Towards Composable Distributed Real-time and Embedded Software

Krishnakumar Balasubramanian, Nanbor Wang, Chris Gill
{kitty,nanbor,cdgill}@cs.wustl.edu
Department of Computer Science
Washington University, St.Louis

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Abstract

The complexity of building and validating software is a growing challenge for developers of distributed real-time and embedded (DRE) applications. While DRE applications are increasingly based on commercial off-the-shelf (COTS) hardware and software elements, substantial time and effort are spent integrating these elements into applications. Integration challenges stem largely from a lack of higher level abstractions for composing complex applications. As a result, considerable application-specific "glue code" must be rewritten for each successive DRE application.

This paper makes three contributions to the study of composing reusable middleware from standard components in DRE applications: it (1) describes the limitations of current approaches in middleware composition, (2) discusses the minimum set of requirements required of reusable middleware components, and (3) presents recurring patterns for software composition as applied to CIAO, our open-source component model implementation.

1 Introduction

1.1 Emerging Trends

With the proliferation of enterprise component technologies, *e.g.*, the CORBA Component Model (CCM) [1], Microsoft .NET [2], and Enterprise Java Beans (EJB) [3], large-scale distributed applications are increasingly being developed and deployed in a modular fashion. Modularity elevates the level of abstraction used to program complex applications, encourages systematic reuse, and enhances maintainability over an application's lifecycle. Projects also increasingly rely upon commercial off-the-shelf (COTS) components and frameworks as the basis for their distributed software infrastructure.

Although reuse of primitive components is important, it is even more useful to compose them into higher-level components and ultimately into complete applications. Composition of software components is not as mature as assembly of hardware components (such as motherboards composed from integrated circuits) or mechanical components (such as automobiles or aircrafts components from standard parts). Nevertheless, in the long-term we expect it will be possible to develop complex software applications built largely by composing and customizing pre-existing components.

1.2 Key Challenges

Although component-based software development techniques are maturing for business and desktop systems, they are less mature for mission-critical domains, such as distributed real-time and embedded (DRE) applications. This paper focuses on the following challenges involved in QoS-enabled software composition in the context of emerging component models:

The need to reduce tight coupling of component metadata with component functionality. Component metadata describes the systemic information for running a component instance, such as the list of files used to implement the component, versioning information, or the required privileges for a component to function. Moreover, a reusable components for DRE applications can be applied in a variety of contexts with differing QoS requirements, such as the deadlines for various time-critical functionality, concurrency levels, and type of synchronization mechanisms. These systemic aspects that allow a component to be reused in multiple contexts need to be separated from component functionality into metadata and described in a manner that can be understood by component users and associated tools.

The need to specify component QoS requirements in a context-insensitive manner. A component in a DRE application may be functionally correct, yet can malfunction due to failure of assumptions stemming from the lack of context-dependent information [4], such as concurrency strategies, component lifetime (*e.g.*, persistent vs. transient), type of invocation (*e.g.*, synchronous or asynchronous), and the QoS of middleware services, such as event channels [5]. Without specifying and enforcing the QoS requirements in composition metadata, component execution environments make unstated assumptions on QoS properties components need and make it hard to enforce and adapt QoS properties in open systems.

The need to validate component implementation properties. A component implementation's properties (*e.g.*, the implementation language, version of the component, level of privileges required, and dependencies on other components)

must be validated. Validation is required for each individual component, as well as the application and system levels.

The need to ensure seamless deployment of a complex software system. To reduce the complexity of installing and maintaining complex applications, it is necessary that all the individual components be deployed using the same framework and follow the same guidelines. If each individual component needs a different mechanism for deployment, the costs of maintenance outweigh the advantages gained by developing applications in a component oriented fashion. It is also hard to track the dependencies of components upon other components and ensure that inter-dependent components are initialized in a particular order. To ease this task, components need to be packaged as a hierarchy that provides various information about the related components and captures dependencies present in component initialization and deployment. This packaging is necessary so that the deployment process can be automated, or at least controlled by an administrator.

