Overview of the CORBA Component Model

Wang, Schmidt, O'Ryan

Section	VI	Other Author(s)	Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan
Chapter	38	E-mail Address	nanbor@cs.wustl.edu, schmidt@uci.edu, coryan@cs.wustl.edu
Pages		Phone Number	(314) 935-6355
Date Complete		Dates Reviewed	GH (5/24/00 12:35 PM), GH (5/25/00 11:10 AM), GH (6/23/00 4:20 PM), GH (8/23/00 2:06 PM), GH (8/30/00 11:24 AM), GH (9/5/00 4:13 PM)

Introduction

In today's globally competitive software market, it is becoming increasingly important to develop, deploy, and maintain complex, distributed software systems. Many companies have developed proprietary software to enable the distribution of applications over a network, but this solution can be prohibitively expensive and time-consuming over the lifecycle of complex software systems. Since 1989, the Object Management Group (OMG) has been standardizing an open middleware specification to support distributed applications. The traditional OMG Common Object Request Broker Architecture (CORBA) (OMG, 2000) shown in `Figure 1` enables software applications to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols, interconnections, or hardware (Henning and Vinoski, 1999).

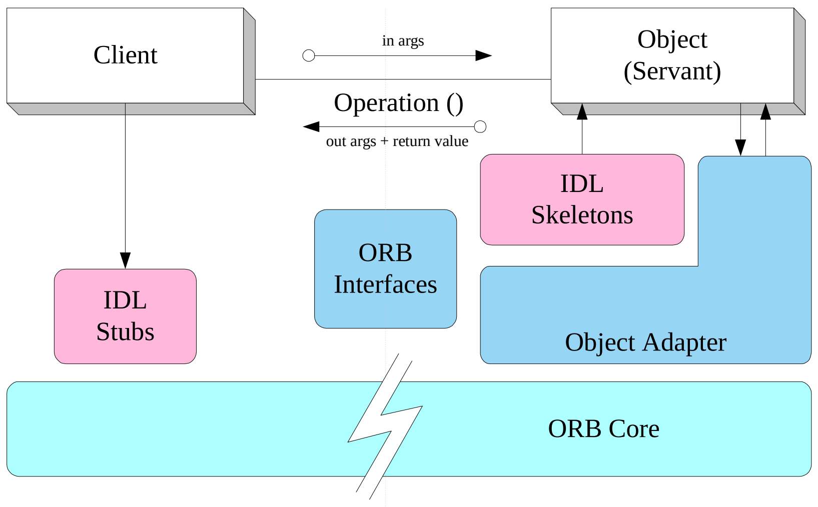


Figure 1: Traditional CORBA Object Model

To provide higher-level reusable components, the OMG also specifies a set of CORBA Object Services that define standard interfaces to access common distribution services, such as naming, trading, and event notification. By using CORBA and its Object Services, system developers can integrate and assemble large, complex distributed applications and systems using features and services from different providers.

Unfortunately, the traditional CORBA object model, as defined by CORBA 2.4 (OMG, 2000), has the following limitations:

1. *No standard way to deploy object implementations.* The earlier CORBA specification did not define a standard for deployment of object implementations in server processes. Deployment involves distributing object implementations, installing those implementations in their execution context, and activating the implementation in an Object Request Broker (ORB). Thus, system designers developed *ad hoc* strategies to instantiate all objects in a system. Moreover, since objects may depend on one another, the deployment and instantiation of objects in a large-scale distributed system is complicated and non-portable.
2. *Limited standard support for common CORBA server programming patterns*. The CORBA family of specifications provides a rich set of features to implement servers. For example, the CORBA 2.2 specification introduced the *Portable Object Adapter* (POA), which is the ORB mechanism that forwards client requests to concrete object implementations. The POA specification provides

standard application programming interfaces (APIs) to register object implementations with the ORB, to deactivate those objects, and to activate object implementations on-demand. The POA is flexible and provides numerous policies to configure its behavior. In many application domains, however, only a limited subset of these features is ever used repeatedly; yet server developers face a steep learning curve to understand how to configure POA policies *selectively* to obtain their desired behavior.

