a different class, given that both messages have arguments of the same type, direction, and order. Both messages must also return the same type of object.

On encountering an indirect method reference, ACA Services first looks at tables in the context object for an attribute that matches the reference. If such an attribute is found, ACA Services uses the attribute value to determine the class and message that should be checked next. Thus, users can provide a mapping to their preferred application for the operation. If no matching attribute is found, ACA Services uses the message and class specified in the indirect method reference as the next place to check.

The approach used in COHESION has many advantages over specifying either a direct reference to a method or an indirect reference to a specific application class. This approach does not limit the user's ability to specify application preferences associated with using direct references to methods, nor does it burden the installation of the application with determining all the data classes that will need to be updated (as required with indirect references to a specific application class). In addition, the approach allows the application developer to do the least amount of work and still provide the maximum level of support for user preferences in applications.

Using ACA Services, the application developer must create an application class definition for each CASE application to be added. Consequently, the class hierarchy contains both abstract and instance classes. The application class is required to contain all the messages defined on its superclass, plus any additional messages that the application supports. The method map of each message on an application class should contain a direct reference to the method that implements the operation. Although better than the other alternatives, the COHESION approach has no default implementation unless one is explicitly specified in a context object. To overcome this problem, an entry for each message defined on the abstract application class must be created in one of the context objects. The values for these entries point to the corresponding message on the class of application used as the default implementation.

Common Classes

Common classes for a CASE environment provide CASE application developers with a description about how an application fits into the environment, the behaviors the application must support, and the messages that result in those behaviors. The notion of plug-and-play in the environment is achieved through the use of common classes. An implementation that adheres to the description of a particular class of applications can be easily switched

with another implementation that adheres to the same application class semantics.

Programs like COHESION are working toward a set of common classes for CASE environments. The set currently defined contains classes for many types of data and applications found in CASE environments focused on the coding and testing phases of the software development process. A graphical view of the data portion of the hierarchy is shown in Figure 5. The hierarchy is partially based on the hierarchy found in ATIS, a standard for tool integration, and utilizes the strength of the ATIS data model.[6] (Shaded boxes indicate the classes that are specific to ATIS.) Encompassing the ATIS model, the hierarchy presents a uniform data model for the integration of data throughout the CASE environment. The set of classes, although not exhaustive, serves as a basis on which a CASE environment can be built. Extensions of the hierarchy will occur as new classes of applications and their associated data objects are integrated into the environment by independent software vendors, customers, and other CASE vendors.

Most data classes are subclasses of the data class SOURCE_FILE, because the initial data class implementation was targeted at a CASE environment consisting of editors, compilers, builders, and analyzers. Additional data classes for both file and nonfile objects will be added when applications that provide and manipulate these objects are integrated into the environment. A number of data classes represent composite objects such as tests and activities. These data classes are used to hide how the object is physically stored in the environment. Classes that represent composite objects have attributes with values that are actually other objects. For example, the test data class typically has attributes that represent the result of a test run, an operating system script or program used to perform the test, and a benchmark against which a test run is compared. Each of these attributes may have as a value a reference to the file object that contains the actual data.

The portion of the hierarchy that is used to specify application classes contains only abstract application classes, as shown in Figure 6. These classes provide structure, but more important, they define the operations that are inherited by any application that is an instance of a class. Abstract classes are provided for a number of the applications found in CASE environments that deal with the coding and testing functions. The hierarchy does not contain any classes that represent particular instances of an application. Such application classes exist only when applications are installed in the environment.

Consistent Integration Interface

Many CASE vendors are building products for a number of different environments, including electronic publishing, office automation, computer-aided design, and computer-aided manufacturing, in addition to CASE. Therefore, vendors must decide how to integrate these applications into the various environments. Until now, most integration was accomplished by

linking one application with another, which resulted in tightly coupled applications. However, such applications tend to be unable to operate independently, without the other member. Also, each coupled member tends to have its own application programming interface (API). Integration performed

in this manner results in an application that must maintain code to support multiple APIs, if the application is to work in a number of environments. Such support can increase the maintenance cost and the time and effort required to integrate with other implementations of applications and environments. Other by-products of this approach are an increased image size and a need to rerelease software when a dependent application changes. The degree to which rerelease occurs varies with the platform and operating system.