1.3 Solution Approach

This paper describes how we address the challenges outlined in Section 1.2 in the implementation of Component-Integrated ACE ORB (CIAO), which is an extention to the CORBA Component Model (CCM) [1], as follows:

Reduced coupling by separating metadata from functionality. CCM specifies and CIAO implements a framework based on *eXtensible Markup Language* (XML) [6] mechanisms to define the grammar for describing component features. Our XML-based approach to describing component properties and systemic metadata makes components amenable to composition from (1) independent portions of a larger application and (2) future applications that can parse XML. This approach helps to decouple the functional aspects of a component-based application from the underlying QoS aspects and configuration details, thereby increasing composition flexibility and systematic reuse. In the CIAO project, we specify metadata for components via XML, using its content-agnostic metalanguage properties to express QoS configuration templates and conforming configuration files. Section 4.1 describes how we decouple metadata from functionality in CIAO.

Context-insensitive specification of QoS requirements. In CIAO, a component's dependencies are specified explicitly using metadata present with each component, thereby reducing the amount of implicit contextual information. This design helps make the implementation assumptions explicit, thereby ensuring that the environment in which the component executes can either satisfy the assumptions or fail gracefully. CIAO extends CCM XML Document Type Definitions (DTDs) to declare critical QoS parameters of component-based DRE applications and to specify properties of components defined by the CCM. There is considerable flexibility in specification of QoS requirements so that the requirements make sense from the perspective of a component, as well as from the end-to-end perspective needed for configuring a complete application. Section 4.2 describes how we support context-insensitive specification of QoS requirements in CIAO.

Validation of component configurations. After component properties are specified, their configurations must be validated at deployment time. In the CIAO project, default attributes are generated by a component-enabled Component Implementation Definition Language (CIDL) compiler (Section 3.1) as part of the metadata for every component. These attributes can be modified or extended by users. XML DTDs can be used to (re)validate metadata attributes *before* components are deployed, thereby avoiding exceptions during run-time. In addition, CIAO provide methods to validate (1) configurations of components, (2) privileges of components, and (3) QoS properties of the system both during and after an application is composed from a set of component building blocks. Section 4.3 describes how we validate component configurations in CIAO.

Component packaging and deployment. After specification and validation, component implementations need to be packaged so that they can be deployed. Packaging involves grouping the component implementations – typically stored in dynamic link libraries (DLLs) – together with metadata that describes properties of each particular implementation. Packaged components are in "passive mode," *i.e.*, all their functionality is present, but they are inert. To perform their functions at run-time, components must be made "active," and inter-connections between components established. Deployment mechanisms are responsible for transitioning components from passive to active mode. Section 4.4 describes how component packaging and deployment is performed in CIAO.

2 Overview of Components and Component Models

Some of the capabilities that are shared among most component models are as follows:

Multiple views of a component: Each component model specifies a set of interfaces that a component can export to its clients. These interfaces vary in the capabilities that they offer. It is therefore possible for a single component to play multiple roles to the component's clients at the same time. Moreover, a client can navigate from one view to another by using the introspection interfaces provided by the component.

Execution environment: Each component model defines an environment, known as a *container*, within which components can be instantiated and run. Containers shield components from low-level details of the underlying middleware. They are also responsible for locating and/or creating component instances, interconnecting components, and enforcing component policies, such as life-cycle, security, and persistence.

Component identity: Component models have mechanisms to identify their components uniquely. For example, .NET uses public key cryptography tokens to tag each component's interface and identify it uniquely across different software domains. EJB uses the Java Naming and Directory Interface (JNDI), which encapsulates low-level naming services such as LDAP, NIS, and DNS. EJB components are identified by hierarchical namespaces which use a directory naming scheme typically associated with an organization's Internet domain. The CCM uses DCE "universally unique ids" (UUIDs) to identify component implementations. Section 3 explains other capabilities that the CCM provides to identify components.

3 Overview of the CCM and CIAO

3.1 Key Capabilities of the CCM

The CORBA Component Model (CCM) is an Object Management Group (OMG) specification that standardizes the development of component-based applications in CORBA. Since the CCM uses CORBA's object model as its underlying object model, developers are not tied to any particular language or platform for their component implementations.

