Towards Composable Distributed Real-time and Embedded Software

Extended Abstract

Krishnakumar Balasubramanian, Nanbor Wang & Christopher 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 growing complexity of building and validating software is a 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 components, substantial time and effort are spent integrating components into applications. Integration challenges stem largely from the lack of higher level abstractions for composing complex applications. As a result, considerable application-specific "glue code" must be written, only to be rewritten from scratch when building subsequent DRE applications.

This paper provides four contributions to the study of composing reusable middleware from standard components in DRE applications: it (1) analyzes the limitations of current approaches in middleware composition, (2) discusses the minimum set of requirements required of reusable middleware components, (3) presents recurring patterns in the domain of software composition and provides empirical evaluation of these patterns as applied to TAO, our open-source Real-Time CORBA Object Request Broker (ORB), and (4) compares our approach to other research done in the area of software composition. Our results show that decoupling systemic QoS properties from functional properties enhances composition flexibility, increases reuse, and shields higher-level middleware and application developers from the complexities of the underlying middleware and OS platforms.

1 Introduction

1.1 Emerging Trends

With the proliferation of enterprise component technologies, such as 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 software maintainability over an application's lifecycle. Projects are also increasingly relying upon commercial off-the-shelf (COTS) components and frameworks as the basis for their distributed software infrastructure.

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 meta-data with component functionality. Component meta-data includes information such as the list of files used to implement a component, versioning information, a checksum to ensure component integrity, or information about the required privileges for a component to function. In DRE applications, a reusable component can be reapplied in a variety of contexts, each with differing QoS requirements, such as the deadlines for various time-critical functionality, concurrency levels, type of synchronization mechanisms, number of simultaneous transport connections allowed, and whether transport connections can be shared by multiple threads.

To reuse a component in more contexts than it was designed originally, the component's functionality needs to be separated from meta-data (such as its QoS properties) and described in a manner that can be understood by component users and associated tools. Composition problems can occur, however, if component meta-data is described at the same level of abstraction as the component functionality. In particular, tightly coupling meta-data and functionality can require applications to be written in the same language as its building block components, which may not be feasible if these entities have been developed independently at different points in time.

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, such as thread creation strategies, component lifetime (*e.g.*, persistent vs. transient), type of

invocation (*e.g.*, synchronous or asynchronous), and presence or absence of middleware services, such as event channels [4]. The context in which a component executes is generally provided by the external environment. Composition problems can arise when QoS requirements of components are specified with implicit assumptions on properties of external entities (*e.g*, the threading model required to run the component) that are external to a particular component of a software system. Unstated assumptions related to QoS properties of components make it hard to enforce QoS requirements effectively in open systems.

The need to validate component 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 should be performed against the component's specification to avoid problems such as (1) errors caused by misconfiguration, (2) unauthorized use of resources by components, and (3) lack of confidence that the QoS assurances provided by the middleware are sufficient from an application's perspective. Validation is required for each individual component, as well as the application and system levels.

The need to ensure that a complex software system can be deployed seamlessly. To reduce the complexity of installing and maintaining complex DRE 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 are implementing the Component-Integrated ACE ORB (CIAO), which extends the CORBA Component Model (CCM) [1], to address the challenges outlined in Section 1.2 as follows:

Reduced coupling by separating meta-data from functionality. CIAO provides a framework based on *eXtensible Markup Language* (XML) [5] mechanisms to define the grammar for describing component features. Our XML-based approach to describing component properties and systemic meta-data 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 meta-data 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 meta-data from functionality in CIAO.

Context-insensitive specification of QoS requirements. In CIAO, a component's dependencies are specified explicitly using meta-data 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 uses XML Document Type Definitions (DTDs) to identify critical QoS parameters of component-based DRE applications and to specify properties of components defined by 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 OMG Component Implementation Definition Language (CIDL) compiler (see Section 3.1) as part of the meta-data for every component. These attributes can be modified or extended by users. XML DTDs can be used to (re)validate meta-data attributes *before* components are deployed, thereby avoiding exceptions during runtime. 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. As shown in Figure 1, packaging involves grouping the implementation of component functionality (which is typically stored in a dynamic link library (DLL)) together with other meta-data that describes properties of this particular implementation. Packaged components are in "passive mode," *i.e.*, all their functionality is present, but they are inert object code. To carry out their functionality at run-time, components must transition to "active mode," where the inter-connection between components is established. Deployment mechanisms are responsible for

