Composition in the CORBA Component Model

Krishnakumar Balasubramanian
kitty@cs.wustl.edu
Department of Computer Science
Washington University in St.Louis
St. Louis. MO

Abstract

The growing complexity of building and validating software is a challenge for developers of distributed real-time and embedded (DRE) systems. While building blocks of DRE systems are increasingly based on commercial off-the-shelf (COTS) components, substantial time and effort are spent integrating components into systems due to the lack of higher level abstractions for composing complex systems. As a result, considerable system-specific "glue code" must be written, only to be rewritten from scratch when building subsequent systems.

This paper provides four contributions to the study of composing reusable middleware from standard components in DRE systems: (1) it analyzes the problems with current approaches in middleware composition, (2) it quantifies the minimum set of requirements required of reusable middleware components, (3) it presents recurring patterns in the domain of software composition and provides empirical evaluation of these patterns as applied to TAO, our open-source second-generation Real-Time CORBA Object Request Broker (ORB), and (4) it compares our approach to other research done in the area of software composition. Our results show that ...

1 Introduction

Emerging trends. Software components are units of independent production, acquisition, and deployment that interact to form a functioning system [1]. Software component models, such as COM+ and Javabeans, have long formed the basis for graphical user interfaces and other stand-alone system developments. A component model is responsible for:

- Describing the properties and semantics of component building blocks and
- Keeping the context sensitive dependencies to a minimum and defining these explicitly when necessary.

With the proliferation of enterprise component technologies, such as the CORBA Component Model (CCM), .NET, and Enterprise Java Beans (EJB), large-scale distributed applications are increasingly being developed and deployed in a modular fashion. Modularity elevates the level of abstraction used to program complex systems, encourages systematic reuse, and enhances software maintainability over the system lifecycle. Projects are also increasingly relying upon commercial off-the-shelf (COTS) components and frameworks as the basis for their distributed software infrastructure.

Although reuse of individual components is useful, it is even more useful to compose the individual components into higher-level reusable components and even 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, it should be possible to develop complex software applications built largely by composing and customizing pre-existing components.

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) systems. In DRE systems, composition of component functionality alone is not sufficient since these types of systems must ensure end-to-end quality of service (QoS), of which components are simply the basic building blocks. Mixing QoS specifications and enforcement mechanisms with the application functionality results in complications that tend to grow rapidly as the number of components – and hence the complexity – of the software increases.

This paper focuses on the following challenges involved in QoS-enabled software composition in the context of the emerging component models:

• **The need to minimize overly tight coupling of component meta-data with component functionality.** To reuse a component in more contexts than it was designed originally, the component's functionality and its feature set need to be described in a manner that can be understood by component users. Component meta-data includes (but is not limited to) information such as the list of files used to implement a component, version number information, a checksum to ensure component integrity, and information about the required privileges for this component to function. Various composition problems occur when component meta-data is described at the same level of abstraction as the component functionality. In particular, this tight coupling can require applications to be

written in the same language as the lower-level components, which may not be feasible if these entities have been developed independently and at different points in time.

• **The need to specify component QoS requirements in a context-insensitive manner.** Composition problems arise when QoS requirements of components are specified with implicit assumptions on properties of external entities (such as threading models required by a component or support for concurrency needed to run the component) outside a particular component of a software system. Unstated or underspecified assumptions related to QoS properties of components make it hard to use the requirements effectively. The context in which these assumptions hold true are provided by the environment in ideal conditions. A component can malfunction due to failures of assumptions stemming from the lack of context-dependent information.

• **The need to validate component properties.** A component implementation's properties (such as the implementation language, version of the component, level of privileges required, and dependencies on other components) must be validated against its specification to avoid problems such as (1) errors caused by misconfiguration, (2) attacks by malicious components that request resources from the underlying ORB improperly, and (3) lack of confidence that the QoS assurances provided by the middleware are sufficient from an application's perspective. Validation is required at the granularity of an individual component, as well as at the system level.

• **The need to ensure that a complex software system can be deployed seamlessly.** It is hard to track the dependencies of components upon other components and ensure that interdependent components are initialized in a particular order. To ease this task, components need to be packaged as entities that provide a variety of information about the resident components and capture the dependencies present in initialization. This packaging is necessary so that the deployment process can be automated completely or at least controlled by an administrator.