ACA Services can be used to minimize the number of interfaces that an application must maintain without removing functionality; a common API provides the interface to all potential functionality. The ACA Services API, along with a set of common classes, allows the same level of interaction between applications that can be accomplished through a private API, without the negative side effects previously described. Through the use of common classes, an application can integrate with multiple implementations of another application without requiring a separate effort for each. On platforms where dynamic loading of libraries or shareable images are supported, applications can use ACA Services to locate the appropriate library, find the proper entry point, and transfer control to the appropriate routine. ACA Services also provides a transparent mechanism for encapsulating applications that have no callable interfaces. Use of this mechanism extends the number of applications that can be integrated and removes the need to develop operating system-specific code to start applications.

6 Styles of Application Interfacing

Creating an interface to an application that is to be integrated is different from integrating an application into an environment. Application interfacing deals with the public interface or interfaces that the application provides to another application. In turn, these interfaces provide the primitives that can be used in the integration of applications.

Application interfaces can be created in various ways, with differing levels of effort. Software developers can design new applications to utilize all the capabilities of ACA Services. Existing applications can also take advantage of the full capability of ACA Services, if the source code to the application is available and if the application can be easily adapted to use an event-driven model. However, even if the source code to an application is not available, applications can still be integrated into the environment using ACA Services. If the application has a callable interface, a server can be written that receives messages and calls the appropriate API routines. If the application does not have a callable interface, the application can be integrated by encapsulation through the use of an operating system script. The remainder of this section describes how to use each of these techniques to create an interface through which

the application can be integrated into a CASE environment.

Application Modifications

An existing application can easily be adapted to use ACA Services, if the source code to the application is available. With minimal changes, an application that utilizes an event-driven design, like that used by most window-based applications, can operate as an application server. The actual modifications required to provide ACA Services support differ across applications, but for most window-based applications the changes are similar. As an illustration of this style of integration, consider an editor.

Most editors are implemented as event-driven applications, which allows easy integration because the structure of the code requires no major changes. To register the current executing instance of the application with ACA Services, a call to the ACAS_RegisterServer routine must be added to the application's initialization routine. During the process of run-time registration, ACA Services registers various information about the application, including the identifier of the process in which the application is executing, the owner of the process, and the class- and instance-unique identifiers for the application. As part of the registration, an application can specify an abstract name by which it can be located and the routines to be called when an ACA Services event arrives, e.g., when the server is instructed to shut down or when a session ends.

Once registered with ACA Services, the application must enter its event dispatching loop. Because many applications have existing event dispatching mechanisms, ACA Services has been designed for easy integration with most mechanisms. ACA Services provides this support by allowing the application to define a routine called the event notifier, which is called at signal level each time an ACA Services event occurs. The event notifier routine places an event on the applications work queue for the ACA Services event. Upon encountering the event, the application's event dispatcher routine calls the ACAS_Dispatch routine to allow ACA Services to dispatch the appropriate method or management routine for the event. A description of how ACA Services dispatches operation requests follows.

Application Servers

When the application to be integrated does not have a user interface but provides a callable interface, integration is best accomplished by creating an application server. Considered a form of encapsulation, an application server provides a consistent programming interface to the application. An application server provides jacket routines that use the application's callable interface, hiding the actual details of this interface. This technique is also used to create applications that have a clean separation of presentation and functions.

Applications that implement persistent data stores, such as databases, code managers, and repositories, are prime candidates for this style of integration. By using an application server to access persistent data stores, a requesting application need not know how the data store is

implemented and which implementation is to be used. This technique promotes the reuse of existing functions contained in the environment regardless of the actual implementation of the function. Digital's Code Management System (DEC/CMS) and CDD/Repository software are examples of applications that have been integrated using the application server technique. Figure 7 illustrates the typical structure of the various components involved in this style of integration.

