# Integration of Object-Oriented Software Components for Distributed Application Software Development*

Stephen S. Yau and Fariaz Karim
Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287, USA
{yau, karim}@asu.edu

## Abstract

*The process of component integration for distributed application software development requires identifying the candidate components and performing compatibility checks based on the functional as well as non-functional requirements of the target application software. Since these requirements vary, it is important that distributed components themselves provide a set of specific services to facilitate component integration. In this paper, an approach to component integration for distributed application software is given. An object-oriented distributed component framework and a distributed connector model are presented to facilitate component integration.*

**Keywords**: component-based software development, integration, distributed system, framework, connector

## 1. Introduction

The component-based software development (CBSD) approach has shown significant promises in distributed application software development. Unlike traditional software development practices, CBSD focuses on construction of software, rather than programming. Although programming is still required at the implementation phase, CBSD removes the detailed programming task from software developers to component developers, who are usually well versed in specific problem areas. Then, distributed software development often becomes the tasks of a third party, which identifies a set of pre-developed components from a repository, possibly customizes them to fit specific requirements, and finally integrates them to build the application software. The CBSD approach thus facilitates better software reuse and higher productivity in software development.

Recent improvements in standardizing component and middleware specifications, such as COM, CORBA, JavaBeans, TINA-C Service Component, lead to standardized infrastructures and communication protocols, which facilitate the development and integration of object-oriented distributed software in a heterogeneous environment. Our previous work [1,2] was focused on transparent adapter generation for integrating heterogeneous and distributed software components to construct fault-tolerant distributed application software. A different approach was presented in [3], which separates the interactions among components from components themselves and integrates them based on a set of pre-determined connection points. A similar approach was adopted in [4], which also provides support for specification of distributed software architecture and customization of components through wrappers. A formal model was presented in [5], where the compatibility between a component and a connector is checked based on the nature of interactions each expects from the other.

Despite these advances, component integration in general continues to be a difficult task [6]. The problem becomes harder to solve in case of distributed software development due to the issues related to different non-functional requirements (e.g. types of fault-tolerance, event-handling, resource-management, etc.) in addition to the functional requirements of the target application software. Before the actual integration it becomes necessary to successfully check the compatibility of a component with the target distributed software architecture. Moreover, this aspect of integration becomes more complicated if the components are used as black boxes. Clearly, a mechanism is needed to automatically identify the compatibility of components with the target architecture based on the specified attributes. In this paper we will present a distributed component framework and a distributed connector model to facilitate such an integration mechanism.

## 2. Our Approach

Our approach to component integration for distributed application software includes an integration mechanism that performs compatibility check of a black box component with the target

*This work is supported under a collaborative research agreement between Arizona State University and Fujitsu, Ltd.

architecture based on both functional and non-functional requirements. We consider the following non-functional attributes for compatibility checks:

- Types of protocols used in the architecture
- Types of security protocols used
- Types of fault tolerance
- Types of event handling
- Types of exception handling
- Amount and types of resources used

Compatibility checks based on fault-tolerance, event, and exception handling are also useful if the target architecture has some real-time requirements and thus the compatibility of the candidate components need to be checked before being integrated into a distributed real-time software.

Our approach uses the Distributed Component Architecture (DCA) [1] as the common underlying environment to integrate and use distributed software components. An object-oriented distributed component framework is used to facilitate compatibility checks of components with the target architecture during integration time. Concerning the communication aspect, an object-oriented model of distributed connectors is used for connecting components and providing various communication Quality of Service (QoS).

The entire integration process can be summarized as follows:

1) Specify the target architecture of the distributed application software as a model adopted from I/O Automata [7].
2) Select the candidate components from repository based on desired functionality.
3) Perform non-functional compatibility checks on the components from 2) that are based on the component framework by supplying a part of the automaton from 1).
4) Identify the compatible components based on 2) and 3).
5) Customize the components from 4) that are not completely compatible based on the results in 3).
6) Visually integrate the components from 4) and 5) using distributed connectors to generate the distributed application software.

In this paper, we limit our discussions to the distributed component frameworks and distributed connectors (Sections 4 and 5), and their roles in the integration mechanism (Section 6). The details of the I/O Automata-based component and architecture models and the associated compatibility checks during integration will be covered in a future paper. We will use an example to illustrate the use of the component framework and distributed connector in developing a component-based distributed network management application. We will also discuss the implementation issues.