Solution approach. Early ORBs did not provide features or optimizations to support the challenges of component-based distributed systems described above – particularly not for DRE systems with stringent QoS requirements. To better meet these requirements, we have developed a third-generation ORB called the Component-Integrated ACE ORB (CIAO) [2], which is based on the CORBA Component Model (CCM) [**?**] specification that standardizes the development of platform- and language-independent component-based applications. CIAO extends our previous work on The ACE ORB (TAO) [3] by providing more powerful component-based abstractions using the specification, validation, packaging, and deployment techniques discussed above.

Prior work on TAO has explored many dimensions of high-performance and real-time ORB design and performance, including scalable event processing [4], request demultiplexing [5], I/O subsystem [6] and protocol [7] integration, connection architectures [8], asynchronous [9] and synchronous [10] concurrent request processing, adaptive load balancing [11], meta-programming mechanisms [12], and IDL stub/skeleton optimizations [13]. This paper describes how we have extended CIAO to avoid the various challenges that arise when developing flexible and high-performance DRE systems. Specifically, the CIAO project addresses the challenges outlined earlier as follows:

• **Reduced coupling by separating meta-data from functionality.** We provide a framework based on *eXtensible Markup Language* (XML)-based [14] mechanisms to define the grammar for describing component features. The XML-based approach makes components amenable to composition by (1) independent portions of a larger system and (2) future applications that can parse XML. This results in a decoupling of the functional aspects of a component-based system (that can be written using a variety of COTS programming languages) from the underlying QoS aspects and configuration details. This decoupling increases composition flexibility and systematic reuse. In the CIAO project, we specify meta-data of a component via XML, using its content-agnostic metalanguage properties to express QoS configuration templates and conforming configuration files.

• **Context-insensitive specification of QoS requirements.** We identify critical QoS parameters of component-based DRE software systems and specify them using extensions to the XML Document Type Definitions (DTD) for specifying properties of components defined by CCM. There is considerable flexibility in the extension so that the requirements make sense from the perspective of a component, as well as from the end-to-end perspective needed for the system as a whole. All of a component's assumptions are explicitly specified using meta-data and are present within each component, *i.e.*, the component is context-insensitive, and the amount of implicit contextual information is minimal.

• **Component Validation.** After a component is specified and packaged, it must be validated at deployment time. In the CIAO project, default attributes are generated by a component-enabled OMG Interface Definition Language (IDL) compiler as part of the meta-data for every component. These attributes can be modified or extended by the user. XML Document Type Definition (DTDs) can be used to (re)validate meta-data attributes *before* components are deployed, thereby avoiding exceptions during run-time. We 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.

• **Component packaging and deployment.** After specification and validation, component implementations must be packaged and deployed. As shown in Figure 1, packaging involves grouping the implementation of component functionality, which is typically stored in a dynamic link library (DLL), with other meta-data that describes properties of this particular implementation. Packaged components are in "passive