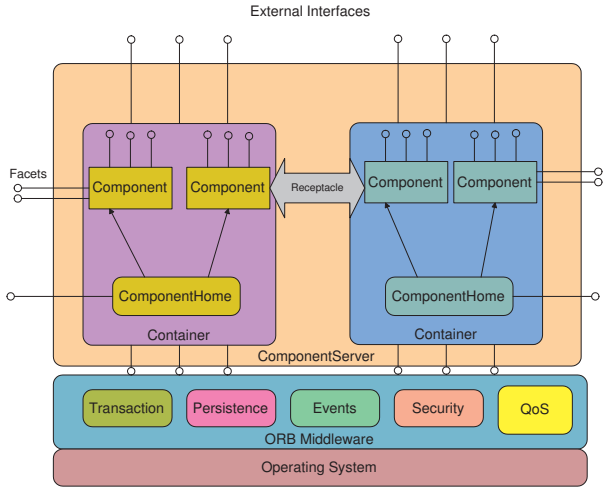

Figure 1: Key Elements in the CORBA Component Model

Key elements of the CCM include:

- **Component**, the basic building block used to encapsulate application functionality
- **ComponentHome**, a factory that creates and manages components
- **Container**, which provides components with an abstraction of the underlying middleware and regulate their shared access to the middleware infrastructure,
- **Component Implementation Framework**, which defines the programming model for constructing component implementations, using the Component Implementation Definition Language (CIDL) descriptions for automating generation of programming skeletons,
- **Component server**, which groups components and containers together to form an executable program.
- **ORB Services**, which provide common middleware services, such as events, security and persistence.

Figure 1 illustrates some of the above described elements. The remainder of this section explains why these elements are needed in the CCM by illustrating the key software development challenges they address.

3.1.1 Reusing Commonality in Software Applications

Context: A family of applications exhibiting commonality that can be refactored into reusable units, each of which offers specific functionality.

Problem: If application software is implemented in a monolithic fashion, it is hard to identify and refactor common functionality among related applications. Choosing module boundaries is hard without appropriate abstractions for describing functionality. Lack of functional abstractions leads to unnecessary duplication across different modules and prevents systematic reuse.

CCM Solution → Component: Define a *component* abstraction that serves as the building block for the structure of software applications, as well as the candidate for demarcating modularity and functionality. The capabilities of a CCM component are defined using extensions to OMG's CORBA 2.x Interface Definition Language (IDL).

3.1.2 Deoupling Components and Middleware

Context: Development of component software relies on services provided by the middleware.

Problem: In earlier generation middleware based on object models, programmers were responsible for connecting to and configuring the policies of the underlying middleware. For example, before the advent of the CCM, CORBA developers had to explicitly bind to and configure the policies of middleware entities such as event channels and security services. These manual programming activities required developers to (re)write substantial amounts of "glue-code," which was often

larger than that required to use the functionality. These activities were error-prone since they required application developers to manage low-level details of the underlying middleware.

CCM Solution → Containers: The *container* abstraction provides the context in which components run. A container acts as a bridge between the low-level middleware and a component by interacting with underlying middlware based on the policies defined in the component. A container also provides an execution environment for components, *e.g.*, it defines interception points where various run-time policies such as security can be imposed and validated. Components can also use the capabilities provided by the containers to avoid undue dependence on the underlying middleware.

3.1.3 Specifying Component Interconnections

Context: A complex system consisting of individual components that must interoperate with each other at run-time.

Problem: A component can provide functionality at different granularities. In software developed using object models, a one-to-one association typically exists between an object and the roles played by the object *i.e.*, a user of an object either gets all the functionality and the artifacts of that functionality or nothing. In complex software applications, however, a one-to-one association of component and component roles can result in an unwieldy proliferation of interfaces that must be managed explicitly by client application developers.

CCM Solution → Ports: Define a *port* abstraction that can expose multiple views of a component to clients, based on context and functionality. CCM ports define a set of connection points between components to expose various roles supported by a component interface. These port mechanisms specify the interaction model among interdependent components.

3.1.4 Configuring Components

Context: A system where a component needs to be configured differently depending on the context in which it is used.

Problem: As the number of component configuration parameters and options increase, it can become overwhelmingly complex to configure applications consisting of many individual components. The problem stems not only from the number of alternative combinations, but also from the disparate interfaces for modifying these configuration parameters. Object models have historically required application developers to write large amounts of "glue code" to interconnect and configure components. In addition to being tedious and error-prone, this coding process exposes the application developers to low-level details of the underlying middleware.

CCM Solution → Assembly: Define an *assembly* abstraction to group components and characterize their metadata that describes the components present in the assembly. Each component's metadata in turn describes the features available in it (*i.e.*, properties) or the features that it requires (*i.e.*, a dependency). After an assembly is defined, the task of modifying the parameters need not involve writing glue code. Instead, metaprogramming techniques [7] can be applied to generate code to configure the component in a context-dependent fashion, due to the decoupling of the properties of components and the code needed to configure these properties in the components.