## 3. Desirable Properties of a Distributed Software Component

In addition to implementing the common interfaces [2], a distributed software component [1] should satisfy the following requirements to address the issues specified in the previous section:

a. Appropriate interface and corresponding implementation for checking compatibility during integration.
b. Suitable support for performing customization – both during integration and maintenance time. If customization is performed during maintenance, then support for on-line maintenance in a distributed environment is also needed.
c. Independent of any specific collaboration or interaction protocol for increasing reusability.
d. Support for hierarchically composing a component from a set of components to address complexity.

We will present a framework for distributed components to satisfy the specified properties. A framework is a set of cooperating classes that make up a reusable design of a specific class of software, which can be customized by application software developer [8]. On the other hand, a component framework, as described in [9], is a software entity that supports components conforming to certain standards and allows instances of components to be plugged into the component framework. Unlike the aforementioned definitions, the focus of our framework is at the granularity of individual components, which can be either atomic or composite. We define an *atomic component* as a component that does not encapsulate other components to implement its services. A *composite component*, on the other hand, is a container that encapsulates two or more components to implement its services. The encapsulated components are called the subcomponents of the composite component.

## 4. Distributed Component Framework

A distributed component framework is a reusable architecture that serves as a skeleton of a distributed component. The framework implements a set of specific services to facilitate various activities relating to a distributed component. The interfaces of the framework, as in a component, are represented in a common representation language, such as CORBA Interface Definition Language (IDL).

The framework satisfies the requirements [a-d] in the following way: It separates the run-time operations (i.e. service invocations) of a component from its integration and maintenance time operations (i.e. compatibility check, customization, etc.) by distributing them over five standard interfaces. To facilitate customization, the framework implements an architecture query mechanism to provide the information about its internal architecture at runtime. To provide protocol independence, it enforces a uniform method invocation style. Since the framework is itself a component, it can be instantiated, and then be integrated as a subcomponent into a larger and more complex component in a hierarchical fashion.

As shown in Figure 1, the architecture of the framework is divided into two layers:

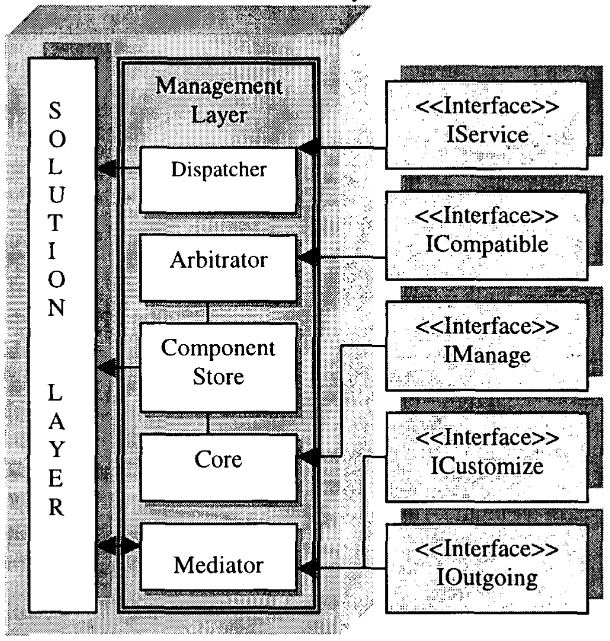

Figure 1: Simplified View of our Distributed Component Framework

*Management Layer:* The management layer is responsible for implementing the standard interfaces (described later) of a distributed component. It is mainly responsible for the following operations:

- Dispatch method invocations to the appropriate objects or subcomponents inside the framework
- Manage references and other information about the objects or subcomponents and connectors that reside in the Solution Layer.
- Perform compatibility checks based on the specification of a given target architecture
- Mediate the communication among the external plug-in components with the Solution Layer.
- Provide secure architectural-reflection to facilitate remote and on-line customization of a component.

*Solution Layer:* This layer consists of objects or components that actually implement the functionality of the component. If the component is atomic, then this layer only consists of objects. On the other hand, if the component is composite, then two or more subcomponents and connectors reside in this layer. The objects and components do not communicate with the management layer except to propagate any event or method invocation to an external component.