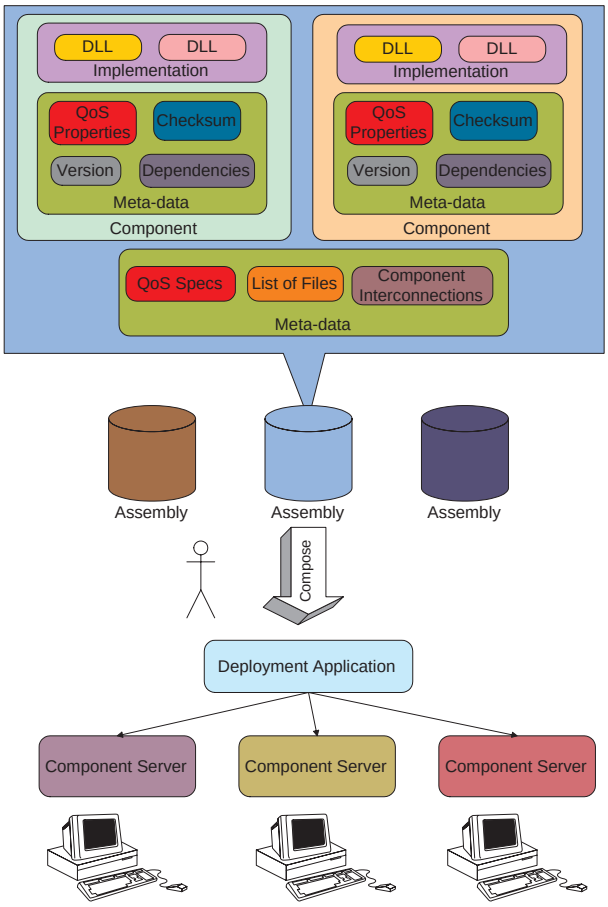

Figure 1: Component Packaging and Deployment

mode," *i.e.*, all their functionality is present, but they are inert object code. To function at run-time, components must transition to "active mode," where the inter-connection between components is established. Deployment mechanisms are responsible for transitioning components from passive to active mode.

Paper organization. This paper is organized as follows: Section 2 first gives a general overview of component models; Section 3 then focuses motivating and describing the capabilities of the CORBA Component Model (CCM) and CIAO; Section 4 examines in detail the techniques used in CIAO to solve key challenges with component models and CCM; Section 5 compares and contrasts our work on software composition in CCM and CIAO with other approaches; and Section 6 presents concluding remarks.

2 Overview of Components and Component Models

In the early days of computing, software was developed from scratch to achieve a particular goal on a specific hardware platfrom. Since computers were much more expensive than the people who programmed them, relatively little attention was paid to systematic software reuse and composition of applications from existing software artifacts. Over the past four decades, the following two general trends have spurred the transition from hardware-centric to software-centric development paradigms:

- **Economic factors** – Due to advances in VLSI and the commoditization of hardware, most computers are now *much* less expensive than the people who program them.
- **Technological advances** – With the advent of object-oriented programming languages and distributed object computing technologies, software can now be developed in a much more modular fashion.

This section provides an introduction to components and component models. We begin with an overview of the software paradigms that culminated in component-based software technologies and outline a promising new enhancement to component models. We also describe the functionalities shared by all component models and show how component models differ from the popular object models.

2.1 A Brief History of Software Programming Paradigms

Below, we present a brief history of software programming paradigms. A common theme underlying all of these paradigms is the desire to compose and customize systems largely from pre-existing software building blocks. What differs is the types of building blocks envisioned for each paradigm.

Information hiding and data abstraction. Composing software from reusable artifacts has been a goal of software researchers for over three decades. For example, Doug McIlroy [15] motivated the need for software "integrated circuits" (ICs) and mass-produced software ICs, as well as examines the types of variability needed in software ICs and the types of ICs that can be standardized usefully. McIlroy envisioned an IC to be a standard catalogue of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters.

Early efforts [16, 17, 18] to realize the vision of software ICs resulted in information hiding and data abstraction techniques that placed more emphasis on organization of data than the design of procedures. Data abstraction resulted in formalizing the concept of *modules* as a set of related procedures and the data that they manipulate, resulting in partitioning of programs so that data is hidden inside them.

These techniques were embodied in programming languages, such as Clu, Modula 2, and Ada. They provided modules as a fundamental language construct apart from explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage of the above mentioned features. Although these languages allowed programmers to create and apply user-defined types, it was hard to extend these types to new usages scenarios without modifying their interface definitions and implementations.

Object-oriented techniques. The next major advance in programming paradigms came from object-oriented design techniques, such as the OMT and Booch notations/methods, and object-oriented programming languages, such as C++ and Java. Object-oriented techniques focus on decomposing software systems into classes and objects that have crisply-defined interfaces and are related via inheritance and aggregation. A key advantage of object-oriented techniques is their direct support for the distinction between an class's general properties and its specific properties. Expressing this distinction and taking advantage of it programmatically was simplified by object-oriented language support for inheritance, which allows the commonality in class behavior to be explicit, as well as allowing customization of this behavior by allowing redefinition of methods in subclasses.

