DEPLOYMENT AND CONFIGURATION OF COMPONENT-BASED DISTRIBUTED, REAL-TIME AND EMBEDDED SYSTEMS

By

Gan Deng

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2008

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

Dr. Janos Sztipanovits

Dr. Gabor Karsai

Dr. Jeff Gray

To dear Mom and Dad for their patient support over the years

To dear Hui for the love over the years

ACKNOWLEDGMENTS

I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate life has been one that I will cherish forever.

I wish to express my deepest gratitude to my advisor, Dr. Douglas C. Schmidt, for his invaluable intellectual support, encouragement, friendship, enthusiasm, and wisdom which made it possible for me to complete this dissertation. I have been amazingly fortunate to have Doug as my advisor, who taught me how to question thoughts and express ideas. His patience and support helped me overcome many difficulties and finish this dissertation. I would also like to express my gratitude to my co-advisor Dr. Aniruddha (Andy) Gokhale, who has been always there to listen and give advice. I am deeply grateful to him for the discussions that helped me flesh out research ideas. I am thankful to both of them for their carefully reading and commenting on countless revisions of many papers we have written together.

I would like to express my gratitude to Dr. Janos Sztipanovits, Dr. Gabor Karsai and Dr. Jeff Gray for serving as my dissertation committee and contributing suggestions for the improvement of my dissertation. I am especially grateful for the time Jeff devoted to reviewing my proposal and dissertation draft and would like to thank him for his help. I would also like to thank Dr. Chris Gill at Washington University for his helpful suggestions about my research.

Over the years, my research has been supported by a variety of agencies and companies. I would like to take this opportunity to thank the following people: John Slaby and Steve Baker at Raytheon Co., Partick Lardieri, Gautam Thaker at Lockheed Martin Advanced Technology Laboratories, Dipa Suri at Lockheed Martin Advanced Technology Center, and Gunther Lenz at Siemens Corporate Research.

I shared the experience of my graduate study with an extraordinarily talented and supportive group of lab mates. The discussions that I have shared with the following individuals Jaiganesh Balasubramanian, Krishnakumar Balasubramanian, James Hill, Joe Hoffert, Amogh Kavimandan, William Otte, Jeff Parsons, Nishanth Sankaran, Sumant Tambe, Jules White and Ming Xiong at Vanderbilt University will be fresh in my memory for a long time.

Last, but not least, I want to express my deepest appreciation to my parents and my brother as well as my beloved, Ms. Hui (Lindsey) Yang, who provided continuous spiritual support, encouragement and love throughout my graduate life, without which my PhD study would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Emerging Trends and Technologies

Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources or QoS requirements is an important and challenging R&D problem. Systems with such characteristics are called *open* DRE systems [29] since they operate in an open environment and must be prepared to accommodate changing operating conditions or requirements, such as power levels, CPU/network bandwidth or mission modes. Examples of open DRE systems include shipboard computing environments [109], next-generation coordinated unmanned air vehicle systems [44], intelligence, surveillance and reconnaissance systems [111], and large-scale warehouse inventory tracking systems [19].

Such systems operate in open environments where system operational conditions, input workload, and resource availability cannot be characterized accurately *a priori*. As a result, the requirements of open DRE systems can be characterized as follows:

- Multiple quality of service (QoS) properties, such as predictable latency/jitter/throughput, scalability, dependability, and security, must be satisfied simultaneously and often in real-time;

- Different levels of service will occur under different configurations, environmental conditions, and cost, and must be handled judiciously by system infrastructure and applications;

- The need for autonomous and time-critical application behavior requires flexible system infrastructure and application components that can be dynamically deployed at run-time in a *predictable* manner to accommodate mission mode changes or environmental changes.

Due to these characteristics, open DRE systems become more large in scale, complex, and hard to control, which often results in high development cost, low productivity and unmanageable software quality. Consequently, there is a growing demand for a new, efficient, and cost-effective software development paradigm. One of the most promising approaches for open DRE systems is *component based software engineering* (CBSE), which is a promising software engineering paradigm for achieving systematic reuse and composition of software artifacts [37].