We now briefly describe the standard interfaces implemented by the framework. In combination, these interfaces actually provide the mechanism to communicate with a component from different viewpoints. Among the interfaces, the ICompatible, IManage, and ICustomize interfaces are crucial during component integration for their respective functionality. The IService and IOutgoing are mainly used when the component is already integrated and is ready for execution. The objects inside the management layer, shown in Figure 1, implement the services published in the interfaces.

- **IService:** The IService interface publishes the method-signatures of the services offered by the component that is built by instantiating the framework. Depending on the component, the interface also exposes appropriate properties for selecting a customized operation to configure the component for a specific QoS.

- **ICompatible:** This interface provides methods for performing non-functional compatibility checks of the associated component. The inputs are mainly passed as I/O Automata. The interface provides separate methods for checking compatibility based on each non-functional attribute mentioned in Section 2.

- **ICustomize:** The ICustomize interface is used to publish one or more plug-in interfaces of a component. An external component, which implements a plug-in interface, can be integrated with the component that provides the methods as published in the ICustomize interface. This allows a component to accept any third-party implementation during integration to extend its functionality.

- **IManage:** The IManage interface is used to facilitate more extensive customization activities that may be required during integration. In particular, it implements the *Architectural Reflection* service, which provides query facilities (called *QueryArchitecture* in the implementation) to retrieve a component's internal architecture for adding, deleting, or modifying subcomponents and subconnectors. A portion of the retrieved internal architecture consists of the references to the subcomponents and connectors. To support hierarchical analysis, these references can be used to recursively query the internal architecture of the subcomponents until no composite subcomponents can be found.

- **IOutgoing:** This interface allows a component to publish any required service that is expected from the underlying environment. In addition, the interface includes the methods for the events that the component may generate during its execution.

## 5. Distributed Connector

As mentioned in Section 3, a desirable property of a distributed component is to make it independent of any specific interaction protocol as much as possible to increase its reusability. We address this issue by using distributed connectors.

A *Distributed Connector* (DC) is a specialized component that encapsulates a particular interaction protocol, and provides specific interfaces to use it to connect a set of distributed components that needs to collaborate with each other using that particular protocol.

The principal benefit of using DC in distributed component integration is that it frees the components from implementing any complex interaction protocol, which makes the service interface of a component (e.g. IService in our case) simple.

Although the concept of a connector is well known for analyzing specific protocol in the software architecture community, in our case a DC performs the following additional responsibilities:

- Publishes the supported protocol through a standard interface for easy retrieval
- Provides a mechanism to transfer method invocations or event notifications among a set of integrated components.
- Implement the supported protocol through a collaboration of two or more distributed role objects
- Provides customized services, such as event ordering, encrypted communication, or other services with different QoS.
- Encapsulates actual communication protocol, such as TCP/IP or ATM.

As shown in Figure 2, the DC implements the following standard interfaces:

- **IEConnector:** This interface is mainly used when a DC is used to integrate two or more components. It implements the following functionality:

--Protocol Publishing: It provides I/O Automata-based description of the protocol that is encapsulated by the connector. This information is used by the integration mechanism to identify the suitability of the connector with the target architecture.

--Role Publishing: Since an interaction protocol is the outcome of two or more collaborating roles, *IEConnector* also publishes the specification of individual role. Each role is assigned to a specific component during integration.

--Customization: DCs expose properties that can be used by an application developer to select the