Component-based techniques. Although object-orientation represented an advance over previous programming paradigms approaches, it also had certain deficiencies. For example, object-oriented techniques are based on assumptions that different entities in a software system have interfaces that are (1) amenable to inheritance and aggregation and (2) are written in the same programming langauge. Many applications must run in multi-lingual and even multi-paradigm [19] environments, however, which necessitates a higher level of abstraction than can be provided by a single programming language or programming paradigm.

These conditions outlined above motivated the need for component-based software techniques [1]. A component is an encapsulated part of a software system that implements a specific service or set of services. A component model defines (1) the properties of components, such as the version of the component, its dependencies, language of implementation, (2) the set of interfaces that components use to interact amongst themselves and with other participants, and (3) the infrastructure needed to support the composition, run-time behavior, and set of external interfaces. Components and component models differ from objects and object models in the following ways:

- Component models define a higher level abstraction of the run-time execution environment than the operating systems level, which is often the case with object models. This helps in imposing policies on components and verifying them at run-time using the execution environment.
- Components are a higher-level abstraction than objects. This abstraction leads to differences in the following topics:
  - **External view** – Object models treat objects at the level of programming language abstractions of the same name and associated interfaces are defined are also defined on each such object. In contrast, component models only associate semantics of functionality with each component and allow flexibility in the packaging and implementation of component functionality.
  - **Encapsulation** – Component models provide "tighter encapsulation" of component functionality and hence limit the dependency on implementation level artifacts in component usage.
- Component programming is based on the Extension Interface design pattern [20], which defines a standard protocol for creating, composing, and evolving groups of interacting components. Unlike object-oriented designs (which reply on inheritance), component-based designs generally rely on aggregation for composition, which is more powerful since:
  - A component might not share any commonality with other components
  - A component might need to be integrated with existing components written in languages that do not support inheritance.
- Components are "context-insensitive", which allows clients to (1) interrogate a component to find out what interfaces it supports and (2) nagivate amongst these interfaces at run-time. This capability can be achieved through the separation of the functional properties from the behavioral properties and compositional aspects of components.

Aspect-oriented techniques. More recently, another programming paradigm has emerged that focuses on advanced techniques to separate common concerns in software systems. Known as *aspect-oriented programming* (AOP) [21], this paradigm provides a systematic, language-based approach for programming separate facets, such as memory management, logging, and synchronization, that cross-cut program

functionality. Research on AOP has also concentrated on enabling development of component software through specialized programming languages, such as Component Pascal [1], through introduction of higher-level constructs, such as Pluggable Composite Adapter [22], and language extensions, such as AspectJ [23] that extends Java to support the composition of separately developed software aspects.

2.2 Common Capabilities in Component Models

The most commonly used component models today include:

Microsoft's .NET , which allows developers to write application logic in different languages and generate components and assemblies targetted to a Common Language Runtime (CLR). CLR forms the foundation for the .NET Web services, which combine aspects of component-based development and Web technologies. Like earlier Microsoft component models (such as COM+ and ActiveX), .NET provides black-box functionality that can be described and reused without concern for how a service is implemented. In practice, however, .NET is only available on Windows platforms. Moreover, since .NET is targeted at desktop and enterprise applications, it is not suitable for DRE systems with stringent QoS requirements.

Sun's Enterprise Java Beans (EJB) technology allows developers to create n-tier distributed systems by linking a number of pre-built software services-called "beans" without having to write much code from scratch. Since EJB is built on top of Java technology, EJB service components can only be implemented using the Java language, which can be limiting for applications that are written in other languages. Another disadvantage of EJB stems from Java's inability to provide stringent real-time QoS guarantees, which makes it impractical for use in DRE systems.

OMG's CORBA Component Model (CCM) , which defines a superset of EJB capabilities that can be implemented using all the programming languages supported by CORBA. Since CORBA and CCM are also platform-independent, they can run atop most operating systems. It is possible to integrate CCM and EJB components seamlessly since they both use the Internet Inter-ORB Protocol (IIOP) as their underlying communication protocol. Since CORBA and its support for capabilities (such as asynchronous messaging, publisher/subscriber communication, fault tolerance, and real-time control of processor and networking resources) provides the middleware technology that is most well-suited for QoS-enabled DRE systems, our work (and Section 3 in this paper) focuses on CCM.