3. *Limited extension of object functionality*. In the traditional CORBA object model, objects can be extended only via inheritance. To support new interfaces, therefore, application developers must: (1) use CORBA's Interface Definition Language (IDL) to define a new interface that inherits from all the required interfaces; (2) implement the new interface; and (3) deploy the new implementation across all their servers. Multiple inheritance in CORBA IDL is fragile, because overloading is not supported in CORBA; therefore, multiple inheritance has limited applicability. Moreover, applications may need to expose the same IDL interface multiple times to allow developers to either provide multiple implementations or multiple instances of the service through a single access point. Unfortunately, multiple inheritance cannot expose the same interface more than once, nor can it alone determine which interface should be exported to clients. (Henning and Vinoski, 1999).
4. *Availability of CORBA Object Services is not defined in advance*. The CORBA specification does not mandate which Object Services are available at run-time. Thus, object developers used *ad hoc* strategies to configure and activate these services when deploying a system.
5. *No standard object life cycle management*. Although the CORBA Object Service defines a Life cycle Service, its use is not mandated. Therefore, clients often manage the life cycle of an object explicitly in *ad hoc* ways. Moreover, the developers of CORBA objects controlled through the life cycle service must define auxiliary interfaces to control the object life cycle. Defining these interfaces is tedious and should be automated when possible, but earlier CORBA specifications lacked the capabilities required to implement such automation.

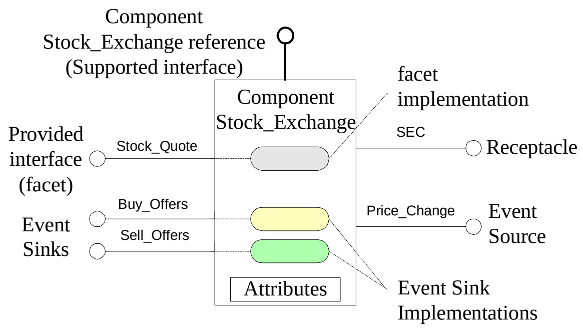
In summary, the inadequacies outlined above of the CORBA specification, prior to and including version 2.4, often yield tightly coupled, *ad-hoc* implementations of objects that are hard to design, reuse, deploy, maintain, and extend.

Overview of the CORBA Component Model (CCM)

To address the limitations with the earlier CORBA object model, the OMG adopted the CORBA Component Model (CCM) (OMG, 1999b) to extend and subsume the CORBA Object Model. The CCM is planned for inclusion in the CORBA 3.0 specification, which should be released by the OMG during 2001. The CCM extends the CORBA object model by defining features and services that enable application developers to implement, manage, configure, and deploy components that integrate commonly used CORBA services, such as transaction, security, persistent state, and event notification services, in a standard environment. In addition, the CCM standard allows greater software reuse for servers and provides greater flexibility for dynamic configuration of CORBA applications. With the increasing acceptance of CORBA in a wide range of application domains (Schmidt, 1995; Levine, 1998; O'Ryan, 1999; OMG, 2000), CCM is well positioned for use in scalable, mission-critical client/server applications.

Component Overview

CCM components are the basic building blocks in a CCM system. A major contribution of CCM derives from standardizing the component development cycle using CORBA as its middleware infrastructure. Component developers using CCM define the IDL interfaces that component implementations will support. Next, they implement components using tools supplied by CCM providers. The resulting component implementations can then be packaged into an assembly file, such as a shared library, a JAR file, or a DLL, and linked dynamically. Finally, a deployment mechanism supplied by a CCM provider is used to deploy the component in a *component server* that hosts component implementations by loading their assembly files. Thus, components execute in component servers and are available to process client requests. Figure 2 shows an example CCM component implementing a stock exchange and its corresponding IDL definition.



```
interface Sell, Buy;

// Define an equivalent, supported interfaces
component Stock_Exchange supports Sell, Buy {
  provides Stock_Quote;    // Facet

  consumes Buy_Offers;     // Event Sinks
  consumes Sell_Offers;

  publishes Price_Change;  // Event Source
  uses SEC;                // Receptacle
    ...                    // Other definitions
};
```

