QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems

Gan Deng and Douglas C. Schmidt
EECS Department
Vanderbilt U., Nashville, TN
{gan.deng, d.schmidt}@vanderbilt.edu

Christopher D. Gill
CSE Department
Washington U., St. Louis, MO
cdgill@cse.wustl.edu

Nanbor Wang
Tech-X Corp.
Boulder, CO
nanbor@txcorp.com

September 13, 2006

1 Introduction

Component middleware technologies are playing an increasingly important role in the development of distributed real-time and embedded (DRE) systems, in a variety of application domains ranging from military shipboard computing [1] to commercial inventory tracking [2]. The challenges of designing, implementing, and evaluating component middleware that can meet the needs of such diverse DRE systems have motivated several important advances in the state-of-the-art. This chapter summarizes those advances and uses examples from several application domains to show how these resulting technologies can be applied to meet the needs of DRE systems.

Section 2 surveys R&D challenges for DRE systems. Section 3 describes other middleware paradigms for DRE systems to motivate our work on quality of service (QoS)-enabled component middleware for DRE systems. Section 4 describes how we developed the *Component-Integrated ACE ORB* (CIAO) and the *Deployment And Configuration Engine* (DAnCE) QoS-enabled component middleware to address key challenges for DRE systems described in Section 2. Section 5 describes three application domains in which CIAO has been applied: avionics mission computing [3], shipboard computing [1], and inventory tracking [2]. Section 6 surveys related work on middleware for DRE systems, and Section 7 presents concluding remarks.

2 R&D Challenges for DRE Systems

Some of the most challenging R&D problems are those associated with producing software for DRE systems where computer processors control physical devices for sensing and actuation. Examples of such systems include avionics mission computing systems which help pilots plan and execute airborne missions, shipboard computing systems which help crews maintain situational awareness of the ship and its surroundings, and inventory tracking system that help to maintain an up-to-date picture of the location and status of parts and products needed by complex and dynamic supply chains. Despite advances in standards-based commercial-off-the-shelf (COTS) middleware technologies, several challenges must be addressed before COTS software can be used to build mission-critical DRE systems effectively and productively. In particular, as DRE systems have increased in scale and complexity over the past decade, a tension has arisen between stringent performance requirements and the ease with which systems can be developed, deployed, and configured to meet those requirements.

DRE systems require different forms of configuration–both at design-time and at run-time–to allow customization of reusable components to meet QoS requirements for applications being developed. In addition to being configured individually, components must be assembled to form a complete application and deployed across a set of endsystems upon which the applications will run. Characteristics of the components, their interdependencies when assembled, the endsystems onto which components are deployed, and the networks over which they communicate can vary *statically* (*e.g.*, due to different hardware/software platforms used in a product-line architecture) and *dynamically* (*e.g.*, due to damage, changes in mission modes of the system, or due to differences in the real vs. expected behavior of applications during actual operation). When these variabilities are combined with the complex requirements of many DRE systems and the dynamic operating environments in which they operate, it becomes tedious and error-prone to configure operational characteristics of these systems, particularly when components are deployed manually or using middleware technologies that do not provide appropriate configuration support.

Developers of complex DRE systems therefore need middleware technologies that offer (1) explicit configurability of policies and mechanisms for QoS aspects such as priorities, rates of invocation, and other real-time properties, so that developers can meet the stringent requirements of modern DRE systems, (2)

a component model that explicitly separates QoS aspects from application functionality so developers can untangle code that manages QoS and functional aspects, resulting in systems that are less brittle and costly to develop, maintain, and extend and (3) configuration capabilities that can customize functional and QoS aspects of each DRE system, flexibly, efficiently at different stages of the system lifecycle.

3 Comparison of Middleware Paradigms

This section examines previous approaches to middleware for DRE systems and explains why only QoS-enabled component middleware addresses all the challenges described in Section 2.

Conventional DOC middleware. Standards-based COTS middleware technologies for distributed object computing (DOC), such as The Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [4] and Sun's Java RMI [5], shield application developers from low-level platform details, provide standard higher-level interfaces to manage system resources, and help to amortize system development costs through reusable software frameworks. These technologies significantly reduce the complexity of writing client programs by providing an object-oriented programming model that decouples application-level code from the system-level code that handles sockets, threads, and other network programming mechanisms. Conventional DOC middleware standards, however, have the following limitations for DRE systems [6]:

- **Only functional and not QoS concerns are addressed.** DOC middleware addresses *functional* aspects, such as how to define and integrate object interfaces and implementations, but does not address crucial QoS aspects, such as how to define and enforce deadlines for method execution. The code that manages these QoS aspects often becomes entangled with application code, making DRE systems brittle and hard to evolve.

- **Lack of functional boundaries.** Application developers must explicitly program the connections among interdependent services and object interfaces, which can make it difficult to reuse objects developed for one application in a different context.

- **Lack of generic server standards.** Server implementations are *ad hoc* and often overly coupled with the objects they support, which further reduces the reusability and flexibility of applications and their components.

- **Lack of deployment and configuration standards.** The absence of a standard way to distribute and start up implementations remotely results in applications that are hard to maintain and even harder to reuse.

QoS-enabled DOC middleware. New middleware standards, such as Real-time CORBA (RTCORBA) [7] and the Real-Time Specification for Java (RTSJ) [8], have emerged to address the limitations of conventional DOC middleware standards described above. These technologies support explicit configuration of QoS aspects, such as the priorities of threads invoking object methods. They do not support generic server environments or component deployment and configuration, however, which can lead to tangling of application logic with system concerns, similar to the problems seen with conventional DOC middleware.

For example, although Real-time CORBA provides mechanisms to configure and control resource allocations of the underlying endsystem to meet real-time requirements, it lacks the flexible higher level abstractions that component middleware provides to separate real-time policy configurations from application functionality. Manually integrating real-time aspects within Real-time CORBA application code is unduly time consuming, tedious, and error-prone [3]. It is hard, therefore, for developers to configure, validate, modify, and evolve complex DRE systems consistently using conventional QoS-enabled DOC middleware.