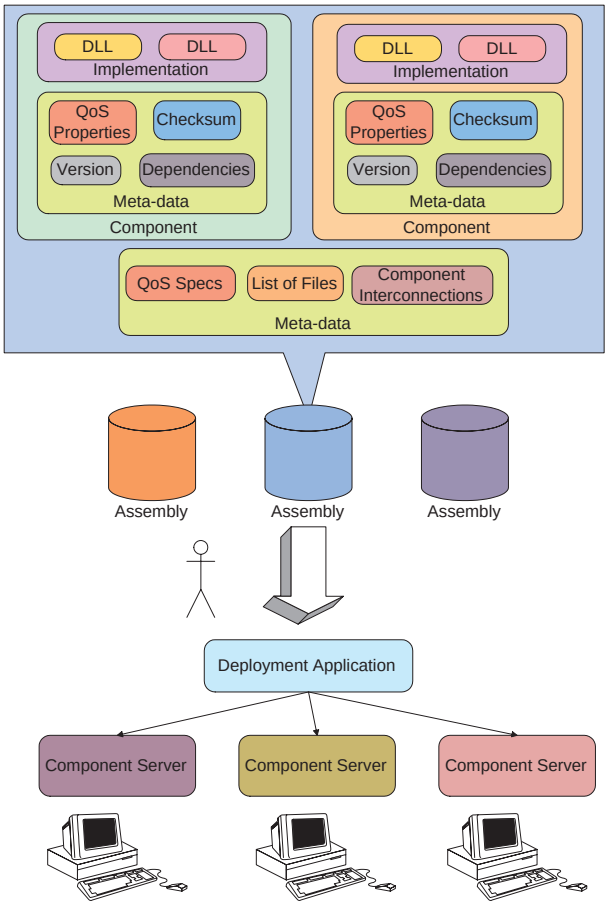

Figure 1: Component Packaging and Deployment

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 component models are as follows:

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. 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 together, and enforcing component policies, such as life-cycle, security, and persistency.

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 CCM provides to identify components.

Association with an object model. Today's component models are developed atop underlying object models that define the basic units of encapsulation and interoperability. The object models associated with component models include:

- EJB uses the Java programming language object model, exemplified by *java.lang.Object*, with the Java Virtual Machine (JVM) [6] providing run-time support.
- .NET is based on the Microsoft Intermediate Language (IL) [7], exemplified by *System.Object*, with the Common Language Runtime (CLR) [8] providing run-time support.
- CCM is based on the CORBA object model, exemplified by *CORBA::Object*, with a CORBA [9] ORB providing run-time support.

The JVM and CLR are similar in that they provide a runtime environment that manages running code and simplifies software development via automatic memory management mechanisms, translating bytecodes into actions or operating system calls, a common deployment model, and a security mechanism. Automatic memory management via garbage collection coupled with a virtual machine architecture is a source of non-determinism and impacts performance in DRE applications. In contrast, since CCM uses CORBA as its underlying object model, it need not use a virtual machine or garbage collection and hence is a more suitable platform for DRE applications with stringent QoS requirements.

3 Overview of the CCM and CIAO

3.1 Key Capabilities of the CCM

The CORBA Component Model (CCM) is an OMG specification that standardizes the development of component-based applications in CORBA. Since 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.

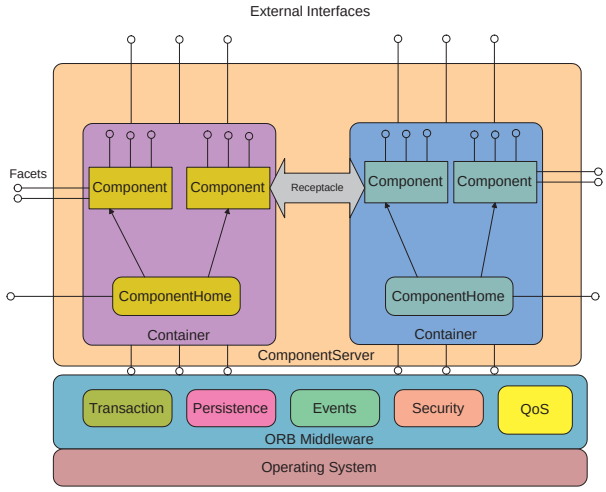

Figure 2: Key Elements in the CORBA Component Model

Key elements of the CCM include:

- **Component**, which is the basic building block used to encapsulate application functionality
- **ComponentHome**, which is 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 transaction, events, security and persistence.