CBSE promises to reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of software systems [34]. This paradigm has raised a tremendous amount of interest both in the research community and in the software industry. It significantly differs from the traditional functional software design approach in the way that large-scale software systems are assembled from commercial off-the-shelf (COTS) components, which were in turn developed by different software organizations, rather than developed from scratch. The key concept of CBSE is "component". A component in CBSE has three main features. First, a component is an independent and replaceable part of a system that fulfills a well-defined function. Second, a component works in the context of well-defined system architecture. Finally, a component communicates with other components through its interfaces.

The lifecycle of CBSE is shown in Figure I.1, where COTS components can be obtained from a component repository, assembled together through certain well-defined interfaces, configured based on given functional requirements (*e.g.*, feature requirements) and non-functional requirements (*e.g.*, QoS requirements), and then deployed into the target execution platforms.

Although CBSE is a very promising software engineering paradigm for DRE systems, to ensure DRE systems can be effectively developed, packaged and assembled, a

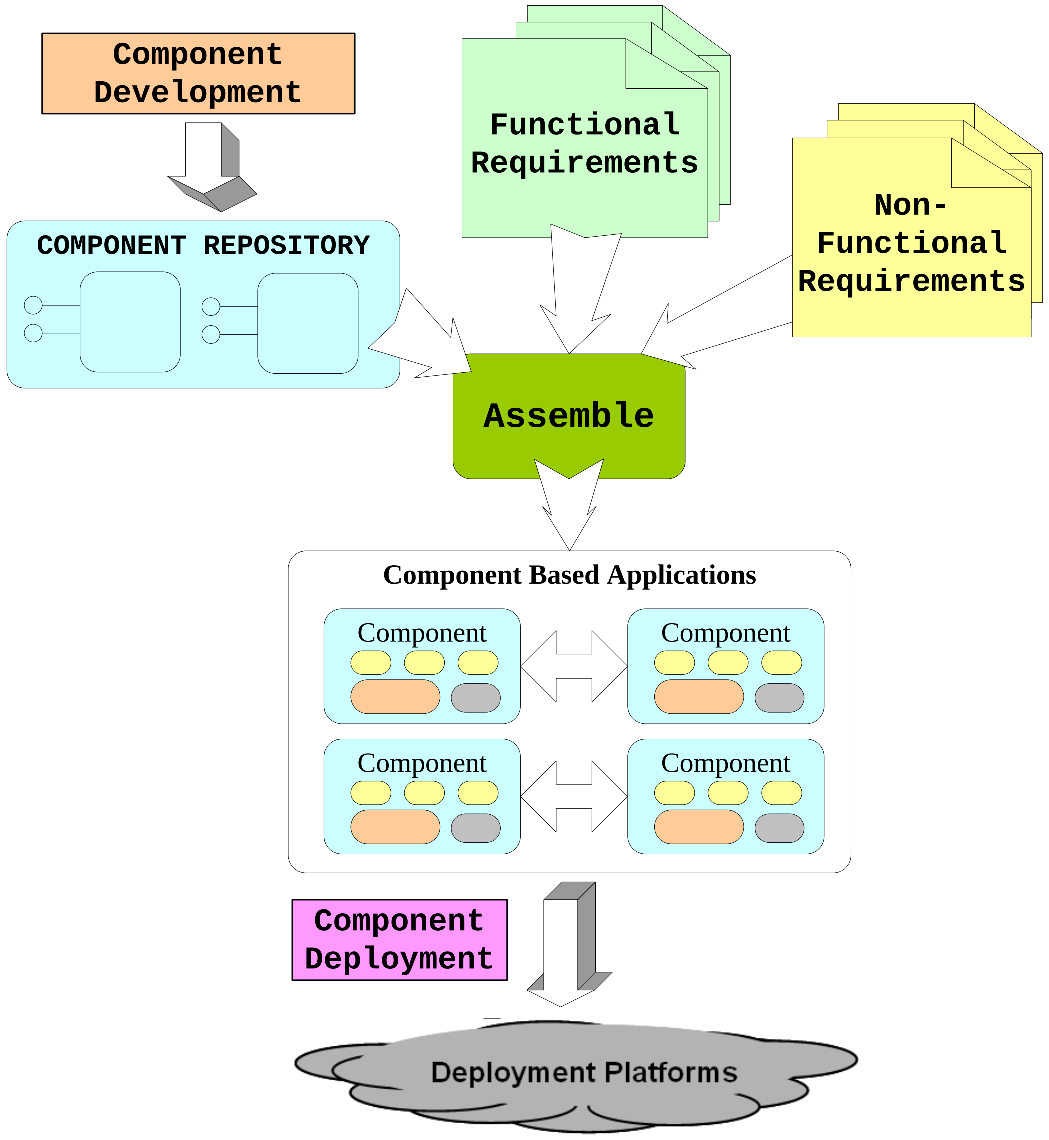


Figure I.1: Overview of Component Based Software Engineering

system architecture that supports the CBSE process must be present. According to prior research [10, 33, 53], system architectures of component-based software systems are often in the form of a layered and modular architecture. Over the past decade, various system architectures have been devised to alleviate many complexities associated with developing and applying execution environments to DRE systems. Their successes have added a new category of software to the familiar operating system, programming language, and networking offerings of prior generations. In particular, some of the most successful emerging execution environment technologies have centered on *middleware*, which is software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware.