Conventional component middleware. Component middleware technologies, such as the CORBA Component Model (CCM) [9] and Enterprise JavaBeans [10], have evolved to address the limitations of conventional DOC middleware in addressing the functional concerns of DRE systems, by (1) separating application components so that they interact with each other only through well-defined interfaces, (2) defining standard mechanisms for configuring and executing components within generic containers and component servers that divide system development concerns into separate aspects, such as implementing application functionality vs. configuring resource management policies, and (3) providing tools for deploying and configuring assemblies of components.

In CCM, for example, components interact with other components through a limited set of well defined interface called *ports*. CCM ports include (1) *facets*, which provide named interfaces that service method invocations from other components, (2) *receptacles*, which provide named connection points to facets provided by other components, (3) *event sources* which indicate a willingness to send events to other components, and (4) *event sinks* which indicate a willingness to receive events from other components. Components can also have *attributes* that specify named parameters. which are configurable at various points in an application's lifecycle.

Components are implementation entities that can be installed and instantiated independently in standard component server runtime environments stipulated by the CCM specification. For each type of component, a *home* interface provides lifecycle management services. A *container* provides the server runtime environment for component implementations, and handles common concerns such as persistence, event notification, transaction, security, and load balancing.

These component middleware features ease the burden of developing, deploying, and maintaining complex large-scale systems by freeing developers from connecting and configuring numerous distributed subsystems manually. Conventional component middleware, however, is designed more for the needs of enterprise business systems, rather than for the more complex QoS provisioning needs of DRE systems. Developers are therefore often forced to configure and control these mechanisms imperatively within their component implementations.

While it is possible for component developers to embed QoS provisioning code directly within a component implementation, doing so often prematurely commits each implementation to a specific QoS provisioning strategy, making component implementations harder to reuse. Moreover, many QoS capabilities, such as end-to-end provisioning of shared resources and configuring connections between components, cannot be implemented solely within a component because the concerns they address span multiple components.

QoS-enabled component middleware. Due to the limitations described above, it is necessary to extend standard middleware specifications so that they provide better abstractions for controlling and managing both functional and QoS aspects of DRE systems. In particular, what is needed is *QoS-enabled component middleware* that preserves existing support for heterogeneity in standard component middleware, yet

also provides multiple dimensions of QoS provisioning and enforcement and offers alternative configuration strategies to meet system-specific QoS requirements across the diverse operating environments of DRE systems.

The remainder of this chapter describes how we have extended the standard Lightweight CCM specification to support QoS-enabled component middleware that can more effectively compose real-time behaviors into DRE systems and help make it easier to develop, verify, and evolve applications in these systems. Specifically, we cover:

- How real-time CORBA policies can be composed into Lightweight CCM applications during their development lifecycle, using an XML-based metadata format for describing how real-time policies can be coupled with existing CCM metadata that define application assemblies.
- How the Deployment And Configuration Engine (DAnCE) framework we developed can translate an XML-metadata specification into the deployment and configuration actions needed by an application, and how static configuration optimizations we developed for DAnCE optimize that capability to support applications with constraints on configuration times or on real-time operating system features.
- How these added capabilities can be used to develop DRE systems via examples from several application domains.
- How extending various CCM metadata to document behavioral characteristics of components and applications can help streamline the optimization of DRE systems.

As we describe in Section 4, the vehicles for our R&D activities on QoS-enabled component middleware are (1) the *Component-Integrated ACE ORB* (CIAO), the *Deployment And Configuration Engine* (DAnCE), which are based on the OMG's Lightweight CORBA Component Model (LCCM) specification [11], and (2) the Component Synthesis of Model Integrated Computing (CoSMIC) [12], which is a Model-Driven Engineering (MDE) [13] tool chain.

4 Achieving QoS-enabled Component Middleware: CIAO, DAnCE, CoSMIC

This section presents the design of CIAO and DAnCE, which are open-source QoS-enabled component middleware developed at Washington University and Vanderbilt University atop *The ACE ORB* (TAO) [14]. TAO in turn is open-source Real-time CORBA DOC middleware that implements key patterns [15] for DRE systems. CIAO and DAnCE enhance TAO to simplify the development of applications in DRE systems by enabling developers to provision QoS policies declaratively end-to-end when deploying and configuring DRE systems using CCM components. We also describe how the CoSMIC MDE tools support the deployment, configuration, and validation of component-based DRE systems developed using CIAO and DAnCE.

4.1 Integration of Real-time CORBA and Lightweight CCM Capabilities in CIAO

CIAO supports Lightweight CCM mechanisms that enable the specification, validation, packaging, configuration, and deployment of component assemblies and integrates these mechanisms with TAO's Real-time CORBA features, such as thread-pools, lanes, and client-propagated and server-declared policies. The architecture of CIAO is shown in Figure 1.1. CIAO extends the notion of Lightweight CCM component assembly