As shown in Figure 7, the integration process involves the following steps. (1) An invoke from the client application of the message "Reserve" on the object "foo.c" goes through the resolution code and (2) out the transport to the server application. This may result in starting the server application, if no server was available to service the request. (3) The server application's main routine calls the event dispatcher and waits for work to arrive, when the server is started. (4) When the "Reserve" message arrives on the transport, the transport notifies the server application, (5) causing the event dispatcher to dispatch the "Reserve" message by calling the method dispatcher routine. (6) The method dispatcher routine calls the appropriate method interface routine. (7) The method interface routine does any work required to call the appropriate callable interface routine. (8) When the callable interface routine returns control to the method interface routine, the routine can perform any work necessary before (9) returning control to the method dispatcher routine. (10) The method dispatcher routine then puts any arguments to be returned in the proper format and sends this information to the transport, which actually sends the information back to the client application.

Using the DEC/CMS application server as an example, the software developer must create a main routine to (1) perform any setup required to use the callable interface and (2) register the existence of the server with ACA Services. Registration includes specifying the method dispatcher routine, which is generated by ACA Services, so that the appropriate method routine will be dispatched for the message received.

A method routine exists for each operation that the server is capable of performing. The set of method routines is analogous to the operating system script for compilation used to explain application encapsulation later in this section. Because the DEC/CMS application server is not an operating system script, message arguments are passed into the method routine directly. As mentioned earlier in the section CASE Integration in Object-oriented Terms, the object on which the current operation is to be performed is available to the method routine through the use of the invocation context structure. Information about the object, such as its class, name, and generation, can be obtained by calling the ACAS_ ParseInstanceHandle routine. The class of the object can then be used to determine if the object is an element under version control, a collection, or a group.

The name of the object and its generation are contained in the reference data field of the instance handle that represents the object. Because each different code management system has its own representation of generation, it was necessary to create a canonical format to represent all implementations. Therefore, the method must convert the canonical generation representation to a format that is native to the implementation, i.e., DEC/CMS specific. In addition, any method that returns a reference to a versioned object must convert the native generation representation to its canonical format. Table 1 shows how an object reference can be mapped between its canonical and DEC/CMS-specific formats.

Once the necessary information about the object has been retrieved and converted to a format native to the implementation, the method can call to the appropriate callable interface routine, possibly based on the object's data class. Once the call completes, the method must convert any objects to be returned into a canonical format, at which point the method can return the status of the operation and output arguments.

Application Encapsulation

Encapsulation, the simplest integration technique, is appropriate for applications that do not have a callable interface or in cases where no source code is available. Compilers are an ideal candidate for this style of integration, because they perform synchronous operations. Encapsulation of compilers provides a consistent programming interface to any compiler that is integrated into the environment, regardless of the qualifiers used to specify particular compilation options. This technique can also be used to provide a generic compile command that is platform independent. Encapsulation of a compiler is best accomplished through the use of an operating system script. Figure 8 illustrates an example of an encapsulated compiler.

The purpose of an operating system script for compilation is to convert the generic compilation qualifiers, which are passed as message arguments, into the compiler-specific options. The /DEBUG and /NOOPT qualifiers shown in Figure 8 are examples of generic compilation qualifiers. Many operating system scripting languages limit the number of parameters that can be passed on the command line. The compilation scripts avoid these limitations by passing the name of the file to be compiled as the only command line parameter, as shown in the command @SYS$LIBRARY:COMPILE.COM %INSTANCE() in Figure 8. ACA convenience commands, such as APPL/CONT GET ARGUMENT, are used to retrieve and set the values of the message arguments in the operating system script. When all the switch values are gathered, the operating system script converts the generic values into specific qualifiers. Finally, the actual command line is constructed and executed. This same technique can also be used to encapsulate linkers and any other types of applications where no source code or callable interface

is available. When applications provide a callable interface, even tighter integration can be achieved by creating an application server.