Although each of these component models differ from each other, there are key similarities, particularly in terms of their patterns [24, 25]. Below, we describe the most common capabilities that are shared among these component models.

Multiple views per component. Each component model specifies a collection of interfaces that a component can export to its clients. These interfaces vary in the capabilities that they offer to clients. For example, [1] refers to so-called *black interfaces*, *gray interfaces*, and *white interfaces*, with each type of interface providing introspection capabilities with increasingly powerful semantics, respectively.

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 together, and enforcing component policies (such as their life-cycle, security, and persistence state).

Component identity. Each component model has a means to identify its components uniquely. For example, .NET uses public key cryptography tokens to tag each component's interface to identify it uniquely across different software domains. EJB uses the Java Naming and Directory Service (JNDI), which encapsulates low-level naming services, such as LDAP, NIS, and DNS. EJB components are written using a hierarchical directory naming scheme typically associated with an organization's Internet domain. The CCM uses DCE "universally unique ids" (UUIDs) to identify component implementations. Section **??** explains other capabilities that CCM provides to identify components.

Based on an underlying object model. Each of today's popular component models are based on an underlying object model, as outlined below:

- EJB uses the Java Virtual Machine (JVM) as its underlying object model.
- .NET is based on the Common Language Runtime (CLR) and executes byte-code in Microsoft Intermediate Language (IL) [26].
- CCM is based on the CORBA object model.

The JVM and CLR are similar in that they provide a run-time environment that manages running code and simplifies software development via automatic memory management mechanisms, translating bytecode into an action or operating system call, a common deployment model, and a security system. The JVM has mostly been used for byte-code generation from Java. Likewise, in practice CLR is a run-time that works only under Windows.

The use of a virtual machine architecture is a source of non-determinism in DRE systems. In contrast, since CCM uses CORBA as its underlying object model, it need not use a virtual machine and hence is a more suitable platform for DRE systems with stringent QoS requirements.

3 Overview of the CORBA Component Model (CCM) and CIAO

The CORBA Component Model (CCM) is an OMG specification that standardizes the development of component-based applications. Since CCM uses CORBA as its underlying object model, developers are not tied to any particular language or platform for their component implementations. The CCM helps alleviate the problems with software composition by separating some concerns and thus reducing coupling. Sidebar 1 explains why the CIAO project is based on CCM rather than other popular component models, such as EJB or .NET.

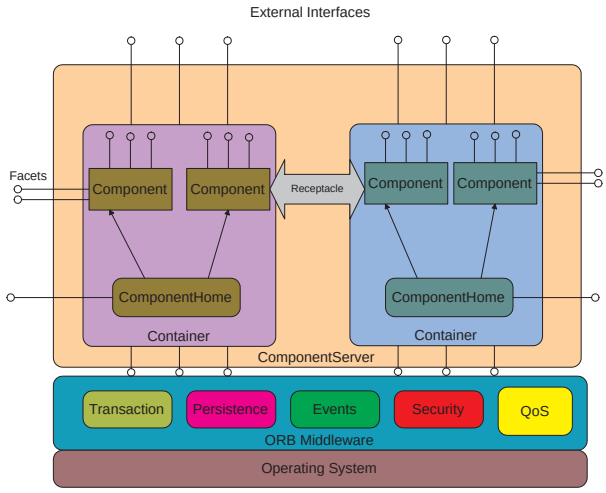

Figure 2: Key Elements in the CORBA Component Model

Figure 2 illustrates the following key elements of the CCM that we are implementing in CIAO:

- Component, the basic building block
- ComponentHome, the component type manager
- Container, the execution enivronment of a component
- ComponentServer, the container manager
- ORB Services, middleware services

The preceding discussion outlines the elements of the CCM and CIAO, but does not motivate what these elements do or more importantly *why* they are important. In the remainder of this section, we explain why these elements are needed in CCM by explaining the key software development challenges they address, which include:

1. Identifying and reusing commonality in software systems
2. Reducing coupling between components and underlying middleware
3. Specifying component interconnections
4. Using adaptive strategies for creating components
5. Configuring components
6. Resolving dependencies automatically
7. Maintaining component software

We also briefly outline how we have implemented these features in CIAO and reference the material in Section 4 that describes these key CIAO capabilities in more detail.