In CCM assemblies are defined using XML DTDs, which provide an implementation-independent mechanism for describing component properties. With the help of these XML DTD templates, it is possible to generate default configurations for CCM components. These configurations can preserve the required QoS properties [8] and establish the necessary configuration and interconnection among the components.

3.1.5 Resolving Dependencies Automatically

Context: Run-time deployment of distributed applications built using components as the core software building blocks.

Problem: Any non-trivial software system consists of components with various dependencies, such as reliance on specific other components, order of component initialization, or domain-specific requirements (*e.g.*, required sensor rate in the avionics domain [9]). Resolving these dependencies manually does not scale as the number of components in a system grows. Likewise, ignoring or underspecifying these dependencies can result in an unstable system if the system run-time assumes that components are independent and then instantiates them in an invalid order. For example, the wheels of a carrier-based fighter aircraft must open before the aircraft tries to land.

CCM Solution → Deployment application: Define a *deployment application* that is responsible for managing the dependencies among interdependent components. A deployment application can ensure that component interconnections are established correctly and in the right order by using metadata that capture these dependencies, along with information about the interconnections expressed via CCM ports.

3.1.6 Evolving Component Software

Context: Software applications that have been partitioned into many individual components.

Problem: Although partitioning a system into a collection of individual components avoids the many problems discussed in Section 3.1.1, it can be a maintenance problem. For example, the person-hours needed to evolve complex applications increases considerably as the number of individual components in a system increases. This problem is exacerbated by the fact

that it is hard to determine the relationship between a component and its running context solely based on the presence of a component in a live system.

CCM Solution → Component servers: Define a *component server* abstraction responsible for aggregating the "physical" (*i.e.*, implementation of component instances) entities into "logical" (*i.e.*, functional) entities of a system. A component server is equivalent to a server process in an object model.

3.2 Key Capabilities of CIAO

The *Component-Integrated ACE ORB* (CIAO) developed at Washington University, St. Louis extends the CCM. CIAO is designed to bring the component-oriented development paradigm to DRE application developers by abstracting DRE-critical systemic aspects, such as real-time policies, as installable/configurable units. Promoting these DRE-critical aspects as first-class metadata disentangles the code that controls these systemic aspects from application logic. It also makes is easier to compose components into DRE applications flexibly. Since mechanisms to support various DRE-critical systemic aspects can be validated using tools that analyze and synthesize these aspects at a higher level of abstraction, CIAO also makes configuring and managing these aspects easier [10].

The CIAO implementation is based on TAO, our open-source, high-performance, highly configurable Real-time CORBA ORB that implements key patterns [11] to meet the demanding QoS requirements of distributed applications. CIAO enhances TAO to simplify the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system. Figure 2

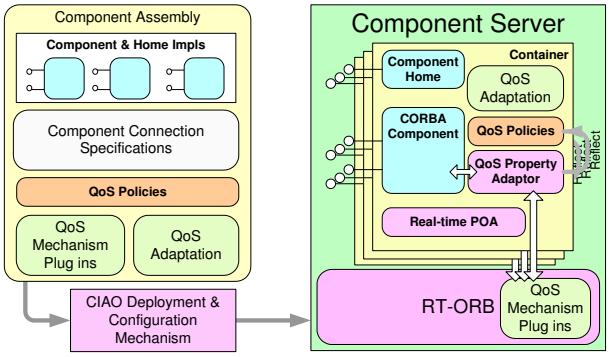

Figure 2: **Key Elements in CIAO**

shows the key extensions to the CCM in CIAO, which include:

Component assembly: CIAO extends the notion of component assembly to include server-level QoS provisioning and implementations for required QoS supporting mechanisms. CIAO's extended assembly descriptor definition also enables specification of QoS provisioning to connect components.

QoS-aware containers: CIAO's QoS-aware containers provide a centralized interface for managing provisioned component QoS policies and interacting with QoS assurance mechanisms required by the QoS policies.

QoS adaptation: CIAO also supports installation of meta-programming hooks, such as Portable Interceptor and smart proxies [7], which can be used to perform dynamic QoS provisioning behaviors that provision QoS resources and adapt applications to changes in system QoS.