Figure 2 illustrates some of the above described elements. The remainder of this section explains why these elements are needed in CCM by illustrating the key software development challenges they address, which include:

1. Identifying and reusing commonality in software applications
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. Evolving component software

3.1.1 Identifying and 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 the right 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. A CCM component has the following properties:

- It is an encapsulated part of a software system that implements a specific service or set of services.
- It has one or more interfaces that provide access to its services.
- It is a meta-type that includes collection of entities, which includes implementation(s) of application functionality in a particular programming language and a set of properties associated with each such implementation.
- It 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).

3.1.2 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 based on object models, programmers were responsible for connecting to and configuring the policies of the 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 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 have expertise in 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 an 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. Components can also use the capabilities provided via the containers to shield component developers from undue dependencies on 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. This design has the advantage of having information from the perspective of not only a single component, but of all components residing within that container.

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. The CCM specifies the following types of ports, which are a set of interfaces that are both external (to the clients) 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 different functional roles to its clients.
- **Receptacles**, which are interfaces used to specify relationships between components. Receptacles allow a component to accept references to other components and invoke operations upon these references. They therefore 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 [10]. 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.

3.1.4 Using Adaptive Strategies for Creating Components

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

Problem. Locating and/or creating components is a potentially expensive operation. Moreover, 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.

For example, different component types might need creation strategies that differ from the other component types depending on the lifetime of instances of each type. In particular, a component instance created as part of a database transaction might have a different lifetime than one that is controlling the trajectory of a missile. Strategies used in the creation of both will involve a different set of tradeoffs, which ought to be handled by the middleware rather than each application.

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 [11] 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 changes in how a component are created need not affect component clients.

3.1.5 Configuring Components

Context. A distributed 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 manually write large amounts of application-specific "glue code" to interconnect and configure components. In addition to being tedious and error-prone, this coding process exposes the component developers to low-level details of the underlying middleware.

CCM Solution → Assembly. Define an *assembly* abstraction to group components and characterize their meta-data that describes the components present in the assembly. Each component's meta-data 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 actual task of modifying the parameters need not involve manual writing of glue code. Instead, meta-programming techniques [12] 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 into the components.

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 assembly configurations can preserve the required QoS properties [13] and establish the necessary configuration and interconnection among the components.

3.1.6 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 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 [14]). 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 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 a collection of interdependent components. A deployment application can ensure that component interconnections are established correctly and in the right order by using meta-data that capture these dependencies, along with information about the interconnections expressed via CCM ports.

3.1.7 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 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 [11] 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.

A component server is typically assigned one high-level functionality within a complex system. For example, a wing sensor of an aircraft might be configured as a component

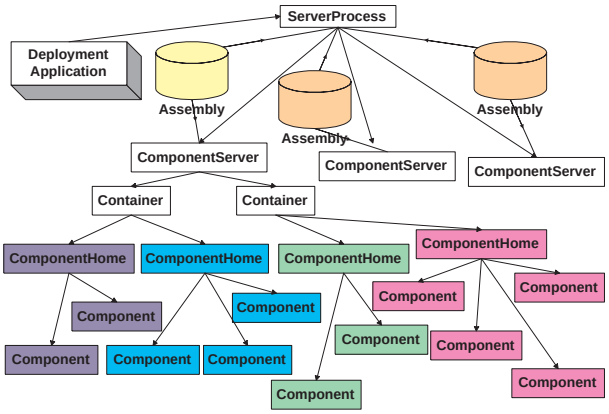

Figure 3: Component Deployment in CCM

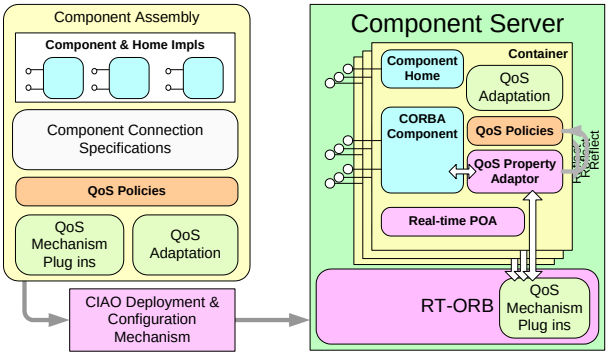

Figure 4: **Key Elements in CIAO**

server consisting of a number of components that work together to control the wings of the aircraft. During deployment, a single component server per assembly is created on each host. The component server reads the description of the metadata from the assembly and is responsible for initiating the construction and teardown of the component/container hierarchy. Multiple containers can exist within a component server – the component server is responsible for managing the lifecyle of containers created within it.