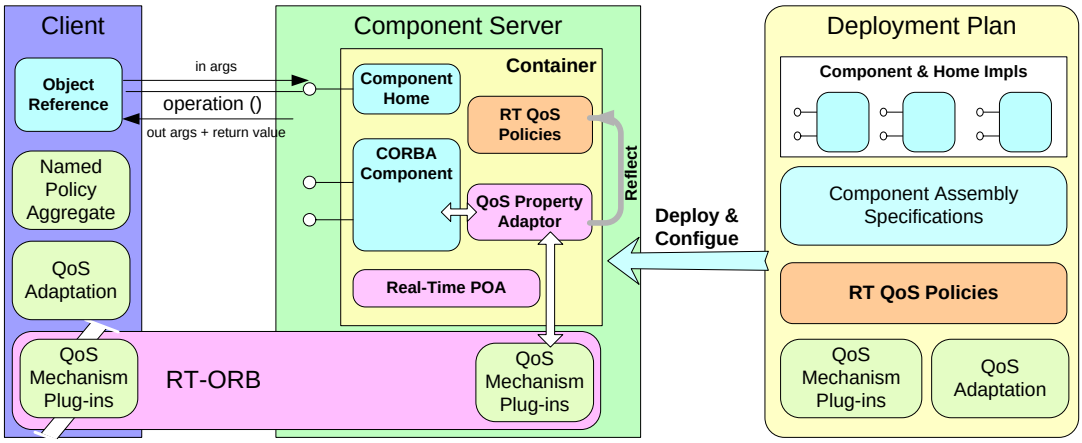

Figure 1.1: The Architecture of CIAO

to include both client-side and server-side QoS provisioning and enables the declarative configuration of QoS features of endsystem ORBs, such as prioritized thread pools [16], via XML metadata. It also allows QoS

provisioning at the component-connection level, *e.g.*, to configure thread pools with priority lanes.

To support the composition of real-time behaviors, CIAO allows developers of DRE systems to specify the desired *real-time QoS policies* and associate them with components or component assemblies. For example, application components can be configured to support different priorities and rates of invocation of methods among components. These configurations are declaratively specified via metadata, and once they are deployed, the properties of CIAO's real-time component server and container runtime environment will be automatically initialized based on the definition of QoS mechanism plug-ins. In particular, this metadata can configure CIAO's QoS-aware containers, which provide common interfaces for managing components' QoS policies and for interacting with the mechanisms that enforce those QoS policies.

To support the configuration of QoS aspects in its component servers and containers, CIAO defines a new file format – known as the *CIAO server resource descriptor* (CSR) – and adds it to the set of XML descriptors that can can be used to configure an application assembly. A CSR file defines *policy sets* that specify policies for setting key QoS behaviors such as real-time thread priorities, and configure resources to enforce them. The resources and policies defined in CIAO's CSR files can be specified for individual component instances. System developers then can use deployment and configuration tools, such as the DAnCE middleware and CoSMIC MDE tools described in Section 4.1, to deploy the resulting application assembly onto platforms that support the specified real-time requirements. The middleware and MDE tools can also configure the component run-time infrastructure that will enforce these requirements.

4.2 The Design of DAnCE

DAnCE is a middleware framework we developed for CIAO based on the OMG's Deployment and Configuration (D&C) specification [17], which is part of the Lightweight CCM specification [11]. This specification standardizes many deployment and configuration aspects of component-based distributed applications, including component configuration, assembly, and packaging; package configuration and deployment; and resource configuration and management. These aspects are handled via a *data model* and a *runtime model*. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describes component assemblies and their deployment and configuration attributes, such as resource require-

ments. The runtime model defines a set of services/managers that process the metadata described in the data model to deploy, execute, and control application components in a distributed manner.

DAnCE implements a data model that describes (1) the DRE system component instances to deploy, (2) how the components are connected to form component assemblies, (3) how these components and component assemblies should be initialized, and (4) how middleware services are configured for the component assemblies. In addition, DAnCE implements a spec-compliant runtime model as a set of runtime entities.

The architecture of DAnCE is shown in Figure 1.2. The various entities of DAnCE are implemented as

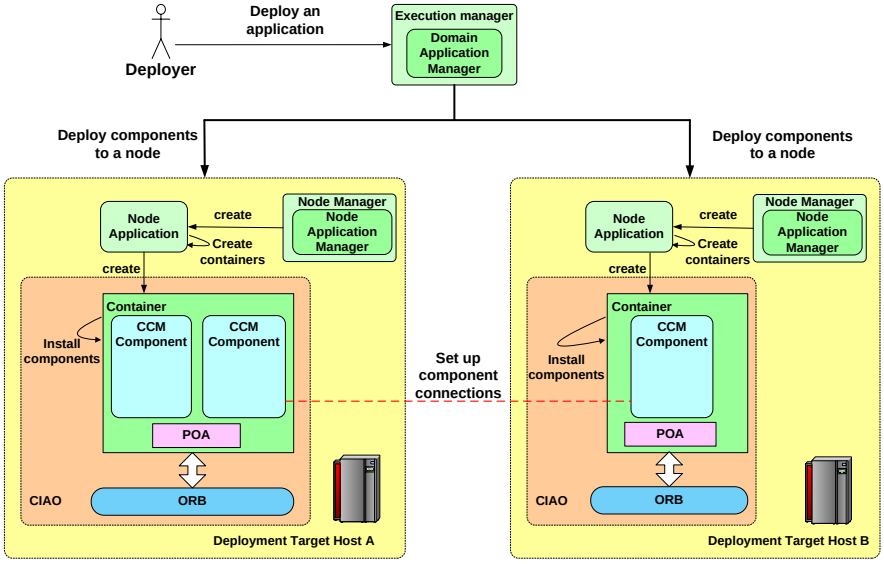

Figure 1.2: The Architecture of DAnCE

CORBA objects[1] that collaborate as follows. By default, DAnCE runs an `ExecutionManager` as a daemon to manage the deployment process for one or more *domains*, which are target environments consisting of *nodes*, *interconnects*, *bridges*, and *resources*. An `ExecutionManager` manages a set of `DomainApplicationManagers`, which in turn manage the deployment of components within a single domain. A `DomainApplicationManager` splits a deployment plan into multiple sub-plans, one for each node in a domain. A `NodeManager` runs as a daemon on each node and manages the deployment of all components that reside on that node, irrespective of the particular application with which they are associated. The `NodeManager` creates the `NodeApplicationManager`, which in turn creates the `NodeApplication` component servers that

[1]The DAnCE deployment infrastructure is implemented as CORBA objects to avoid the circular dependencies that would ensue if it was implemented as components, which would have to be deployed by DAnCE itself!

host application-specific containers and components.