7 Application Integration

Integration of applications goes beyond the interfaces that applications present to the environment; it concerns how applications interact with one another. Integration also takes into account the policies used in an environment to allow a collection of applications to be grouped into a single composite object. This section discusses concepts such as an activity, locating an application within an activity, context sharing, and the sharing of applications across multiple activities.

Activity Participation

Since more than one activity may be active at any given time, an activity must be able to locate the other applications participating in the activity. Data-oriented environments provide a means to loosely couple the various data and application objects into a single composite object. The COHESION integrated environment refers to this composite object as an activity. The implementation of an activity differs depending upon the environment: ATIS uses a persistent process; file system-based environments generally use a directory hierarchy; and environments built on a private data store can use a data file. In the COHESION environment, an activity is represented as an ACA Services context object that contains attributes that reference a directory hierarchy. The context object is used to set up the execution environment in which a set of applications will operate and to locate other applications that are executing within the activity.

Locating Activity Applications

The ability to locate an application that is executing in an activity allows for reuse of the application by other applications executing in that same activity. Such locating provides for better utilization of applications and reduces the amount of context that must be propagated from one application to another. To locate an application within an activity, an application must have registered its presence in the activity. When registering with ACA Services, the application must specify the activity name as the value of the attribute ACAS_SERVER_REGISTRY. The application must also register itself with the event manager to allow centralized management of the activity and to participate in the flow of work within the activity.

CASE applications determine if they are executing within an activity by checking for the existence of the environment variable ACTIVITY_NAME. If this environment variable exists, its value is the activity identifier. To allow an activity to extend beyond a single host and to support different activities with the same name, the activity is identified by a unique identifier.

Sharing within Activities

Applications executing within an activity operate in a common context. ACA Services provides a set of mechanisms that can be used to provide this common context. The environment variable ACTIVITY_NAME is defined each time a method server is started in the COHESION environment. The method server definition specifies as the value of the start-up environment attribute, the names of the context tables and attributes that are to be defined as environment variables upon start-up.

Another way of providing a common context across an activity is to propagate context object tables and attributes as implicit arguments to method servers. Specifying this information as implicit arguments instructs ACA Services to propagate these attributes to the context object of the method server servicing the request.

The context object can also be used directly to create a common context across an activity, i.e., by holding information that needs to be shared. This information can include references to directories, preferences of applications, and default values.

Sharing between Activities

Reusing applications that are active within an activity reduces the overall system resources required to perform the activity. However, a problem occurs when two or more activities are active at the same time and require the same application. With the addition of windowed interfaces and the need to utilize other services, application sizes have greatly increased. Consequently, it is often impractical to expect a separate instance of an application to be associated with each activity that is active.

In order for an application to be shared between multiple activities, the application needs a means by which to determine if a request is part of an ongoing dialog with another application or is the beginning of a new dialog. These dialogs, called "sessions," represent a conversation between a pair of applications. Each time a client application makes a request to a new application server, a session is established and an identifier is associated with the session. ACA Services passes the session identifier to the server application.

The management of sessions can be accomplished by using the session ID as a lookup key into a list of structures that represent the active sessions. When the server application locates the structure associated with the session identifier, the application can establish the appropriate context for that session. In the example of DEC/CMS application server, the structure would contain the handle to the library associated with the session.

ACA Services also notifies an application server when a session is to be terminated between a client and a server application. When notified, the application server determines the appropriate course of action. Using the CMS example, the server releases any cached information it has kept about

the session, closes the specific CMS library, and then frees the library data block.

8 Environment Management

After defining application interfaces and integrating applications into an activity, CASE environment developers must focus on the management of the environment as a whole. This includes the management of references to applications and data, the transformation of object references into platform-specific formats, and the flow of work within the environment.

Handle Management

In the CASE environment, objects are the targets of all operations. Sending a message to an object requires understanding how to create and manage references to the object. Since ACA Services does not manage instances of objects, it uses references to instances of objects. These references take the form of instance and application handles, which reference data and application objects, respectively. Proper management of these handles leads to more efficient use of application objects, thus reducing the amount of network resources and memory consumed by the application. Appropriate handle management can also enhance performance and guarantee predictable behavior.