Middleware technologies were invented originally to help simplify the development, execution, and management of distributed computing systems [7, 102], and bring those capabilities within the reach of many more developers than the few experts at the time who could master the complexities of these environments. Middleware was necessary since complex system integration requirements were not being met from either the application perspective, where it was too hard for most application developers and not reusable, or the network or host operating system perspectives, which were focused on providing the communication and endsystem resource management layers, respectively, rather than the distributed computing and communication layers. The primary role of middleware is to (1) functionally bridge the gap between application programs and the lower-level hardware and software infrastructure to coordinate how parts of applications are connected and how they execute and interoperate, (2) enable and simplify the integration of components developed by multiple technology suppliers, and (3) provide a common reusable accessibility for functionality and patterns that formerly were placed directly in applications, but in actuality are application-independent and need not be developed separately for each new application.

Middleware isolates DRE applications from lower-level infrastructure complexities, such as heterogeneous platforms and error-prone network programming mechanisms. It also enforces essential end-to-end quality of service (QoS) properties, such as low latency and bounded jitter; fault propagation/recovery across distribution boundaries; authentication and authorization; and weight, power consumption, and memory footprint constraints.

Over the past decade, middleware has evolved to support the creation of applications via

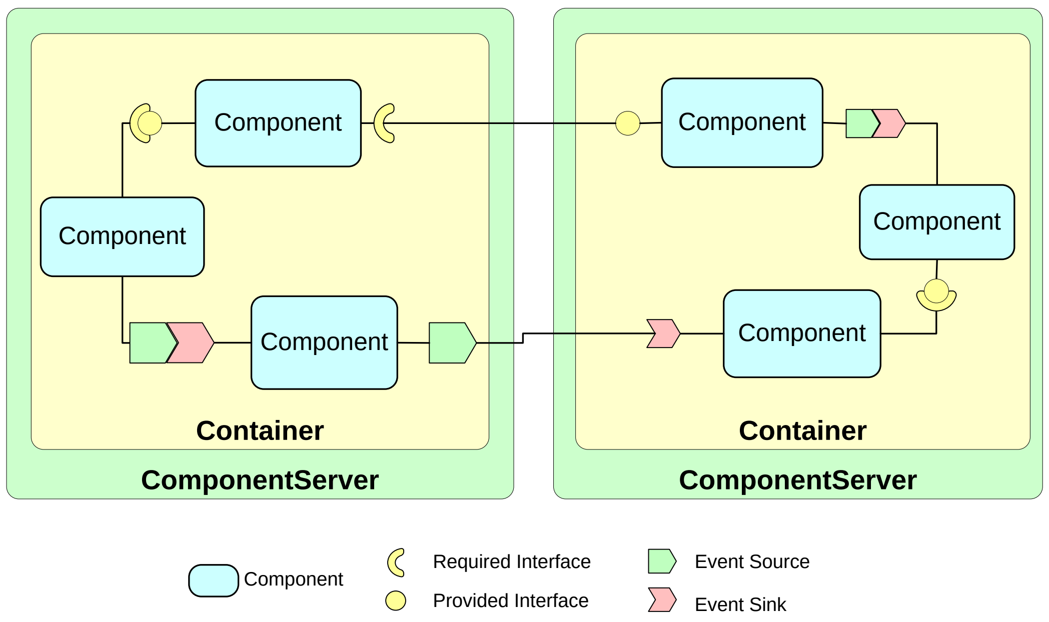


Figure I.2: Overview of Component Middleware

composition of reusable and flexible software components. Components are implementation/integration units with precisely-defined interfaces that can be installed in application server run-time environments. Examples of COTS component middleware include the CORBA Component Model (CCM) [78], Enterprise Java Beans (EJB) [68], and Microsoft .NET [121].