To support additional D&C concerns not addressed by the OMG D&C specification, but which are essential for QoS-enabled DRE systems, we enhanced the specification-defined data model by describing additional deployment concerns, which include fine-grained system resource allocation and configuration. To enforce real-time QoS requirements, DAnCE extends the standards-based data model by explicitly defining (1) component server resource configuration as a set of *policies*, and (2) how components bind to these policies, which influence end-to-end QoS behavior.

Figure 1.3 shows the policies for server configurations that can be specified using the DAnCE *server resources XML schema*. With such enhancements, the DAnCE data model allows DRE system developers

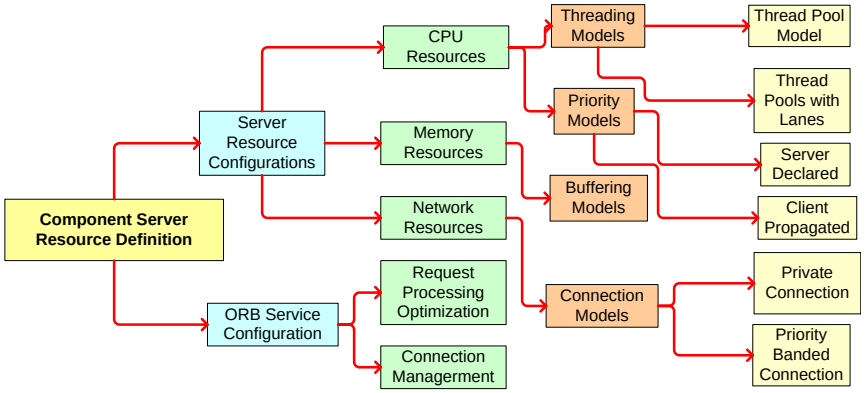

Figure 1.3: Specifying RT-QoS requirements

to configure and control *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service; *communication resources* via protocol properties and explicit bindings; and *memory resources* via buffering requests in queues and bounding the size of thread pools.

DAnCE's metadata-driven resource configuration mechanisms help ensure efficient, scalable, and predictable behavior end-to-end for DRE systems. Likewise, DAnCE helps enhance reuse since component QoS configuration can be performed at a much later phase, *i.e.*, just before the components are deployed into the target environment. Server resource specifications can be set via the following options: (1) *ORB command-line options*, which control TAO's connection management models, protocol selection, and optimized request processing, and (2) *ORB service configuration options*, which specify ORB resource factories that control server concurrency and demultiplexing models. Using this XML schema, a system deployer can specify the

designated ORB configurations. The ORB configurations defined by the data model are used to configure *NodeApplication* runtime entities that host components, thereby providing the necessary resources for the components to operate.

To fulfill the QoS requirements defined by the data model, DAnCE extends the standards-based runtime model as follows. An `XMLConfigurationHandler` parses the deployment plan and stores the information as IDL data structures that can transfer information between processes efficiently and enables the rest of DAnCE to avoid the runtime overhead of parsing XML files repeatedly. The IDL data structure output of the `XMLConfigurationHandler` is input to the `ExecutionManager`, which propagates the information to the `DomainApplicationManager` and `NodeApplicationManager`. The `NodeApplicationManager` uses the server resource configuration options in the deployment plan to customize the containers in the `NodeApplication` it creates. These containers then use other options in the deployment plan to configure TAO's Real-time CORBA support, including thread pool configurations, priority propagation models, and connection models.

4.3 Optimizing Static Configuration in DAnCE

To extend our QoS-enable component middleware to a wider range of applications and platforms, we have implemented static techniques and mechanisms for deployment and configuration where as much of the configuration lifecycle as possible is performed off-line. This approach is especially important for systems where availability of platform features or stringent constraints on configuration times would otherwise limit our ability to support component deployment and configuration services. The fundamental intuition in understanding our static configuration approach is that stages of the overall DRE system configuration lifecycle similar to those in the dynamic approach must still be supported. In our static approach, however, several stages of the lifecycle are *compressed* into the compile-time and system-initialization phases so that (1) the set of components in an application can be identified and analyzed before run-time and (2) the overheads for run-time operation following initialization are minimized and made predictable.

Due to the nuances of the platforms traditionally used for deploying DRE systems, not all features of conventional platforms are available or usable. In particular, dynamically linked libraries and on-line XML parsing are often either unavailable or too costly in terms of performance. We therefore move several

configuration phases earlier in the configuration lifecycle. We also ensure that our approach can be realized on highly constrained real-time operating systems, such as VxWorks or LynxOS, by re-factoring the configuration mechanisms to use only mechanisms that are available and efficient.

In CIAO's static configuration approach, the same automated build processes that manage compilation of the application and middleware code first insert a code generation step just before compilation. In this step, CIAO's XML configuration files (.cdp, .csr) are translated into C++ header files that are then compiled into efficient run-time configuration drivers, which in turn are linked statically with the main application. These run-time drivers are invoked during application (re)initialization at run-time, and rapidly complete the remaining on-line system configuration actions at that point.

With CIAO's static configuration approach, all XML parsing is moved before run-time to the off-line build process, and all remaining information and executable code needed to complete configuration is linked statically into the application itself. Each endsystem can then be booted and initialized within a single address space, and there is no need for inter-process communication to create and assemble components.

An important trade-off with this approach is that the allocation of component implementations and configuration information to specific endsystems must be determined in advance. In effect, this trade-off shifts support for run-time flexibility away from the deployment steps and towards the configuration steps of the system lifecycle. Our static configuration approach in CIAO thus maintains a reasonable degree of configuration flexibility, while reducing the run-time cost of configuration.