Figure 2: An example CCM Component With IDL Specification

A CORBA object reference is an abstract handle referring to an instance of a CORBA object. An object reference hides the location where the actual object resides and contains protocol information defined by the CORBA specification, as well as an opaque, vender-specific *object key* used to identify a servant that implements the object. To developer end-users, the format of a reference to a `Stock_Exchange` component is identical to the format of a reference to a `Stock_Exchange` interface. Thus, existing component-unaware software can invoke operations via an object reference to a component's *equivalent interface*, which is the interface that identifies the component instance uniquely. As with a regular CORBA object, a component's equivalent interface can inherit from other interfaces, called the component's *supported interfaces*. In our example, the supported interfaces perform transactions to buy and sell stock. As mentioned earlier, it is inflexible to extend CORBA objects solely using inheritance. Thus, CCM components provide four types of mechanisms called *ports* to interact with other CORBA programming artifacts, such as clients or collaborating components. These port mechanisms specify different views and required interfaces that a component exposes to clients (Marvie, 2000). Along with component attributes, these port mechanisms define the following capabilities of a component:

1. **Facets**: Facets, also known as *provided interfaces*, are interfaces that a component provides, yet which are not necessarily related to its supported interfaces via inheritance. Facets allow component to expose different views to its clients by providing different interfaces that can be invoked synchronously via CORBA's two-way operations or asynchronously via CORBA's asynchronous method invocations (AMI) that are part of the forthcoming CORBA 3 specification. For instance, the `Stock_Quote` interface in Figure 2 provides a stock price

querying capability to the component. CCM facets apply the Extension Interface pattern (Schmidt, 2000) and are similar to component *interfaces* in Microsoft's Component Object Model (COM) (Box, 1997).

2. **Receptacles**: Before a component can delegate operations to other components, it must obtain the object reference to an instance of the other components it uses. In CCM, these references are “object connections” and the port names of these connections are called receptacles. Receptacles provide a standard way to specify interfaces required for the component to function correctly. In Figure 2, the `Stock_Exchange` component uses the `SEC` (Securities and Exchange Commission) interface to function correctly. Using these receptacles, components may *connect* to other objects, including those of other components, and invoke operations upon those objects synchronously or asynchronously (via AMI).
3. **Event sources/sinks**: Components can also interact by monitoring asynchronous events. These loosely coupled interactions, based on the Observer pattern (Gamma, 1994), are commonly used in distributed applications (Pyarali, 2000). A component declares its interest to publish or subscribe to events by specifying *event sources* and *event sinks* in its definition. For example, the `Stock_Exchange` component can be an event sink that processes `Buy_Offers` and `Sell_Offers` events and it can be an event source that publishes `Price_Change` events.
4. **Attributes**: To enable component configuration, CCM extends the notion of *attributes* defined in the traditional CORBA object model. Attributes can be used by configuration tools to preset configuration values of a component. Unlike previous versions of CORBA, CCM allows operations that access and modify attribute values to raise exceptions. The component developer can use this feature to raise an exception if an attempt is made to change a configuration attribute after the system configuration has completed. As with previous versions of the CORBA specification, component developers must decide whether an attribute implementation is part of the transient or persistent state of the component.

These new port mechanisms significantly enhance component reusability when compared to the traditional CORBA object model. For instance, an existing component can be replaced by a new component that extends the original component definition by adding new interfaces *without affecting existing clients* of the component. Moreover, new clients can check whether a component provides a certain interface by using the CCM `Navigation` interface, which enumerates all facets provided by a component. In addition, since CCM allows the *binding* of several unrelated interfaces with a component implementation entity, clients need not have explicit knowledge of a component's implementation details to access the alternative interfaces that it offers.

To standardize the component life cycle management interface, CCM introduces the `home` IDL keyword that specifies the life cycle management strategy of each component. Each `home` interface is specific to the component it is defined for and manages exactly one type of component. A client can access the `home` interface to control the life cycle of each component instance it uses. For example, the `home` interface can create and remove components instances.

To use a component, a client first acquires the `home` interface of the component.Naturally, there must be a standard bootstrapping mechanism to locate the `home` interface of a specific component. To simplify this bootstrapping process, references to available component `home`s can be stored in a centralized database accessed through a `HomeFinder` interface similar to the CORBA Interoperable Naming Service (OMG, 1998). A client first uses the standard CORBA API `resolve_initial_references` to acquire the object reference to a `HomeFinder` interface. `HomeFinder` enables clients to acquire a reference to the desired `home` interface of the component. After a client acquires a reference to the `home` interface of the desired component, the client can invoke the appropriate factory operation (Gamma, 1994) to create or find the target component reference.