Sidebar 1: Motivation for Using the CCM

We base CIAO on the CCM since CORBA is the only COTS middleware that has made a substantial progress in satisfying the QoS requirements of DRE systems. For instance, the OMG has adopted the following DRE-related specifications in recent several years:

- **Minimum CORBA**, which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded systems.
- **Real-time CORBA**, which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end.
- **CORBA Messaging**, which exports additional QoS policies, such as asynchronous invocations, timeouts, request priorities, and queueing disciplines, to DRE applications.
- **Fault-tolerant CORBA**, which uses entity redundancy of objects to support replication, fault detection, and failure recovery.

These QoS specification and enforcement capabilities are essential to support DRE systems. Moreover, multiple interoperable and robust implementations of these CORBA capabilities and services are now available. Many of these CORBA implementations are freely-available in open-source format, which is conducive to research and whitebox evaluation. For these reasons, our work on CIAO focuses on the CCM as the basis for QoS-enabled component models to support DRE systems.

3.1 Identifying and Reusing Commonality in Software Systems

Context. A family of applications exhibiting commonality that can be refactored into reusable functional blocks.

Problem. When applications are implemented in a monolithic fashion, it is hard to identify and refactor common functionality. Choosing the right modulde boundaries is hard without appropriate abstractions for describing functionality. Lack of functional abstractions leads to unnecessary duplication across different modules and prevents effective reuse.

CCM Solution → Component. Define a component abstraction that serves as both the building block for the structure of software systems and as the candidate for demarcation of

modularity and functionality. A component is an encapsulated part of a software system that implements a specific service or set of services. A component has one or more interfaces that provide access to its services.

A CCM component is a meta-type that includes collection of entities, ranging from implementation(s) of application functionality in a particular programming language, a set of properties associated with each such implementation. A CCM component is both an extension and a specialization of the CORBA object meta-type that is defined by the original OMG CORBA specification. The capabilities of a CCM component are defined using extensions to the OMG Interface Definition Language (IDL).

Applying the solution in CIAO.

3.2 Reducing Coupling Between Components and Underlying Middleware

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

Problem. In earlier generation middleware that was based solely on object models, programmers had to explicitly handle the complexity of connecting to and configuring the policies of underlying middleware. For example, before the advent of CCM, CORBA developers had to explicitly bind to, and configure the policies of, middleware entities, such as event channels, transaction services, and security services. These manual programming activities resulted in the production of considerable, repetitive "glue-code" (which in some cases was larger than that required for the usage of the functionality). Likewise, these activities were error-prone since they required application developers to have expertise with many low-level details of the underlying middleware.

CCM Solution → Containers. Define a container abstraction that provides the context in which components run. A container acts as a bridge between the low-level middleware and a component by configuring the underlying middlware based on the policies defined in the component. A container also provides the execution environment for components, *e.g.*, it defines interception points where various run-time policies (such as security and transaction) can be imposed and validated. Although the capabilities provided via the containers are used by the components, they shield component developers from detailed knowledge of the underlying middleware.

An important consequence of decoupling components from containers is that the containers and the underlying middleware can transparently perform optimizations, such as component pooling, caching, and on-demand linking and load balancing of components. Likewise, the lifecycle of a component can be managed by its container, which has the advantage of having information from the perspective of not only a single component, but of all components residing within that container.

Applying the solution in CIAO.

3.3 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 gets all the functionality and the artifacts of that functionality or nothing. In complex software systems, however, a one-to-one association of component and component roles can result in an unwieldy proliferation of interfaces that must be managed by users explicitly.

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. The CCM specifies the following types of ports, which are a set of interfaces that are both external (to the user) and internal (to the underlying middleware):

- **Facets**, which are distinct named interfaces provided by the component. Facets enable a component to export a set of functional roles to its clients.
- **Receptacles** are interfaces used to specify relationships between components. This interface allows a component to accept references to other components, and invoke operations upon these references. Thus they enable a component to use the functionality provided by other components.
- **Event sources and sinks**, which define a standard interface for the Publisher/Subscriber architectural pattern [27]. Event sources/sinks are named connection points that send/receive specified types of events to/from one or more interested consumers/suppliers. These types of ports also hide the details of establishing and configuring event channels [4] needed to support The Publisher/Subscriber architecture.
- **Attributes**, which are named values exposed via accessor and mutator operations. Attributes can be used to expose the properties of a component that are exposed to tools, such as application deployment wizards that interact with the component to extract these properties and guide decisions made during installation of these components, based on the values of these properties. Attributes typically maintain state about the component and

can be modified by these external agents to trigger an action based on the value of the attributes.