4.4 Model-Driven Deployment and Configuration with CoSMIC

Although DAnCE provides mechanisms that make component-based DRE systems more flexible and easier to develop and deploy, other complexities still exist. For example, using XML descriptors to configure the QoS properties of the system reduces the amount of code written imperatively. XML also introduces new complexities, however, such as verbose syntax, lack of readability at scale, and a high degree of accidental complexity and fallibility.

To simplify the development of applications based on Lightweight CCM and to avoid the complexities of handcrafting XML, we developed CoSMIC, which is an open-source MDE tool chain that supports the

deployment, configuration, and validation of component-based DRE systems. A key capability supported by CoSMIC is the definition and implementation of *domain-specific modeling languages* (DSMLs). DSMLs use concrete and abstract syntax to define the concepts, relationships, and constraints used to express domain entities [18]. One particularly relevant DSML in CoSMIC is the *Quality of Service Policy Modeling Language* (QoSPML) [19], which configures real-time QoS properties of components and component assemblies. QoSPML enables graphical manipulation of modeling elements and performs various types of generative actions, such as synthesizing XML-based data model descriptors defined in the OMG D&C specification and the enhanced data model defined in DAnCE. It also enables developers of DRE systems to specify and control the QoS policies via visual models, including models that capture the syntax and semantics of priority propagation model, thread pool model and connection model. QoSPML's model interpreter then automatically generates the XML configuration files, which allows developers to avoid writing applications that use the convoluted XML descriptors explicitly, while still providing control over QoS policies.

5 Applications of CIAO, DAnCE, and CoSMIC

This section describes our experiences using CIAO, DAnCE, and CoSMIC in several application domains, including avionics mission computing, shipboard computing, and inventory tracking. It also summarizes our lessons learned from these experiences.

5.1 Avionics Mission Computing Applications

Avionics mission computing applications [20] have two important characteristics that affect how components are deployed and configured in those systems. First, components are often hosted on embedded single-board computers within an aircraft, which are interconnected via a high speed backplane such as a VME-bus. Second, interacting components deployed on different aircraft must communicate via a low-and-variable bandwidth wireless communication link, such as Link-16. Figure 1.4 illustrates how image selection, download, and display components may be deployed in an avionics mission computing system.

The timing of the interactions between components within an aircraft is bound by access to the CPU, so that configuring the rates and priorities at which components invoke operations on other components is

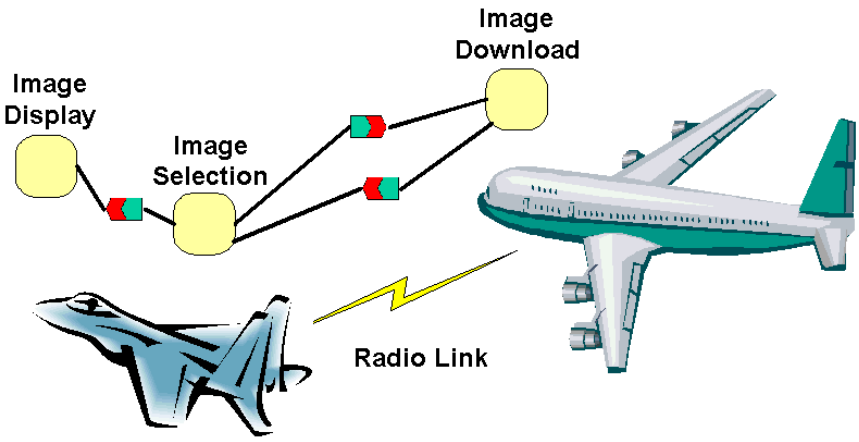

Figure 1.4: Component Interactions Within and Between Aircraft

crucial. Component connections involving receptacles that make prioritized calls to facets can be configured as standard Real-time CORBA calls in CIAO, which maps those configuration options directly onto TAO features, as described in Section 4.1. Moreover, since components are deployed statically in these systems, the static deployment and configuration techniques described in Section 4.3 are applicable to these systems. In fact, they are often necessary due to stringent constraints on system initialization times and limitations on platform features, *e.g.*, the absence of dynamic library support on some real-time operating systems.

Even within an aircraft, however, it is often appropriate to use more dynamic forms of resource management, such as using hybrid static/dynamic scheduling to add non-critical processing to the system without violating deadlines for critical processing [21]. These more dynamic forms of resource management require configuration of more sophisticated interconnection services between component ports, such as TAO's Real-Time Event Channel [22]. We are currently expanding the configuration options exposed by CIAO to include attributes of the event channel and scheduling services provided by TAO.

Applications whose component interactions span multiple aircraft *mandate* even more dynamic forms of resource management, such as using adaptive scheduling together with adaptive control to manage quality of service (QoS) properties of application data transmission and processing [23, 20] across a low-and-variable wireless communication link. We are currently adding configurable adaptive resource management capabilities to CIAO so system developers can configure QoS properties end-to-end in DRE systems.

5.2 Shipboard Computing Applications

Modern shipboard computing environments consist of a grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations that support multiple *simultaneous* QoS requirements, such as survivability, predictability, security, and efficient resource utilization. To meet these QoS requirements, *dynamic resource management* (DRM) [24, 25] can be applied to optimize and (re)configure the resources available in the system to meet the changing needs of applications at runtime. The primary goal of DRM is to ensure that enterprise DRE systems can adapt dependably in response to dynamically changing conditions (*e.g., evolving multi-mission priorities*) to ensure that computing and networking resources are best aligned to meet critical mission requirements. A key assumption in DRM technologies is that the levels of service in one dimension can be coordinated with and/or traded off against the levels of service in other dimensions to meet mission needs, *e.g.*, the security and dependability of message transmission may need to be traded off against latency and predictability.