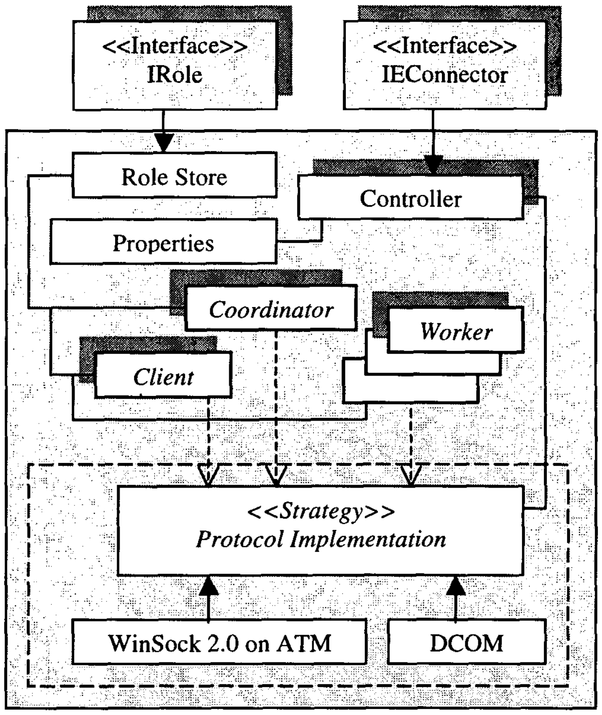

Figure 2: A sample distributed connector that implements the 2-phase Multi-server Commit protocol

appropriate communication QoS, such as turning on/off encryption, reliable event broadcast, priority-based event delivery, etc.
--Persistency: The state of a connector consists of the assignment of component to the roles and the mapping of methods among the integrated components. Such information is made persistent to reactivate an application without re-integrating the components and connectors at later times.

- **IRole:** This interface returns the references to the role objects that reside in a DC. After the connector is integrated into an application, the references are used by the associated components to collaborate with each other.

As described before, the protocol supported by a DC is implemented through the collaboration two or more *role* classes. Each role class can have multiple instances. The *Role Store* object implements the IRole interface, and stores the references to the role objects. The *Controller* implements the *IEConnector* interface. It is also responsible for setting the property values and connection information. The role classes do not implement any specific transmission protocol. Instead, the *strategy* pattern is used to decouple the actual implementation of the protocol from the references used by the role objects. Figure 2 shows two different implementations including the default DCOM protocol, and a WinSock2 implementation that exploits the communication QoS of an underlying ATM network.

## 6. Component Integration using DC

As described in [2], the integration tool plays the central role during component integration. The tool visualizes distributed components and generates adapters for resolving parameter mismatches and providing fault tolerance. In the following paragraph, we describe part of the integration mechanism, as it is used as a component integration tool. In particular, we describe how two or more components are integrated through a DC. The main task involves deciding which role a component will play to collaborate with other components. The choice of roles is restricted to the types of connector used in the integration. Once it is decided, the corresponding component and the role are integrated using the *Role-Embedding* procedure as follows:

1) The integration tool retrieves a reference to the role object from the corresponding DC using its IRole interface. To accomplish this, a role object implements a DCA-compatible interface. (e.g. a COM interface if the DCA is DCOM).
2) The reference from 1) is passed to the location of the component through the underlying DCA.
3) The component uses its IManage interface to store the reference to the role from 2).
4) A reference to the IService interface of the component is passed to the DC using the same mechanism described in 2).
5) During runtime, the DC uses the reference to the component from 4) to forward any method invocation to the component. Similarly, the component uses the reference to the role from 2) to dispatch its outgoing events.

After the procedure is applied, each component in the application software uses its assigned role object to communicate with other components. The integrated components produce an effect as if the interaction protocol were already built into each component.

## 7. An Example

In this section, we illustrate the use of distributed component framework and distributed connector. This example, which is implemented using DCOM on a 650 mb/sec Fujitsu ATM communication test bed, integrates a set of components based on the framework to develop a distributed network management system.

The main operation of a distributed network management system is to monitor a set of *network elements (NE)*, such as routers and gateways. The *NE*s generate different events that must be acknowledged and handled properly to keep the network free of congestion, or connected all the time.

The requirements include concurrent and prioritized processing of events, separation of event handlers from the status monitors, fast propagation of event-processing status, and others.

- Components and Connectors

Based on the requirements, three different components are used in the implementation: *Event Dispatcher (CED), Event Processor (CEP)*, and *Monitor (CM). CEDs* are installed on the NEs. They monitor the associated device, generate, and dispatch events to the *CEP* component. The *CEP* is responsible for processing the events, and forwarding the status to the CM component. The *CM* is responsible for providing GUI interface to present different information, such as the number of pending events and percentage of deadlines missed.