Applying the solution in CIAO.

3.4 Using Adaptive Strategies for Creating Components

Context. Distributed software systems that consist of components with different lifetimes.

Problem. Locating and/or creating components are (potentially) expensive operations. Different component types might need creation strategies that differ from the other component types depending on the lifetime of instances of each type. For example, a component instance created as part of a database transaction might have a different lifetime than one which is controlling the trajectory of a missile.

Strategies used in the creation of both will involve a completely different set of tradeoffs. Requiring client applications to know how to locate and/or create components is tedious and introduces unnecessary dependencies between clients and the components they use. It also limits the flexibility of component creation strategies by tightly coupling component creation with component use.

CCM Solution → Component homes. Define a component home abstraction that is responsible for creating and subsequently locating certain types of components in a software system. Components reside in component homes, which embody the Factory [28] design pattern. Component homes shield clients from the details of creation strategies of components and subsequent queries to locate a component instance. This capability increases the flexibility of a system since any changes in how a component is created does not affect clients of the component.

Applying the solution in CIAO.

3.5 Configuring Components

Context. A distributed system where the same component needs to be configured differently, depending the context in which it is used.

Problem. As the number of component configuration parameters and options increase, it can become overwhemingly complex to configure applications consisting of a number of individual components. The problem stems not only from the number of alternative combinations, but also from the disparate interfaces for modifying these parameters. Object models have historically required application developers to manually write considerable application-specific "glue code" to interconnect and configure components. This coding process is tedious and error-prone since it exposes the component developers to low-level details of the underlying middleware.

CCM Solution → Assembly. Define an assembly abstraction to characterize meta-data. This meta-data describes a list of components present in the assembly. Each component's meta-data in turn describes the features available in it, or the features that it requires, *i.e.*, a dependency. After an assembly is defined the actual task of modifying the parameters need not involve manual writing of glue code. Instead, meta-programming techniques [12] can be applied to configure the component in a context dependent fashion since the properties of components and the code needed to configure these properties into the components are separated.

CCM assemblies are based on XML DTDs, which provide an implementation-independent mechanism for describing component properties. With the help of these XML templates, it is possible to generate default configurations for CCM components, which preserve the required QoS properties and establish the necessary configuration and interconnection among the components, as part of each assembly.

Applying the solution in CIAO.

3.6 Resolving Dependencies Automatically

Context. Run-time deployment of distributed systems built using components as the basic software building blocks.

Problem. Any non-trivial software system consists of a collection of components that have various dependencies, such as reliance on a particular group of components, order of component initialization, or domain-specific requirements (*e.g.*, required sensor rate in the avionics domain []). 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 chooses to instantiate these in any order. For example, it is imperative that the wheels of an aircraft open up before the aircraft tries to land.

CCM Solution → Deployment application. Define a deployment application that is responsible for managing the dependencies among a collection of interdependent components. By using meta-data which capture these dependency along information about the interconnections expressed via CCM ports, a deployment application can ensure that the component interconnections are established correctly and in the right order.

Applying the solution in CIAO.

3.7 Maintaining Component Software

Context. Software systems 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, it can be a maintenance problem. For example, the person-hours needed to maintain complex systems increases considerably when the number of invidual components in a system increases. This problem is aggravated by the fact that it is hard to determine the relationship between a component and its running context from just the presence of a component in a live system.

CCM Solution → Component servers. Define a Component Server abstraction that is 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 a singleton [28] that plays the role of a factory to create containers. A component server is the equivalent of a server process in the object models. Figure 3 shows the steps involved in deploying component software through Component Servers in a top-down fashion.