In the DARPA's *Adaptive and Reflective Middleware Systems* (ARMS) program DRM technologies were developed and applied to coordinate a computing grid that manages and automates many aspects of shipboard computing. As shown in Figure 1.5, the ARMS DRM architecture integrates resource management

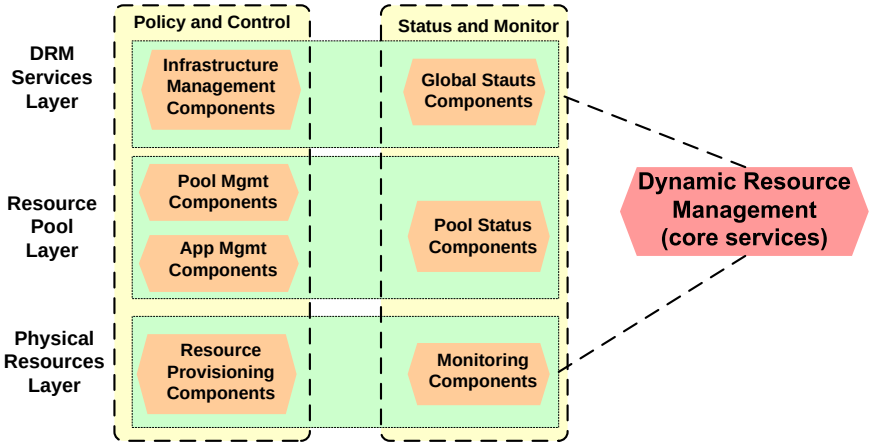

Figure 1.5: DRM Layered Architecture in a Shipboard Computing Environment

and control algorithms enabled by standards-based component middleware infrastructure, and is modeled as a layered architecture comprising components at different levels of abstraction, including the DRM Services layer, the Resource Pool layer, and the Physical Resources layer. The top level **DRM Services layer** is responsible for satisfying global shipboard computing missions, such as mission mode changes. It then de-

composes global mission requests into a coordinated set of requests on resource pools within the **Resource Pool layer**, which is an abstraction for a set of computer nodes managed by a local resource manager. The **Physical Resources layer** deals with specific instances of resources within a single node in the system.

To simplify development and enhance reusability, all the three layers of the ARMS DRM infrastructure are implemented using CIAO, DAnCE and CoSMIC, which simplifies and automates the (re)deployment and (re)configuration of DRE and application components in a shipboard computing environment. The ARMS DRM system has hundreds of different types and instances of infrastructure components written in ∼300,000 lines of C++ code and residing in ∼750 files developed by different teams from different organizations. ARMS DRM research and experiments [26] show that DRM using standards-based middleware technologies is not only feasible, but can (1) handle dynamic resource allocations across a wide array of configurations and capacities, (2) provide continuous availability for critical functionalities–even in the presence of node and pool failures–through reconfiguration and redeployment, and (3) provide QoS for critical operations even in overload conditions and resource-constrained environments.

5.3 Inventory Tracking Applications

An *inventory tracking system* (ITS) is a warehouse management infrastructure that monitors and controls the flow of items and assets within a storage facility. Users of an ITS include couriers (such as UPS, DHL, and Fedex), airport baggage handling systems, and retailers (such as Walmart and Target). An ITS provides mechanisms for managing the storage and movement of items in a timely and reliable manner. For example, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the inventory throughout a highly distributed system (which may span organizational boundaries), and track warehouse assets using decentralized operator consoles. In conjunction with colleagues at Siemens [2], we have developed and deployed an ITS using CIAO, DAnCE, and CoSMIC.

Successful ITS deployments must meet both the functional and QoS requirements. Like many other DRE systems, an ITS is assembled from many independently developed reusable components. As shown in Figure 1.6, some ITS components (such as an `OperatorConsole`) expose interfaces to end users, *i.e.*, ITS operators. Other components (such as a `TransportUnit`) represent hardware entities, such as cranes,

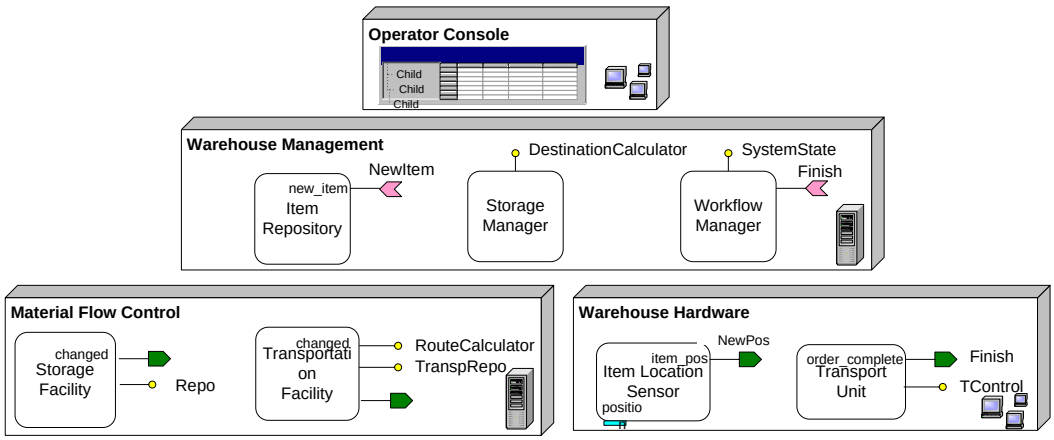

Figure 1.6: The Architecture of an Inventory Tracking System

forklifts, and belts. Database management components (such as `ItemRepository` and `StorageFacility`) expose interfaces to manage external backend databases (such as those tracking items inventory and storage facilities). Finally, the sequences of events within the ITS is coordinated by control flow components (such as the `WorkflowManager` and `StorageManager`). Our QoS-enabled component middleware and MDE tools allow ITS components to be developed independently.