I.2 Overview of Component Middleware and Deployment and Configuration

Component middleware technologies raise the level of abstraction by providing higher-level entities like components and containers. Components encapsulate "business" logic, and interact with other components via *ports*.

As shown in Figure I.2, key elements and benefits of component middleware technologies include:

- **Component**, which is the basic building block used to encapsulate an element of cohesive functionality. Components separate application logic from the underlying middleware infrastructure.

- **Component ports**, which allow a component to expose multiple views to clients. Component ports provide the primary means for connecting components together to form assemblies.

- **Component Assembly**, which is an abstraction for composing components into larger reusable entities. A component assembly typically includes a number of components connected together in an application-specific fashion. Unlike the other entities described here, there is no run-time entity corresponding to a component assembly.

- **Container**, which is a high-level execution environment that hosts components and provides them with an abstraction of the underlying middleware. Containers provide clear boundaries for QoS policy configuration and enforcement, and are also the lowest unit at which policy is enforced and it regulates shared access to the middleware infrastructure by the components.

- **Component server**, which aggregates multiple containers and the components hosted in them in a single address space, *e.g.*, an OS process. Component servers facilitate management at the level of entire applications.

Components interact with clients (including other components) via component ports. Component ports implement the Extension Interface pattern [108], which allows a single component to expose multiple views to clients. For example, CCM defines four different kinds of ports:

- **Facets**, which are distinct named interfaces provided by the component. Facets enable a component to export a set of distinct—though often related—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 enable a component to use the functionality provided by facets of other components.

- **Event sources and sinks**, which define a standard interface for the Publish/Subscribe pattern [9]. Event sources/sinks are named connection points that send/receive specified types of events to/from one or more interested consumers/suppliers. These ports also hide the details of establishing and configuring event channels [35] needed to support the publish/subscribe pattern.

- **Attributes**, which are named values exposed via accessor and mutator operations. Attributes can be used to expose the properties of a component 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 clients to trigger an action based on the value of the attributes.

After components are developed and component assemblies are defined, they must be deployed and configured properly by deployment and configuration (D&C) services. The D&C process of component-based systems usually involves a number of service objects that must collaborate with each other. Figure I.3 gives an overview of the OMG D&C model, which is standardized by OMG to promote component reuse and allow complex applications to be built by assembling existing components. As shown in the figure, since a component-based system often consists of many components that are distributed across multiple nodes, in order to automate the D&C process, these service objects must be distributed across the targeted infrastructure and collaborate remotely.