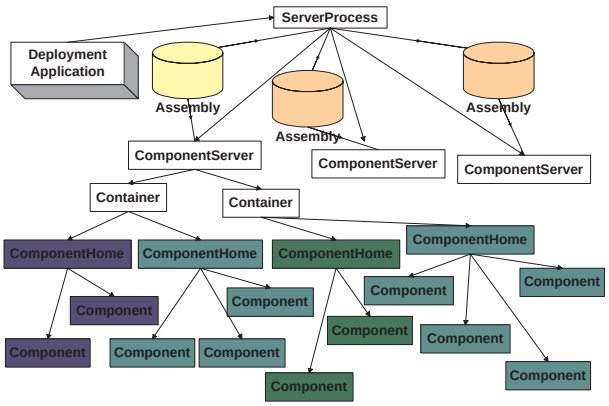

Figure 3: Component Deployment

Typically, a component server is assigned one high-level functionality within a complex system. During deployment, a single component server per assembly is created on each host, which reads the description of the meta-data from the assembly and is responsible for initiating the creation of the system hierarchy as well as teardown of the system hierarchy. Multiple containers can exist within a component server and the component server is responsible for managing the lifecyle of containers created within it.

Applying the solution in CIAO.

4

4.1 Reduction of Coupling Between Components and Underlying Middleware

As discussed in Section 3, CCM decouples components from their containers. This separation of concerns supports the following two different programming models:

- **Imperative programming**, which involves characterizing the state of a program entity and specifying a set of operations that modify the state. In CCM, imperative programming is commonly used for components, which are the basic entities in CCM that implement the core application functionality. Clients are exposed to and interact with interfaces offered by the components. These components are programmed imperatively, *i.e.*, the component developer must specify in detail the exact steps needed to provide the functionality offered by the component.
- **Declarative programming**, which involves specifying the result as either a function of the input or as a relation between the input and the input. There is no notion of state or direction of evolution of computation i.e it is bi-directional. In CCM, declarative programming can be used for the containers and the application servers. For example, the clients specify the set of actions needed from the Container, but it is the job of the container to compute and return the result.

CCM specifies the interaction of the above entities with the other auxillary tools that perform activities like packaging of components, visualization of components, deployment of components, and the validation of components.

4.2 Context-insensitive Specification of Component Properties

4.3 Validation of Component Configurations

4.4 Component Packaging and Deployment

5 Related Work

6 Concluding Remarks

CIAO addresses key challenges that arise when applying component models to DRE systems by separating the various aspects of DRE software systems and enabling application developers, system engineers, and end-users to select components that can then be composed to build complete systems.

Lessons learned. Java-based component models require using Java throughout the system, which might be infeasible either because major portions of the existing system is written

in another language or the real-time guarantees provided by Java based solutions are do not meet the requirements of DRE systems. .NET-based solutions are suffer the same problems because of their Windows-centric view of distributed objects, which precludes the possibility of developing a cross-platform solution, and requires software bridges between disparate systems, leading to increasing the complexity of building composable DRE systems.

Future work. The long 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.

References

[1] C. Szyperski, *Component Software — Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

[2] 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.

[3] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[4] 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.

[5] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[6] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[7] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[8] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[9] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[10] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA," *Concurrency and Computing: Practice and Experience*, vol. 13, no. 2, pp. 507–541, 2001.

[11] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.

[12] 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, Oct. 2001.

[13] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

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

[15] M. D. McIlroy, "Mass Produced Software Components," in *Proceedings of the NATO Software Engineering Conference*, Oct. 1968.

[16] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, Dec. 1972.

[17] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Mar. 1979.

[18] D. L. Parnas, P. Clements, and D. Weiss, "Enhancing Reusability with Information Hiding," *ITT Proceeding of the Workshop on Reusability in Programming*, 1983.

[19] Jim Coplien, *Multi-paradigm Design for C++*. Addison-Wesley, 1999.

[20] 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.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[22] M. Mezini, L. Seiter, and K. Lieberherr, "Component integration with pluggable composite adapters," 2000.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[24] Deepak Alur and John Crupi and Dan Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[25] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[26] A. D. Gordon and D. Syme, "Typing a multi-language intermediate code," *ACM SIGPLAN Notices*, vol. 36, no. 3, pp. 248–260, 2001.

[27] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.