After individual components are developed, there are still a number of crosscutting concerns that must be addressed when assembling and configuring an ITS, including (1) identifying dependencies between component implementation artifacts, such as the `OperatorConsole` component having dependencies on other ITS components (*e.g.*, a `WorkflowManager` component) and other third-party libraries (*e.g.*, the QT library, which is a cross-platform C++ GUI library compatible with the Embedded Linux OS), (2) specifying the interaction behavior among ITS components, (3) specifying components to configure and control various resources, including processor resources, communication resources, and memory resources, and (4) mapping ITS components and connections to the appropriate nodes and networks in the target environment where the ITS will be deployed. We used the CoSMIC MDE tool chain to simplify the assembly of ITS components and then used DAnCE to deploy and configure them.

QoS requirements (such as latency and jitter) are important considerations of an ITS. For instance, in an ITS, whenever a hardware unit (such as a conveyor belt) fails, the component that controls and monitors this hardware unit must notify another ITS component (such as a `WorkflowManager` component) in real-time to avoid system damage and to minimize overall warehouse delay. Likewise, other components that

monitor the location of items, such as an `ItemLocationSensor`, must have stringent QoS requirements since they handle real-time item delivery activities. To satisfy these real-time QoS requirements, the CIAO `NodeApplications` hosting `ItemLocationSensor` components can be configured declaratively with a server-declared priority model at the highest CORBA priority, with thread pools having preset static threads, and with priority-banded connections.

5.4 Lessons Learned

The following are our lessons learned from applying CIAO, DAnCE, and CoSMIC to the three kinds of DRE systems described above:

• **Developing complex DRE systems require a sophisticated and integrated engineering paradigm.** While modern DRE systems are increasing in size and complexity, creating stove-piped one-of-a-kind applications is unsatisfying, although that has been the state-of-the-practice until recently. Component middleware is designed to enhance the quality and productivity of software developers by elevating the level of abstraction used to develop distributed systems. Conventional component middleware, however, lacks mechanisms to separate QoS concerns from component functional aspects, and it also lacks mechanisms to handle deployment and configuration for such concerns in DRE systems. We therefore designed and integrated CIAO, DAnCE, and CoSMIC based on advanced software principles, such as separation of concerns, metaprogramming, component-based software engineering, and model-driven engineering. We incorporated these concepts as key design principles underlying these technologies and used them to develop representative applications in a variety of DRE system domains.

• **Model-Driven Engineering (MDE) tools alleviate complexities associated with component middleware.** Although component middleware elevates the abstraction level of middleware to enhance software developer quality and productivity, it also introduces new complexities. For example, the OMG Lightweight CCM and D&C specifications have a large number of configuration points. To alleviate these complexities we designed and applied the CoSMIC MDE tools. Our experiences in applying CIAO and DAnCE to various application domains showed that when component deployment plans are incomplete

or must change, the effort required is significantly less than using the raw component middleware without MDE tool support since applications can evolve from the existing domain models. Likewise, if the component assemblies are the primary changing concerns in the system, little new application code must be written, yet the complexity of the MDE tool remains manageable due to the limited number of well-defined configuration "hot spots" exposed by the underlying infrastructure.

• **Automated MDE tools are needed to configure and deploy DRE systems effectively.** Despite the fact that CIAO and DAnCE component middleware facilitate the configuration of complex DRE systems based on Real-time CORBA and Lightweight CCM, developers are still faced with the question of what constitutes a "good" configuration. Moreover, they are still ultimately responsible for determining the appropriate configurations. We observed that scheduling analysis and verification tools tools would be helpful in performing this evaluation and should be integrated into MDE toolsuite to help system designers address such challenges. In addition, despite the benefits of using visual MDE tools to describe different aspects of the large scale DRE systems, it is still tedious and error-prone to manually show the details for all the components in large-scale DRE systems. This observation motivates the need for further research in automating the synthesis of large-scale DRE systems based on the different types of meta-information about assembly units, such as high-level design intention or service requirements.

6 Related Work

This section compares our work on CIAO and DAnCE with other related work on middleware for DRE systems. In particular, we compare two different categories of middleware for DRE systems: QoS-enabled DOC middleware and QoS-enabled component middleware.

QoS-enabled DOC middleware. The *Quality Objects* (QuO) framework [27, 28] uses aspect-oriented software development [29] techniques to separate the concerns of QoS programming from application logic in DRE systems. It allows application developers to explicitly declare system QoS characteristics, and then use a specialized compiler to generate code that integrates those characteristics into the system implementation. It therefore provides a high-level QoS abstraction on top of conventional DOC middleware technologies such

as CORBA and Java RMI.

The *dynamicTAO* [30] project applies reflective techniques to reconfigure ORB components at runtime. Similar to *dynamicTAO*, the *Open ORB* project [31] also aims at highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements. This approach may not be suitable for some DRE systems, however, since dynamic loading and unloading of ORB components can incur unpredictable overheads and thus prevent the ORB from meeting application deadlines. Our work on CIAO allows MDE tools, such as Cadena [32] and QoSPML [19], to analyze the required ORB components and their configurations and ensure that a component server contains only the required ORB components.

Proactive [33] is a distributed programming model for deploying object-oriented grid applications and is similar to CIAO and DAnCE since it also separately describes the target environment using XML descriptors, but CIAO goes further to allow application to be explicitly composed with a number of components, and CIAO and DAnCE together ensure end-to-end system QoS at deployment time.

QoS-enabled component middleware. The container architecture in component middleware provides a vehicle for applying meta-programming techniques for QoS assurance control in component middleware. Containers can also help apply aspect-oriented software development techniques to plug in different systemic behaviors [34]. This approach is similar to CIAO in that it provides mechanisms to inject aspects into applications at the middleware level.

[35] further develops the state of the art in QoS-enabled containers by extending a QoS EJB container interface to support a `QoSContext` interface that allows the exchange of QoS-related information among component instances. To take advantage of the QoS-container, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement, however, adds an unnecessary dependency to component implementations.