3.2 Key Capabilities of CIAO

The *Component-Integrated ACE ORB* (CIAO) developed at Washington University, St. Louis is an extension to CCM. CIAO is designed to bring the component-oriented development paradigm to DRE application developers by abstracting DRE-critical systemic aspects, such as QoS requirements and real-time policies, as installable/configurable units. Promoting these DRE-critical aspects as first-class meta-data 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 aspect from a higher level of abstraction, CIAO also makes configuring and managing these aspects easier [15].

The CIAO implementation is based on TAO, which is our open-source, high-performance, highly configurable Real-time CORBA ORB that implements key patterns [16] 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 4 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 adaptations. CIAO also supports installation of meta-programming hooks, such as Portable Interceptor and smart proxies [12], 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 functionality from component implementation and assemble a DRE application by composing and connecting application functional components, QoS specifications, and reusable QoS adaptation behaviors together. Section 4 describes how CIAO addresses the challenges in 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 CCM provides capabilities that help them develop composable middleware and applications. The capabilities offered by 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 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 meta-data 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 following challenges that were first introduced in Section 1:

- Reducing coupling by separating meta-data from functionality
- Context-insensitive specification of QoS properties
- Validation of component configurations
- Component packaging and deployment

4.1 Reducing Coupling by Separating Meta-data from Functionality

Context. Developing DRE middleware that have considerable amount of systemic meta-data.

Problem. DRE middleware have traditionally contained a considerable amount of meta-data, *i.e.*, information that describes systemic characteristics. As identified in Section 1, these meta-data do not implement application functionality per se. They are nevertheless important for the proper functioning of the application. There are two common problems with meta-data:

- Tangling the meta-data with the implementation of the functionality leads to an overly strong coupling between the two, which can impede application evolution.
- Specifying meta-data in an *ad hoc* manner prevents interaction with portions of the application developed by other suppliers who use non-compatible meta-data specification mechanisms.

Together, these two factors present DRE integrators with a challenge whereby individual components may function satisfactorily, but the composition of these components into higher-level applications may not meet various 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 to describe meta-data. Describe component meta-data separately from the implementation of the component functionality. Designing a language to define the meta-data is hard since it incurs the challenges associated with designing programming languages. For example, the language used to define meta-data should be extensible to allow the specification of meta-data that is open-ended and subject to change. Designing a language for extensibility [17] involves tradeoffs (such as level of expressibility, ease of adding new features, maintaining backward compatibility, and preventing alienation of existing users) that must be handled carefully.

XML provides a basis for defining a meta-language, *i.e.*, a language that can be used to describe another language. In this case, the XML-based meta-language is used to describe DRE application meta-data, while minimizing the effort required to design a full-fledged language. Using XML to specify component meta-data enables designers and integrators of DRE applications to separate the "meta-data" 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) [18] to assist in handling XML used for the specification of meta-data. There are two types of XML APIs:

- **Tree-based APIs**, which map an XML document into an internal tree structure, then allow an application to navigate that tree. The Document Object Model (DOM) working group at the World-Wide Web Consortium (W3C) maintains a recommended tree-based API for XML and HTML documents, and there are many such APIs from other sources, such as DOM model APIs for Mathematical Markup Language [19], Scalable Vector Graphics [20], and Synchronized Multimedia Integration Language [21].
- **Event-based APIs**, which reports parsing events (such as the start and end of elements) directly to an application via callbacks and does not usually build an internal tree. An application implements handlers to deal with the different events, much like handling events in a graphical user interface. SAX is the best known example of such an API.