Application developers can use CIAO to decouple QoS provisioning from component implementations and assemble a DRE application by composing and connecting application components, QoS specifications, and reusable QoS adaptation behaviors together. Section 4 describes how CIAO addresses the challenges of assembling and deploying components.

4 Addressing Key Design Challenges for Composable DRE Applications

As described in Section 3, the CORBA Component Model (CCM) specifies the core infrastructure needed for component-based software development. That section also explains how the CCM provides capabilities that help them develop composable middleware and applications. The capabilities offered by the CCM, however, are targeted towards enterprise and desktop applications, which do not possess key challenges inherent to developing DRE applications.

To address the challenges in developing components for DRE applications effectively, we have extended the CCM specification in CIAO to allow specification of component properties that are critical to support DRE applications with stringent QoS requirements. Specifically, CIAO enhances the CCM to support static QoS provisioning, which allocates resources at various levels in a distributed system *a priori*. This capability is useful when DRE application components need to provide hard real-time guarantees or to simplify the specification of QoS as part of a large system. In CIAO, specification of static QoS provisioning is acheived via extensions to metadata using XML. Through these extensions, key QoS related properties of the TAO Real-time CORBA ORB are exposed to developers of DRE components and applications. The remainder of this section describes how CIAO addresses the challenges described in Section 1.2.

4.1 Reducing Coupling by Separating metadata from Functionality

Context. Developing DRE middleware that have considerable amount of systemic metadata.

Problem. DRE middleware has traditionally contained considerable metadata, *i.e.*, information that describes systemic characteristics. As identified in Section 1, these metadata do not implement application functionality *per se*. They are nevertheless important to the proper functioning of an application. There are two common problems with metadata: 1) tangling of metadata with component implementations leads to excessive coupling between the two and can impede application evolution, and 2) specifying metadata in an *ad hoc* manner prevents interaction with components developed using other non-compatible metadata specification mechanisms. Together, these two factors present the challenge that individual components may function satisfactorily, but the composition of these components into higher-level applications may not meet systemic QoS properties such as time and space constraints. This problem arises from freezing the interoperability options prematurely, *i.e.*, at the end of the component design cycle rather than during the application integration cycle.

Solution → Use a meta-language, *e.g.*, XML DTD, to describe metadata. Describe component metadata separately from the implementation of the component functionality. The language used to define metadata should be extensible to allow the specification of metadata that is open-ended and subject to change. Designing a language for extensibility [12] involves tradeoffs (such as level of expressibility, ease of adding new features, and maintaining backward compatibility) that must be handled carefully. XML-based meta-language is used to describe DRE application metadata, while avoiding the effort required to design a full-fledged language. Using XML to specify component metadata enables designers and integrators of DRE applications to separate metadata from the component implementations, while also enabling the integration and composition of third-party code.

Applying the solution in CIAO. CIAO uses ACEXML, which is an open-source C++ library for parsing XML files. ACEXML provides an API based on the Simple API for XML (SAX) [13] to assist in handling XML used for the specification of metadata. There are two types of XML APIs: 1) tree-based APIs, which map an XML document into an internal tree structure and allow an application to navigate that tree, and 2) Event-based APIs like SAX, which report parsing events directly to an application via callbacks but do not usually build an internal tree. During deployment (Section 4.4) ACEXML reads the metadata from an assembly and uses it to validate (Section 4.3) the contents of the assembly. Since DRE applications often have stringent footprint requirements, they cannot afford to build the entire tree in memory. This problem is exacerbated if the amount of metadata becomes large, such as when metadata is auto-generated by component-aware IDL compilers. ACEXML is therefore based on SAX, and does not build the entire tree in memory.

4.2 Context-insensitive Specification of QoS Properties

Context. Designing component-based DRE applications that rely on underlying middleware to provide multiple levels of QoS assurance to the application, including minimum/average/maximum latency and throughput guarantees, supported sensor rates, default number of network packets queued, maximum size of an allowed packet, and allowed minimum/average/maximum deadlines.

Problem. Building complex DRE applications exposes developers to variations in: 1) the implementation of QoS enabling mechanisms, such as scheduling algorithms, thread pools, connection pooling and caching and event demultiplexing provided by the underlying middleware, and 2) the number of such alternative QoS enabling mechanisms that are exposed to the user as configurable values. This variation can encourage developers to design applications that depend on some or all of the QoS enabling mechanisms outlined above to be provided by the underlying middleware and made available to the component. Critical QoS requirements may not be met when components are used in a scenario where such QoS enabling mechanisms are either unavailable or insufficient to satisfy the design assumptions. Depending on the criticality of the missed QoS property, there might be a localized malfunction or a failure of the entire application.