The QoS Enabled Distributed Objects (Qedo) project [36] is another effort to make QoS support an integral part of CCM. Qedo's extensions to the CCM container interface and Component Implementation Framework (CIF) require component implementations to interact with the container QoS interface and negotiate the level of QoS contract directly. While this approach is suitable for certain applications where QoS is part of the functional requirements, it inevitably tightly couples the QoS provisioning and adaptation behav-

iors into the component implementation, and thus hampers the reusability of the component. In comparison, CIAO explicitly avoids this coupling and *composes* the QoS aspects into applications declaratively.

The OpenCCM [37] project is a Java-based CCM implementation. The OpenCCM Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. OpenCCM and its DCI infrastructure, however, do not support key QoS aspects for DRE systems, including real-time QoS policy configuration and resource management.

7 Concluding Remarks

This chapter has described how the CIAO and DAnCE QoS-enabled component middleware and CoSMIC MDE tools have been applied to address key challenges when developing DRE systems for various application domains. Reusability and composability are particularly important requirements for developing large-scale DRE systems. With QoS-enabled DOC middleware, such as Real-time CORBA and Real-time Java, configurations are made *imperatively* via calls to programmatic configuration interfaces with which the component code is thus tightly coupled. In contrast, QoS-enabled component middleware allows developers to specify configuration choices *declaratively* through metaprogramming techniques (such as XML descriptor files), thus enhancing reusability and composability.

Our future work will focus on (1) developing a computational model to support configuration and dynamic reconfiguration of DRE systems at different levels of granularity to improve both system manageability and reconfiguration performance, and developing a platform model to execute the computational model predictably and scalably, (2) applying specialization techniques (such as partial evaluation and generative programming) to optimize DRE systems using metadata contained in component assemblies, and (3) enhancing DAnCE to provide state synchronization and component recovery support for a fault-tolerant middleware infrastructure, such as MEAD [38].

TAO, CIAO, and DAnCE are available for download at `deuce.doc.wustl.edu/Downlad.html`, and CoSMIC is available for download at `www.dre.vanderbilt.edu/cosmic`.

References

[1] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," *CrossTalk - The Journal of Defense Software Engineering*, Nov. 2001.

[2] A. Nechypurenko, D. C. Schmidt, T. Lu, G. Deng, and A. Gokhale, "Applying MDA and Component Middleware to Large-scale Distributed Systems: a Case Study," in *Proceedings of the 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, (Enschede, Netherlands), IST, Mar. 2004.

[3] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Dec. 2002.

[5] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, pp. 265–290, November/December 1996.

[6] N. Wang, D. C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

[7] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.

[8] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-time Specification for Java*. Addison-Wesley, 2000.

[9] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

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

[11] Object Management Group, *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., May 2003.

[12] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2006 (to appear).

[13] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, 2006.

[14] 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.

[15] 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.

[16] I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-time CORBA Quality of Service," *IEEE Proceedings Special Issue on Real-time Systems*, vol. 91, July 2003.

[17] OMG, *Deployment and Configuration Adopted Submission*, Document ptc/03-07-08 ed., July 2003.

[18] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.

[19] S. Paunov, J. Hill, D. C. Schmidt, J. Slaby, and S. Baker, "Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System QoS," in *Proceedings of 13th Annual International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '06)*, (Potsdam, Germany), IEEE, Mar. 2006.

[20] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," *Journal of Real-time Systems*, vol. 29, no. 2–3, pp. 101–130, 2005.

[21] C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-time Embedded Computing," *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, vol. 91, Jan. 2003.

[22] 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.

[23] X. Wang, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill, "CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission," in *Proc. of the 10th IEEE Real-time and Embedded Tech. and Applications Symp. (RTAS)*, (Toronto, Canada), May 2004.

[24] L. Welch, B. Shirazi, B. Ravindran, and C. Bruggeman, "DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-time Systems," in *Proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS'98)*, Sept. 1998.

[25] J. Hansen, J. Lehoczky, and R. Rajkumar, "Optimization of Quality of Service in Dynamic Systems," in *Proceedings of the 9th International Workshop on Parallel and Distributed Real-time Systems*, Apr. 2001.

[26] J. Balasubramanian, P. Lardieri, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano, "A multi-layered resource management framework for dynamic resource management in enterprise dre systems," *Journal of Systems and Software: special issue on Dynamic Resource Management in Distributed Real-time Systems*, 2006.

[27] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging Quality of Service Control Behaviors for Reuse," in *Proceedings of the* 5^{th} *IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Crystal City, VA), pp. 375–385, IEEE/IFIP, April/May 2002.

[28] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[29] 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*, pp. 220–242, June 1997.

[30] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.

[31] G. Blair, G. Coulson, F. Garcia, D. Hutchinson, and D. Shepherd, "The Design and Implementation of Open ORB 2." dsonline.computer.org/0106/features/bla0106_print.htm, 2001.

[32] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, (Portland, OR), May 2003.

[33] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, "Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications," in *Proc. of the 11th International Symposium on High Performance Distributed Computing (HPDC'02)*, (Edinburgh, UK), July 2002.

[34] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers," *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[35] M. A. de Miguel, "QoS-Aware Component Frameworks," in *The* 10^{th} *International Workshop on Quality of Service (IWQoS 2002)*, (Miami Beach, Florida), May 2002.

[36] FOKUS, "Qedo Project Homepage." `http://qedo.berlios.de/`.

[37] A. Flissi, C. Gransart, and P. Merle, "A component-based software infrastructure for ubiquitous computing," in *Proceedings of the 4th International Symposium on Parallel and Distributed Computing*, July 2005.

[38] P. Narasimhan, T. Dumitras, A. Paulos, S. Pertet, C. Reverte, J. Slember, and D. Srivastava, "MEAD: Support for Real-time Fault-Tolerant CORBA," *Concurrency and Computation: Practice and Experience*, 2005.