Figure 5 shows the how ACEXML can be used to parse XML documents. During deployment (see Section 4.4) ACEXML reads the meta-data from an assembly and uses it to validate (see Section 4.3) the contents of the assembly. Since DRE applications often have stringent footprint requirements, they cannot afford the overhead involved with building the entire tree structure in memory, as is the case with DOM based APIs for parsing XML. This problem assumes greater significance if the amount of meta-data specified becomes large and unwieldy, such as when meta-data is auto-generated by modeling tools [22] and component-aware IDL compilers. To avoid

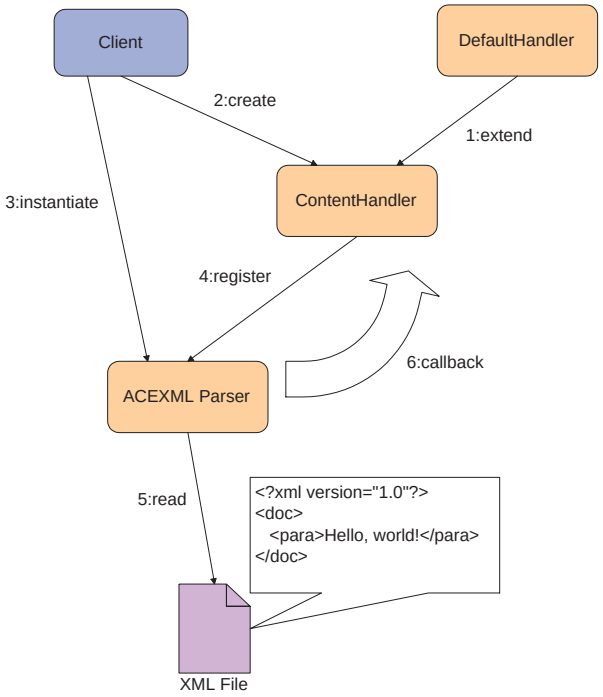

Figure 5: **Event Handling in ACEXML**

this overhead, ACEXML is based on SAX which eliminates the need to build the whole tree structure 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 the following:

- The implementation of QoS enabling mechanisms, such as scheduling algorithms, thread pools, connection pooling and caching and event demultiplexing provided by the underlying middleware
- 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 be elevated to the role of a first-class citizen in the middleware typesystem and 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* (.cpf), which is described in Sidebar 1 on page 10. This file 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. An example component property file (.cpf) is shown below:

```
<properties>
  <simple name=bufSize type="long">
     <description>Size of CDR input buffer
        </description>
     <value>4096</value>
     <defaultvalue>256</defaultvalue>
  </simple>
  <simple name=bandwidth type="long">
     <description>Network bandwidth to reserve
         </description>
     <!-- In Mbps -->
     <value>15</value>
  </simple>
  <sequence name="Latency" type="sequence<long>">
    <!-- Component's min/avg/max latency in us -->
     <simple type="long"><value>5</value></simple>
     <simple type="long"><value>10</value></simple>
     <simple type="long"><value>15</value></simple>
  </sequence>
  <struct name="PathMonitor" type="sensorStruct">
     <description>Flightpath recalculation
        </description>
     <simple name="hour" type="short">
        <value> 0 </value>
     </simple>
```

Sidebar 1: Separating Configuration Concerns in CCM

Configuration of components in CCM is performed at different levels of abstraction and involves different tradeoffs. CCM uses XML-based descriptors to configure components. Each descriptor exposes different aspects of a component-based system. In this sidebar, we describe the different types of descriptors in CCM and explain how they help separate the concerns of component configuration:

- **CORBA assembly descriptor (.cad)**, which is a meta-information file with details of an assembly archive. It includes a list of the set of components that form the assembly, component descriptors and implementations, the inter-connections between these components, and component homes. It is used by the deployment tool to configure the component inter-connections, component homes, etc.
- **CORBA component descriptor (.ccd)**, which is a meta-information file that describes the features of a single component. It includes information, such as the provided, required and supported interfaces; ports information; and QoS policies (*e.g.*, threading, transaction and security and access mode).
- **CORBA software descriptor (.csd)**, which is a compressed file that contains one or more implementations of a component or an interface. It can be used to deploy an individual component or an interface that is not part of an assembly.
- **Component property file (.cpf)**, which is used to describe QoS properties of an assembly as a whole or an individual component or a specific implementation of a component. Properties can optionally be overridden if specified at multiple levels.

By using descriptors at multiple levels of granularity, CCM separates out the concerns of component software and enables the weaving of complete applications from these individual aspects.

```
    <simple name="minute" type="short">
       <value> 0 </value>
    </simple>
    <simple name="second" type="short">
       <value> 0 </value>
    </simple>
    <simple name="millisecond" type="short">
       <value> 5 </value>
    </simple>
  </struct>
</properties>
```

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

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 [23]. 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

- Checking a large number of individual components' QoS properties to ensure that the component satisfies the requirements and
- Ensuring that the overall system composed of these individual components satisfies the QoS guarantees.