Development and Run-time Support Mechanisms for CCM

CCM addresses a significant weakness in CORBA specifications prior to version 3.0 by defining common techniques to implement CORBA servers. We now describe how CCM behaves from a *component developer's* perspective, allowing the developer to generate many types of server applications automatically.

CCM extends the CORBA IDL to support *components*. Component developers use IDL definitions to specify the operations a component supports, just as object interfaces are defined in the traditional CORBA object model. A CCM component can compose together unrelated interfaces and support interface navigation, as described in the *Component Overview* section.

Components can be deployed in component servers that have no advance knowledge of how to configure and instantiate these deployed components. Therefore, components need generic interfaces to assist component servers that install and manage them. CCM components can interact with external entities, such as services provided by an ORB, other components, or clients via *ports*, which can be enumerated using the introspection mechanism. Ports enable standard configuration mechanisms to modify component configurations. `Figure 3` shows how the CCM port mechanism can be used to compose the components of our stock exchange example.

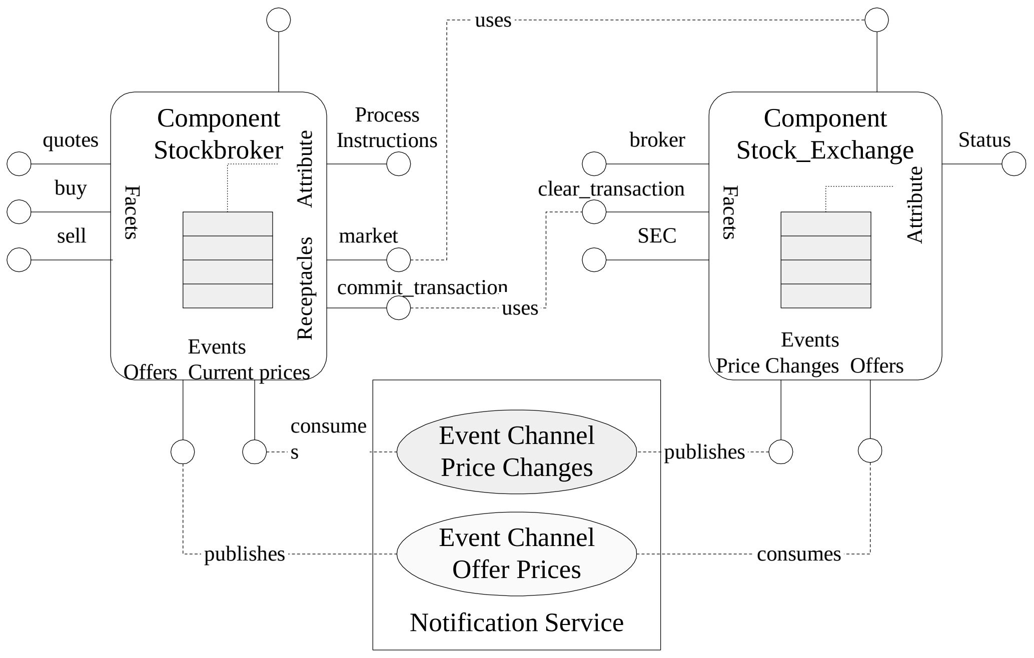


Figure 3: CCM Components Interact with One Another Through Port Mechanisms

The CCM port mechanisms provide interfaces to configure a component, enabling developers to set up object connections, subscribe or publish events, and establish component attributes. For a developer to assemble components into a software component infrastructure or integrate a component into an application, however, there must be a mechanism to express a concrete configuration for a component; in particular they need to designate what component(s) must be connected and how the events published and received by a component relate to each other. Therefore, CCM defines a standard component configuration interface, called `Components::StandardConfigurator`, to help component servers configure components. Component developers can extend this configuration interface to specify how to improve the flexibility of their component implementations.

A component's `home` interface can optionally accept a component configuration object reference that performs the component configuration on a component instance. All CCM components support introspection interfaces, which these configurators use to discover the capabilities of components. The configurator then constructs the component instance by making the necessary interconnections with other components or ORB services.

CCM defines several interfaces to support the structure and functionality of components. Many of these interfaces can be generated automatically via tools supplied by CCM Providers. Moreover, life cycle management and the state management implementations can be factored out and reused. The CORBA *Component Implementation Framework* (CIF) is designed to shield component developers from these tedious tasks by automating common component implementation activities.