Instance Handles

The creation of an object reference is performed by calling the ACAS_CreateInstanceHandle routine. ACA Services (1) creates an instance handle from the information passed as arguments to the routine, (2) allocates memory to the handle and manages this memory, and (3) sends a message to a storage class, if one was specified.

To avoid creating numerous copies of an instance handle, each with its own memory, a cache of objects should be used. This is especially true in CASE environments that use the data-oriented paradigm. Each object structure contains pointers to both the previous and the next object structure in the queue. The structure also contains values for the location and reference data fields that were passed as arguments to the ACAS_CreateInstanceHandle routine and, thus, allows for the unique identification of an object in the cache across multiple hosts. In addition to the location and reference data, the structure contains a pointer to the instance handle returned from the call to the ACAS_CreateInstanceHandle routine. Reuse of the instance handle saves the time required to create the handle, including any overhead associated with using storage classes. Reuse also reduces the total amount of memory required. However, instance handles are not the only handles that require management; application handles need to be managed as well.

Application Handles

Application handles are references to application objects. Each application handle can represent one or more method servers. A method server can generate a handle by calling the ACAS_CreateApplicationHandle routine, or the ACAS_InvokeMethod routine can return an application handle as an output argument. As with instance handles, application handles can be passed as arguments to a message. Management of application handles is similar to the management of instance handles. Each entry in the cache of application handles contains the location of the application and the name of the class of application. The entry also contains a pointer to the application handle and a count of the number of outstanding references to the handle. Freeing an application handle results in the termination of all sessions between the client and any method servers referenced by the handle; it also releases all memory associated with the handle.

Each instance handle should be associated with a corresponding application handle. This association allows the application handle to be reused when sending additional requests to the application concerning the data object. An application handle associated with a cache entry can be used to make the request. Failure to find the application in the cache could indicate that the appropriate invocation flag should be used to obtain an application when calling the ACAS_InvokeMethod routine.

As described, proper handle management can result in better performance, better resource utilization, and predictable behavior within the environment. However, handle management does not deal with how to create an object reference that, when presented to an application on a remote host, is in a format native to that platform. For this capability, we must turn to storage classes.

Data Transformations Using Storage Classes

Distributed CASE environments, whether homogeneous or heterogeneous, must concern themselves with the representation of object references that are shared among different applications. File specifications exemplify this problem. Given multiple hosts, it is unlikely that two hosts have the same path to a specified file, even if both hosts are of the same platform type. Consider the scenario in which Application A sends the Edit message to the file object $PROJ4:[PROJECT.SRC]SORT.C, resulting in a request of Application B to edit the contents of the file. The problem becomes complicated if Application B is executing on a different platform type than Application A.

To solve the problem, the environment can utilize the functionality provided by ACA Services storage classes. Storage classes provide a mechanism for translating an object's reference data from one file system

representation to another. A solution to the scenario described involves implementing a set of methods that would be executed when the object reference uses a storage class.

The SC_COHESION storage class is a CASE-specific storage class, which is a refinement of the SC_FILE storage class provided by ACA Services. As a refinement, SC_COHESION inherits all the messages defined on its parent storage class, including the messages SetInstance and GetInstance. The methods for these two messages provide an implementation for mapping file system specifications from platform-specific formats to platform-independent formats and back again. The storage class methods do this by utilizing device and directory information, called directory mappings, found in the context object.

The directory mappings stored in the context object provide a means to associate a physically shared directory path with a network path name. The network path name is a platform-independent name that, when presented to a remote platform, can be mapped into a format native to the platform receiving the request. A network path name and its mapping are stored as an attribute-value pair in the PATHNAME_REGISTRY table of a context object.