Solution → Specify QoS properties in a context-insensitive fashion. Identify properties of a component (*i.e.*, the set of configurable values) that when set in a particular fashion affect the state and hence the behavior of the component. Specify the properties such that the task of manipulating them is separate from the functionality of the component. Care should be taken to ensure that the amount of context-dependent assumptions is limited, and if present, the dependency on such assumptions are made explicit. It is also important that the specification of these QoS properties, makes it possible to fully exploit additional QoS capabilities, present in some but not all implementations of the underlying middleware.

In general, QoS properties should play a first-class role in the middleware typesystem and be associated with components explicitly. Doing so can also prevent errors during composition by recognizing mismatches in provided and required properties, as explained in Section 4.3. In the long run, standardizing common QoS properties of underlying middleware, from different vendors, is important to ensure interoperability, as well as to enhance the reuse of QoS-aware components.

Applying the solution in CIAO. CIAO extends the CCM *component property file* that specifies the QoS properties that are essential to static QoS provisioning, such as size of the input buffers to allocate, portion of the network bandwidth to reserve, and priority of the packets sent out by this component.

Developers of components based on CIAO can use and configure these properties of the underlying middleware. They can also expose them to other components by defining a mapping between middleware and component properties.

The component property file is a XML-based vocabulary that is read at deployment time and used to configure the component. By explicitly specifying the properties and separating them from the component functionality, CIAO allows the context-insensitive specification of these properties. By removing the specification and manipulation of these properties from the functional properties of the component, CIAO also reduces the amount of tedious and error-prone glue-code that must be written to configure components.

4.3 Validation of Component Configurations

Context. Integrating a complex DRE application from a set of generic and reusable COTS components.

Problem. Developers of reusable COTS components must validate that their implementations satisfy the intended functionality and QoS. A common validation procedure is black-box or whitebox testing [14]. While this validation process yields readily available and tested components, the task of integrating these components and configuring them to customize an application is hard. In particular, manually integrating COTS components is error-prone since it involves 1) checking QoS properties of each individual component to ensure that the component satisfies local requirements and 2) ensuring that the overall system composed of these individual components satisfies the QoS guarantees.

Solution → Validate component configurations. Validate component configurations by checking the metadata associated with a component to ensure that the end-to-end requirements of the application match the capabilities offered by its constituent components. This validation process does not include mechanisms to check whether the functionality advertised by a component is indeed provided by the component. The topic of verifying semantics of a component [15] is vast and merits a detailed discussion [16] of its own.

Validation can be done by using XML-based descriptors, which contain metadata that describes the systemic properties of individual components, component packages, or component assemblies (Section 4.4). The formats of these descriptors are specified via a set of XML DTDs. Validation of metadata involves checking for conformance with the rules specified *a priori* in the DTD. However, this process is only effective when automated and not exposed to human errors. If validation is conducted during deployment (Section 4.4), it can avoid exception conditions after the application is deployed.

Applying the solution in CIAO. CIAO's implementation of the CCM CIDL compiler generates a default configuration for every component and hence a default descriptor. In many use-cases, however, a descriptor may need to be modified and extended by component developers to better suit their requirements or to impose certain policies on components. After a default descriptor generated by the CIDL compiler is modified or extended by a developer (or if a descriptor is specified from scratch by a developer), it is essential to check if the descriptor still conforms to the descriptor's DTD. Descriptors are validated for conformance with their DTDs using the ACEXML library presented in Section 4.1, which provides a general-purpose tool to validate any XML DTD.

4.4 Component Packaging and Deployment

Context. Deploying a DRE application that is built from reusable COTS components.

Problem. In complex DRE applications, there may be hundreds or even thousands of these components. With so many components, it is hard to manage the application or to specify provsioning at the granularity of individual components. Furthermore, some QoS properties cross-cut component boundaries, so they must be handled at multiple levels of granularity. Supporting static provisioning of QoS therefore becomes harder in the presence of a large number of components.