Many business applications use components to model "real world" entities, such as employees, bank accounts, and stockbrokers (refer to Carey and Carlson, Chapter XX,``). These entities may persist over time and are often represented as database entries. Components with persistent state are mapped to a persistent data store that can be used to reconstitute component state whenever the component instance is activated. For example, when a bank account component is instantiated, the CCM component model implementation is able to reconstitute the previous status of the account from a database. The CIF defines a set of APIs that manage the persistent state of components and construct the implementation of a software component .

CCM defines a declarative language, the *Component Implementation Definition Language* (CIDL), to describe implementations and persistent state of components and component homes. As shown in `Figure` 4, the CIF uses the CIDL descriptions to generate programming skeletons that automate core component behaviors, such as navigation, identity inquiries, activation, and state management.

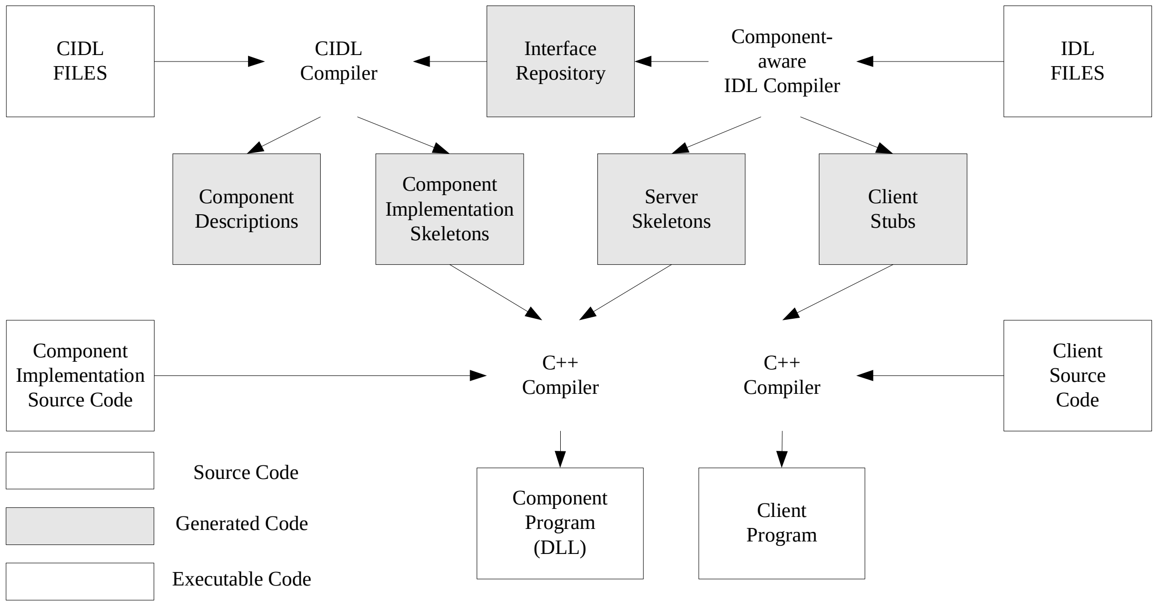


Figure 4: Implementing Components using the Component Implementation Definition Language (CIDL)

Implementations generated by a CIDL compiler are called *executors*. Executors contain the aforementioned auto-generated implementations and provide hook methods (Gamma, 1994) that allow component developers to add custom component-specific behavior. Executors can be packaged in so-called *assembly files* and installed in a *component server* that supports a particular target platform, such as Windows NT or Linux, and programming language, such as C++ or Java. In addition, the CIDL is responsible for generating *component descriptors* that define component capabilities, such as descriptions of component's interfaces, threading policy, or transaction policy, along with the type of services required by the component being described.