Solution → Validate component configurations. Validate component configurations by checking the meta-data 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 [24] is vast and merits a detailed discussion [25] of its own.

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

Applying the solution in CIAO. Component configurations in CIAO are specified through a set of descriptors, as outlined in the preceding paragraph. CIAO's implementation of CCM CIDL compiler generates a default configuration for every component and hence a default descriptor. In many real-life use-cases of components, 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 that can be used to validate any XML DTD.

4.4 Component Packaging and Deployment

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

Problem. DRE applications are composed of many components. In complex DRE applications, there may be hundreds or thousands of these components. As the number of components increase, it is hard to manage the application at the granularity of individual components. Specifying the provisioning at the level of individual components might also be insufficient. Some of the QoS properties cross-cut component boundaries, so they should 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:

- **Component software package**, which contains one or more implementations of a component. Each package needs to have an associated descriptor, as explained in Section 4.3, which assists in composition and in provisioning of QoS properties during deployment. A component software package may be installed on a component server and servers as the vehicle for deploying a single component implementation.
- **Component assembly package**, which contains a set of inter-dependent components and information which describes the dependencies between these components. A component assembly package serves as the vehicle for deploying a set of interrelated components. The definition of a component assembly is recursive and a component assembly itself can be composed further to yield another assembly.

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* (.csd) file, which is described in Sidebar 1 on page 10. This file 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. An example CORBA Software Descriptor (.csd) file is shown below:

```
<softpkg name="Sensor" version="1,0,1,0">
   <pkgtype>CORBA Component</pkgtype>
   <title>Sensor</title>
   <author>
      <company>Qosketeers Inc.</company>
      <webpage href="http://www.qosket.com"/>
   </author>
   <description>
      Yet another QoS package example
   </description>
   <license
      href="http://www.qosket.com/license.html" />
   <propertyfile>
      <fileinarchive name="Sensor.cpf"/>
   </propertyfile>
   <implementation
      id="DCE:700dc518-0110-11ce-ac8f-0800090b5d3e">
      <os name="WinNT" version="4,0,0,0" />
      <os name="Win95" />
      <processor name="x86" />
      <compiler name="Microsoft Visual C++" />
      <programminglanguage name="C++" />
      <dependency type="CORBA 3.0 ORB">
         <name>CIAO</name>
      </dependency>
      <descriptor type="CORBA Component">
         <fileinarchive>
            QoScontainer.ccd
         </fileinarchive>
         <fileinarchive name="QoScontainer.ccd" />
      </descriptor>
      <code type="DLL">
         <fileinarchive name="sensor.dll"/>
         <entrypoint>createSensor</entrypoint>
      </code>
      <dependency type="DLL">
         <localfile name="Monitor.dll"/>
      </dependency>
    </implementation>
    <implementation
       id="DCE:297f3e18-0110-11ce-ac8f-08074982ad3e"
       variation="RemoteHome">
      <os name="Solaris" version="5,5,0,0" />
      <processor name="sparc" />
      <!-- . . . -->
    </implementation>
    <implementation>
       <!-- another implementation -->
```

```
    </implementation>
</softpkg>
```

Each type of component within an implementation is described by a *CORBA component descriptor* (.ccd) file, which is discussed in Sidebar 1 on page 10. This file captures the structure of a component, with respect to its supported interfaces, inherited components, and ports. CIAO uses component descriptor files to facilitate inter-connections between components. An example CORBA component descriptor (.ccd) is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE corbacomponent
          SYSTEM "corbacomponent.dtd">
<corbacomponent>
   <corbaversion> 3.0 </corbaversion>
   <componentrepid
      repid="IDL:FlightPath:1.0" />
   <homerepid
      repid="IDL:FlightPathHome:1.0" />
   <componentkind>
      <entity>
         <servant lifetime="process" />
      </entity>
   </componentkind>
   <security rightsfamily="corba" />
   <threading policy="multithread" />
   <configurationcomplete set="true" />
   <homefeatures
       name="FlightPathHome"
       repid="IDL:FlightPathHome:1.0">
      <operationpolicies>
         <operation name="*">
            <transaction use="never" />
         </operation>
      </operationpolicies>
   </homefeatures>
   <componentfeatures
        name="FlightPath"
        repid="IDL:FlightPath:1.0">
      <inheritscomponent
         repid="IDL:QoSket/Path:1.0" />
      <ports>
         <provides
            providesname="calculate_freq"
            repid="IDL:Frequency:1.0"
            facettag="1"/>
         <provides
            providesname="admin"
            repid="IDL:QoSket/Admin:1.0"
            facettag="3" />
     </ports>
   </componentfeatures>
   <interface name="FlightPath"
              repid="IDL:FlightPath:1.0">
      <inheritsinterface
            repid="IDL:WingMonitor:1.0" />
   </interface>
</corbacomponent>
```

A *component assembly descriptor* (.cad) file describes which components make up the assembly, how those components are partitioned, and how they are connected to each other. During deployment, the CIAO deployment mechanism consults the component assembly descriptor file to bootstrap the deployment. An example component assembly descriptor(.cad) is shown below:

```
<!DOCTYPE componentassembly
          SYSTEM "componentassembly.dtd">
<componentassembly id="ZZZ123">
   <description>Example assembly</description>
   <componentfiles>
      <componentfile id="A">
         <fileinarchive name="ca.ccd"/>
      </componentfile>
     <componentfile id="B">
        <fileinarchive name="cb.ccd"/>
     </componentfile>
     <componentfile id="C">
        <fileinarchive name="cc.ccd">
           <link
            href="ftp://www.QoS.com/cc.aar"/>
        </fileinarchive>
     </componentfile>
     <componentfile id="D">
        <fileinarchive name="cd.ccd"/>
     </componentfile>
     <componentfile id="E">
        <fileinarchive name="ce.ccd"/>
     </componentfile>
     <componentfile id="F">
        <fileinarchive name="cf.ccd"/>
     </componentfile>
   </componentfiles>
   <connections>
      <connectinterface>
         <usesport>
            <usesidentifier>abc</usesidentifier>
            <componentinstantiationref
                idref="Aa"/>
         </usesport>
         <providesport>
            <providesidentifier>abc
                 </providesidentifier>
            <componentinstantiationref
                idref="Bb"/>
         </providesport>
      </connectinterface>
      <connectevent>
         <consumesport>
            <consumesidentifier>pqr
                 </consumesidentifier>
               <componentinstantiationref
                   idref="Aaa"/>
         </consumesport>
         <emitsport>
            <emitsidentifier>mno
                 </emitsidentifier>
               <componentinstantiationref
                    idref="Ee"/>
         </emitsport>
      </connectevent>
   </connections>
</componentassembly>
```

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 (which the daemon can determine by comparing against the UUID of a component implementation), the daemon will ensure it is not installed again.

5 Concluding Remarks

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

[6] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Massachusetts: Addison-Wesley, 1997.

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

[8] E. Meijer and J. Gough, *Technical Overview of the Common Language Runtime*. Microsoft, 2000.

[9] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 ed., June 2002.

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

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

[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] Nanbor Wang and Douglas C. Schmidt and Aniruddha Gokhale and Christopher D. Gill and Balachandran Natarajan and Craig Rodrigues and Joseph P. Loyall and Richard E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *Submitted to the Journal of Microprocessors and Microsystems*, vol. 26, jan 2003.

[14] 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.

[15] 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.

[16] 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.

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

[18] SAX Project, "Simple API for XML." www.saxproject.org, 2001.

[19] R. Ausbrooks, S. Buswell, S. Dalmas, S. Devitt, A. Diaz, R. Hunter, B. Smith, N. Soiffer, R. Sutor, and S. Watt, "Mathematical Markup Language (MathML) Version 2.0." W3C Recommendation, 2001.

[20] J. Bowler and SVG-Working-group, "Scalable Vector Graphics (SVG) 1.0 Specification." W3C Recommendation, 2001.

[21] P. L. Hgaret and P. Schmitz, "Synchronized Multimedia Integration Language Document Object Model." W3C Recommendation, 2000.

[22] D. C. Schmidt, A. Gokhale, and C. D. Gill, "Applying Model-Integrated Computing and DRE Middleware to High Performance Embedded Computing Applications," in *The 6th Annual Workshop on High Performance Embedded Computing*, (Boston, MA), MIT, Sept. 2002.

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

[24] 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*, vol. 25, (White Plains, NY), pp. 234–245, June 1990.

[25] 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.