Solution → Use component assemblies . Specify QoS properties at multiple levels of abstraction to support static provisioning of QoS in an end-to-end fashion. To support specification of QoS properties at multiple levels, component software needs to be packaged in a suitable hierarchical format. This format should also allow specification of QoS policies, which assist in overriding a particular property to maintain end-to-end guarantees. Policies are specified in conjunction to the specification of QoS properties. The levels of abstraction at which the QoS properties can be specified include: 1) the component software package, which contains one or more implementations of a component with an associated descriptor, and 2) the component assembly package, which contains a set of inter-dependent components and information that describes the dependencies between these components.

The use of XML for the descriptors at each level not only serves as a "glue-language" for composition, but also enables the development of value-added services, such as graphical user interface (GUI)- based packaging tools, that are independent of the components or the application.

Applying the solution in CIAO. In CIAO, a component software package is described by a *CORBA software descriptor* that captures the high-level details of components present in a software package, such as ownership information along with a list of implementations of components. Each implementation in turn describes features, such as type and version of of the OS and CPU, along with the type(s) of component present in the implementation.

Each type of component within an implementation is described by a *CORBA component descriptor* that captures the structure of a component, *i.e.*, its supported interfaces, inherited components, and ports. CIAO uses component descriptor files to facilitate inter-connections between components.

A *component assembly descriptor* file describes the components that make up the assembly, how those components are partitioned, and how they are inter-connected. The CIAO deployment mechanism consults the component assembly descriptor file to bootstrap the deployment.

In CIAO, an instance of a daemon process (called `compassd`) runs on every host that will participate in the deployment. This daemon acts as the manager for the components that are installed on a particular host. A new component can be installed by specifying the file containing the implementation along with the hostname and port number where the component has to installed. If another implementation of the same component is already running on a particular host (determined by comparing the UUID of each component implementation), the daemon will ensure it is not installed again.

5 Concluding Remarks

Composable middleware for distributed real-time and embedded (DRE) applications can provide benefits to developers of both DRE middleware and applications, as well as DRE application integrators. This paper describes how our work on the Component-Integrated ACE ORB (CIAO) addresses key challenges that arise when applying state-of-the-practice component model technology to DRE applications. We also describe the CORBA Component Model (CCM) specification and then describe enhancements to the CCM we have implemented in CIAO. By applying the solutions described in this paper, we are decoupling various aspects of DRE software applications, thereby enabling application developers, system engineers, and end-users to select components that can be composed to build complete DRE applications with a shorter time-to-market. Our long-term goal is to provide the same benefits available to developers of desktop and enterprise applications to the much more challenging domain of DRE applications.

The long-term goal of the work described in this paper is to enable reflective ORB behavior and expose these ORB features so that they can be monitored and controlled effectively by higher-level tools and management applications. ACEXML used in the deployment framework of CIAO is available from the ACE CVS repository available at http://cvs.doc.wustl.edu/viewcvs.cgi/ACEXML/.

References

[1] Object Management Group, *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03 ed., Nov. 2001.

[2] Microsoft Corporation, "Microsoft .NET Development." msdn.microsoft.com/net/, 2002.

[3] Sun Microsystems, "Enterprise JavaBeans Specification." java.sun.com/products/ejb/docs.html, Aug. 2001.

[4] N. Wang, K. Balasubramanian, and C. Gill, "Towards a real-time corba component model," in *OMG Workshop On Embedded & Real-Time Distributed Object Systems*, (Washington, D.C.), Object Management Group, July 2002.

[5] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, Oct. 1997.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds), "Extensible Markup Language (XML) 1.0 (2nd Edition)." W3C Recommendation, 2000.

[7] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware," *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, pp. 102–113, Oct. 2001.

[8] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, pp. 45–54, mar 2003.

[9] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[10] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, Oct. 2002.

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[12] J. Guy L. Steele, "Growing a language," *Journal of Higher-Order and Symbolic Computation*, vol. 12, pp. 221–236, Oct. 1999.

[13] SAX Project, "Simple API for XML." www.saxproject.org, 2002.

[14] G. J. Myers, *The Art of Software Testing*. John Wiley and Sons, 1979.

[15] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, (White Plains, NY), pp. 234–245, June 1990.

[16] S. Easterbrook and J. Callahan, "Formal methods for verification and validation of partial specifications: A case study," *The Journal of Systems and Software*, vol. 40, pp. 199–??, March 1998.