Component implementations depend upon the standard CORBA Portable Object Adapter (POA) to dispatch incoming client requests to their corresponding servants. However, unlike previous versions of CORBA, the application developer is no longer responsible for creating the POA hierarchy, the CCM component model implementation uses the component description to create and configure the POA hierarchy automatically and to locate the common services defined by CCM. Moreover, components may require notification via *callbacks* when certain events occur. To support the functionality outlined above in a reusable manner, component servers instantiate *containers*, which perform these tasks on behalf of components they manage. The CCM *container programming model* defines a set of interface APIs that simplify the task of developing and/or configuring CORBA applications. A container encapsulates a component implementation and provides a run-time environment for the component it manages that can:

1. Activate or deactivate component implementations to preserve limited system resources, such as main memory.
2. Forward client requests to the four commonly used CORBA Object Services (COS): Transaction, Security, Persistent State, and Notification services, thereby freeing clients from having to locate these services.
3. Provide adaptation layers (Gamma, 1994) for callbacks used by the container and ORB to inform the component about interesting events, such as messages from the Transaction or the Notification Service.
4. Manage POA policies to determine how to create component references.

Figure 5 shows the CCM container programming model in more detail.

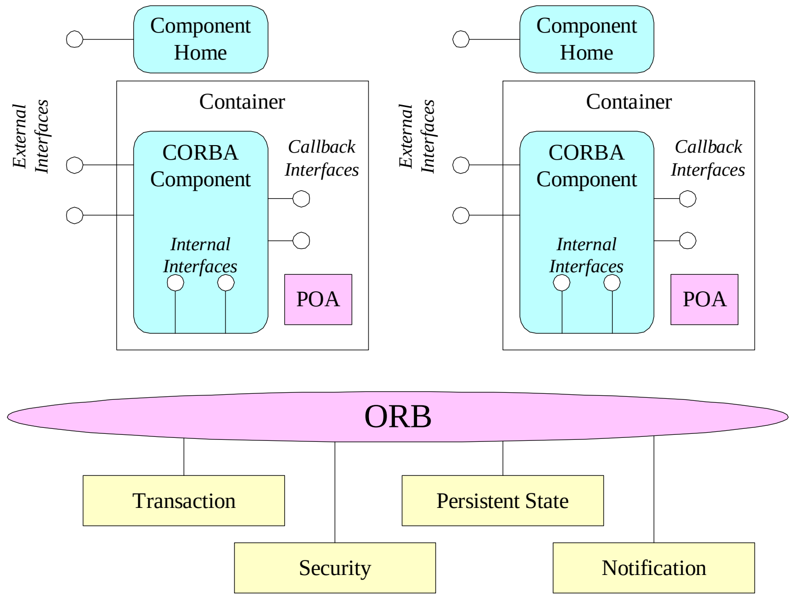


Figure 5: The CORBA Component Model's Container Programming Model

Clients directly access external component interfaces, such as the *equivalent interface*, *facets*, and the *home interface*. In contrast, components access the ORB functionality via their container APIs, which include the *internal interfaces* that the component can invoke to access the services provided by the container, as well as the *callback interfaces* that the container can invoke on the component. Each container manages one component implementation defined by the CIF. A container creates its own POA for all the interfaces it manages.

CCM containers also manage the lifetime of component servants. A CCM provider defines a `ServantLocator` that is responsible for supporting these policies. When a `ServantLocator` is installed, a POA delegates the responsibility of activating and deactivating` servants to it. Four types of servant lifetime policies control the timing of activating and deactivating components: *method*, *session*, *component*, and *container*. *Method* and *session* policies cause `ServantLocator`s to activate and passivate components on every method invocation or session, whereas *component* and *container* policies delegate the servant lifetime policies to components and containers, respectively.

In large-scale distributed systems, component implementations may be deployed across multiple servers, often using different implementation languages, operating systems, and programming language compilers. In addition, component implementations may depend on other software component implementations. Thus, the packaging and deploying of components can become complicated. To simplify the effort of developing components, CCM defines standard techniques that developers can apply to simplify component packaging and deployment. CCM describes components, and their dependencies using *Open Software Description* (OSD), which is an XML *Document Type Definition* (DTD) defined by the WWW Consortium. Components are packaged in assembly files and package descriptors are XML documents conforming to the Open Software Description DTD that describe the contents of an assembly file and their dependencies. A component may depend on other components and may require these components to be collocated in a common address space.

Related Technologies

CCM is modeled closely on the Enterprise Java Beans (EJB) specification (Thomas, 1998). Unlike EJB, however, CCM uses the CORBA object model as its underlying object interoperability architecture and thus is not bound to a particular programming language. Since the two technologies are similar, CCM also defines the standard mappings between the two standards. Therefore, a CCM component can appear as an EJB "bean" to EJB clients, and an EJB bean can appear as a CCM component by using appropriate bridging techniques. EJB also support CORBA IIOP as its communication framework. We believe the CCM and EJB are mutually complementary.