The directory mapping functionality allows references to file objects to be passed between applications on different hosts in a way independent of the platform. This same scheme can also be used to convert object references in object identifiers, such as ATIS element IDs for use with the CDD/Repository software. In the implementation for the file system, the method associated with the SetInstance message must determine the data class of the object reference, as well as transform the reference data into its network format. The determination can be made in a number of ways, the most common of which is to base the class on the extension of the file. Although not the most accurate method of determining the class, this approach does meet the needs of many files.

Work Flow Management

ACA Services manages the various instances of executing applications but does not understand the concept of an activity. Therefore, managing the applications within the activity requires the use of an application that understands this concept. The event manager, which acts as a central registry of active applications and their associated activities, can provide a simple form of work flow management within the environment. However, the event manager is used only in a limited capacity in the COHESION integrated environment. In COHESION, the event manager is notified each time an application is started or stopped in an activity. The application provides an application handle to itself, which is used by the event manager to notify the application of events of interest. The use of the event manager removes the need for an application to forward certain messages, as a result of an event in the environment, to all applications with which it has been communicating. Removing the need to forward messages reduces both the chances of loops forming in a set of applications and any communication deadlocks between applications.

Events and Triggers

On registration, an application can express interest in being notified about particular events. Events are categorized into two classes: system events and application events. System events affect the overall operation of the environment. These events include shutdown and changes in activities. All applications in the COHESION environment are notified of the system events for activity shutdown, iconification, and deiconification. Application events occur when the state of an object in the environment changes. File modification or completion of a build step are typical examples of application events. Other applications in an activity can use these events for synchronization or as notifications that cause a change in behavior. Such notifications have traditionally been called triggers.

For example, in a simple build system such as the make utility, events can create a work flow that would automatically compile and link an application when one module changes. If the build process completes successfully, the work flow automatically starts the debugger to debug the newly built executable file. If the build fails, the work flow loads the faulty module into a program editor and positions the cursor to the line where the error occurred.

9 Summary

ACA Services can be used to resolve many problems encountered in a distributed, multivendor environment. The object-oriented approach provided by ACA Services can aid in the construction of a CASE environment that promotes the plug-and-play concept across a number of different platforms and network transports. ACA Services provides a means of developing client-server applications and of abstracting the network dependencies away from the developer. This feature, together with the use of storage classes and data marshaling, can help to exchange information in a heterogeneous environment. At the same time, ACA Services can provide a consistent programming interface to all components in the system. The dynamic nature of ACA Services allows new components to be added to the environment without the need to rebuild the entire environment. The flexibility of ACA Services allows its use to construct a CASE environment regardless of the integration paradigm used and while supporting a number of interaction models. ACA Services provides the infrastructure necessary to integrate the large number of existing applications into distributed, heterogeneous environments.

10 Acknowledgments

The author wishes to thank Jackie Kasputy, Chip Nylander, and Gayn Winters for their invaluable insights and contributions on distributed, multivendor

CASE environments.

11 References

1. E. Yourdon, Modern Structured Analysis (Englewood Cliffs, NJ: Yourdon Press, 1989).

2. DEC ACA Services System Integrator and Programmer's Guide (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQKMA-TE, 1992).

3. G. Booch, Object Oriented Design with Applications (Redwood City, CA: Benjamin/Cummings Publishing Company, 1991).

4. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990).

5. DEC ACA Services Reference Manual (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQKLA-TE, 1992).

6. J. Liu, "Future Direction for Evolution of IRDS Services Interface," X3H4/92-161, Proposed specification submitted to ANSI X3H4 and ISO IRDS, 1992.

12 Trademarks

The following are trademarks of Digital Equipment Corporation: CDD/Repository, COHESION, and Digital.

13 Biography

Paul B. Patrick, Sr. Paul Patrick is a principal software engineer in the ACA Services Group. He leads Digital's implementation of the Object Management Group's Common Object Request Broker Architecture. Previously, Paul helped design COHESION, an integrated CASE environment based on the DECset architecture. He also contributed to the development of IPSE, an integrated project support environment based on the CDD/Repository software, and designed and implemented the MicroVAX 2000 synchronous controller diagnostic. Prior to joining Digital, Paul held positions at GenRad Inc. and Norand Corp.