Due to the event-based interactions among the components, an Event-Notification (EN) connector is used to connect the *CEDs* with the *CEP*. The connector implements two roles – Event Source (ES) and Event Listener (EL). Accordingly, it includes two role objects with the same names.

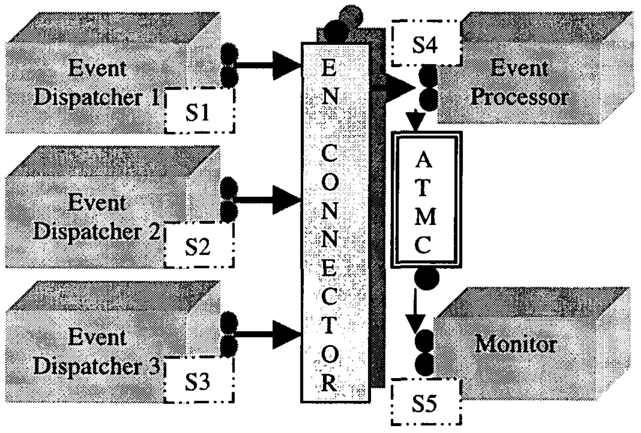

Figure 3: Component-based architecture of the example system

- Integration

As shown in Figure 3, three *CEDs* are considered in this particular example. They are distributed over three different sites S1, S2, and S3. The *CEP* and *CM* components are installed on sites S4 and S5 respectively. Integration is accomplished through embedding the roles of the EN with the appropriate component. Since there are three *CED* components, the connector is customized to provide three different instances of the ES role. The instances are installed on sites S1, S2, and S3. Similarly, the EL object is installed on S4. This object is integrated with the *CEP*, since the component only listens to events. To enable fast propagation of event-status, the *CM* component is integrated with the *CEP* through another EN connector (shown as ATMC in Figure 3), which is customized to provide access to ATM network through the WinSock2 communication library. To extend the functionality of the *CEP* to handle new types of events, new subcomponents are added through its IManage interface.

## 8. Discussion

Although our results are independent of any specific middleware, we use DCOM as the underlying DCA environment to implement the distributed component framework and the distributed connector. The framework is implemented as a COM component. Architectural reflection is implemented through the COM's dynamic interface discovery and invocation mechanism (*IDispatch*) and maintaining a data structure inside the framework that holds references to the subcomponents, subconnectors, and their interconnection information. Currently, the sub-components are implemented as out-process objects, although both in-process and remote components can be used seamlessly with the framework. The connector is also implemented as a COM component with additional capability to use WinSock 2 library to run on our high-speed communication test bed running on a 650 mb/sec Fujitsu ATM switch. Persistency of connector is achieved by implementing COM's *IPersistStream* interface.

An approach to component integration for distributed application software development is presented. The distributed component framework and distributed connectors, and their importance in the overall integration process are discussed. Future research includes validation of the constructed software with respect to the target architecture. The component framework will be extended to allow components to provide customized fault-tolerant services at the component level.

**Reference:**

[1] S. Yau and B. Xia, "An Approach to Distributed Component-based Real-time Application Software Development", *Proc. 1st IEEE Int'l Symp. Object-oriented Real-time Distributed Computing*, April, 1998, pp. 275-283.

[2] S. Yau and B. Xia, "Object-Oriented Distributed Component Software Development Based on CORBA", *Proc. 22nd Int'l Computer Software and Application Conference (COMPSAC 98)*, August, 1998, pp. 246-251.

[3] G. Wang, et. al, "Component Assembly for OO Distributed Systems", *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 71-78.

[4] M. Astley and G. Agha, "Modular Construction and Composition of Distributed Software Architectures", *Proc. Int'l Symp. on Software Engineering for Parallel and Dist. Systems*, 1998.

[5] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Trans. on Software Engineering and Methodology,* Vol. 6, No. 3, July 1997, pp. 213-249.

[6] D. Garlan, et. al, "Architectural Mismatch: Why Reuse Is So Hard", *IEEE Software*, Vol. 12, No. 6, November 1995, pp. 17-26.

[7] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms", *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing,* August 1987, pp. 137-151.

[8] R. Johnson, "Frameworks = (Components + Patterns)", *Comm. ACM*, Vol. 40, No. 10, October 1997, pp. 39-42.

[9] C. Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, 1997.