CCM and CORBA are also related to the Microsoft COM family of middleware technologies. Unlike CORBA, however, Microsoft's COM was designed to support a collocated component programming model initially and later DCOM added the ability to distribute COM objects. The most recent version of Microsoft's technology, COM+, includes commonly used business services, such as the Microsoft Transaction Service (MTS). The CORBA specification defines a bridging mechanism between CORBA objects and DCOM components. However, unlike CORBA and EJB, COM+ is limited mostly to Microsoft platforms.

Conclusion

The CORBA object model is increasingly gaining acceptance as the industry standard, cross-platform, cross-language distributed object computing model. The recent addition of the CORBA Component Model (CCM) integrates a successful component programming model from EJB, while maintaining the interoperability and language-neutrality of CORBA. The CCM programming model is thus suitable for leveraging proven technologies and existing services to develop the next-generation of highly scalable distributed applications. However, the CCM specification is large and complex. Therefore, ORB providers have only started implementing the specification recently. As with first-generation CORBA implementations several years ago, it is still hard to evaluate the quality and performance of CCM implementations. Moreover, the interoperability of components and containers from different providers is not well understood yet.

By the end of next year, we expect that CCM providers will implement the complete specification, as well as support value-added enhancements to their implementations, just as operating system and ORB providers have done historically. In particular, containers provided by the CCM component model implementation provide quality of service (QoS) capabilities for CCM components, and can be extended to provide more services to components to relieve components from implementing these functionalities in an *ad-hoc* way (Wang, 2000b). These container QoS extensions provide services that can monitor and control certain aspects of components behaviors that cross-cut different programming layers or require close interaction among components, containers, and operating systems. As CORBA and the CCM evolve, we expect some of these enhancements will be incorporated into the CCM specification.

References

- Object Management Group, *The Common Object Request Broker: Architecture and Specification*, version 2.4, 2000.
- Object Management Group, *Interoperable Naming Service Specification*, 1998.
- M. Henning and S. Vinoski, *Advance CORBA Programming with C++*, Addison-Wesley Longman, Reading, MA, 1999.

- D. C. Schmidt et. al., *Experience Developing an Object-Oriented Framework for High-Performance Electronic Medical Imaging using CORBA and C++*, Proceedings of the ``Software Technology Applied to Imaging and Multimedia Applications mini-conference" at the Symposium on Electronic Imaging in the International Symposia Photonics West, San Jose, CA, 1995.
- D. Levine et. al., *Dynamic Scheduling Strategies for Avionics Mission Computing,* Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC), Seattle, WA, 1998.
- C. O'Ryan and D. C. Schmidt, *Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation*, 5th International Workshop on Object-oriented Real-Time Dependable Systems, Monterey, CA, 1999.
- I. Pyarali, C. O'Ryan, and D. C. Schmidt, *A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware,* Proceeding of the IEEE/IFIP "International Symposium on Object-Oriented Real-time Distributed Computing", Newport Beach, California, March 15-17, 2000.
- D. Box, *Essential COM*, Addison-Wesley, Reading, MA, 1997.
- E. Gamma et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, MA, 1994.
- N. Wang et. al., *Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation*, 24th Computer Software and Applications Conference, Taipei, Taiwan, 2000a.
- N. Wang et. al., *Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications*, Reflective Middleware Workshop, ACM/IFIP, Pallisades, NY, 2000b.
- Object Management Group, Inc., *CORBA Component Model Joint Revised Submission*, 1999b.
- D. Schmidt et. al., *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Vol. 2.* Wiley & Sons, NY, 2000.
- Thomas, A. et. al., *Enterprise JavaBeans Technology*, 1998.

  URL: http://java.sun.com/products/ejb/white_paper.html

- Object Management Group, Inc., *CORBA Success Stories*, 2000.

  URL: http://www.corba.org/success.htm

- R. Marvie, P. Merle, Jean-Marc Geib, *Towards a Dynamic CORBA Component Platform*, The 2nd. Symposium of Distributed Objects & Applications, Antwerp, Belgium, September 2000.