The run-time of the OMC D&C model standardizes the D&C process into a number of

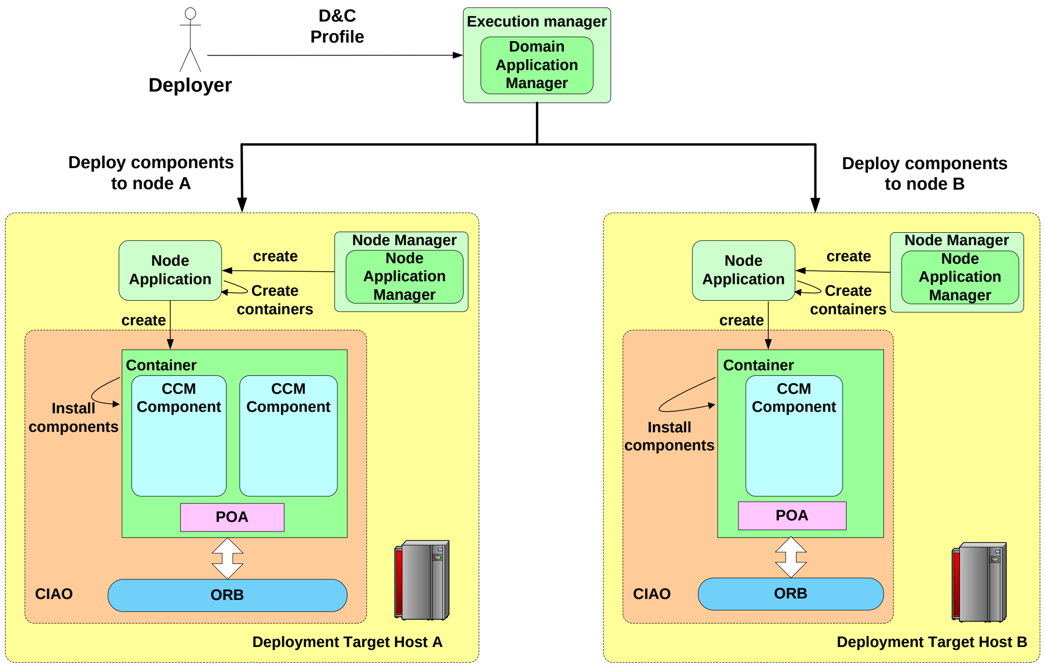


Figure I.3: An Overview of OMG Deployment and Configuration Model

serialized phases. The OMG D&C Model defines the D&C process as a two-level architecture, one at the domain level and one at the node level. Since each deployment task involves a number of subtasks that have explicit dependencies with each other, these subtasks must be serialized and finished in different phases. Meanwhile, each deployment task involves a number of node-specific tasks, so each task is distributed.

I.3 Research Challenges

Although component middleware and the associated D&C model provide a number of advantages over previous technologies, several new challenges arise. Some of the key challenges in deploying and configuring component-based large-scale DRE systems include:

1. **Complexities associated with deploying and configuration components to meet real-time QoS.**

Conventional middleware D&C platforms are poorly suited for assembling DRE systems from pre-existing components that must meet real-time QoS requirements. For example, these D&C platforms do not separate real-time QoS concerns (such as component server threading and priority models) from application business logic and lifecycle management. It is therefore hard to reuse components for DRE systems, and the onus is on humans to manage these crosscutting concerns manually via *ad hoc* techniques, which impedes productivity and quality. Furthermore, for open DRE systems where system resources are both constrained and variable in time, having the ability to dynamically deploy components and system resources at run-time to provide desired QoS becomes an essential requirement. Existing QoS-enabled middleware technologies lack dynamic deployment and configuration capabilities so they are not suitable for open DRE systems. Thus, a key research challenge is the lack of D&C middleware that focuses on systemic QoS issues and allows components and system resources to be deployed and configured at both initial deployment time and run-time.

2. **Complexities associated with deploying and configuring real-time publish/subscribe services in component middleware.**

   The increasing use of QoS-enabled component middleware in DRE systems compounded by the need for real-time publish/subscribe services to support a large class of DRE systems requires the integration of the real-time publish/subscribe paradigm within QoS-enabled component middleware. Although QoS-enabled publish/subscribe middleware services are available in distributed object computing (DOC) middleware platforms such as CORBA 2.x [74], unfortunately standards-based component middleware do not yet specify how publish/subscribe services can be robustly supported within component middleware. Moreover, to date there is a general lack of systematic studies that address these concerns. Instead, developers rely on *ad hoc* techniques (such as writing code using third-generation programming languages like C++ or

Java) to provision real-time publish/subscribe capabilities in DRE systems. Thus, a key research challenge is the lack of D&C middleware that allows different real-time publish/subscribe services to be integrated seamlessly into component-based DRE systems without sacrificing system performance and component reusability.

3. **Lack of assurance to meet the predictability requirement of D&C middleware.**

   The predictability requirement of D&C mechanisms is a key factor for open DRE systems to meet their QoS requirements due to mission mode changes or environmental changes. Open DRE systems are often large and complex. To manage the overall complexity of such systems, open DRE system are often decomposed into many domain-related tasks that can be modeled as *operational strings* [59]. Operational strings are assemblies of software components designed by software architects that accomplish certain domain specific tasks. Unfortunately, when multiple operational strings need to be dynamically deployed and if dependencies exist among these operational strings, *priority inversion* can happen at deployment time. Existing D&C techniques only take the dependency between operational strings into account while ignoring their priorities, which can cause deployment priority inversions for open DRE systems, which adversely affects the predictability of D&C of component-based DRE systems. Thus, a key research challenge is the lack of a D&C middleware framework that can ensure the predictability when a number of operational strings need to be deployed at run-time.

I.4 Research Approach

To address the challenges in the previous section, this dissertation has developed an approach centered around QoS-enabled deployment and configuration of DRE systems. The different dimensions of this approach are shown in Figure I.4 and described below.

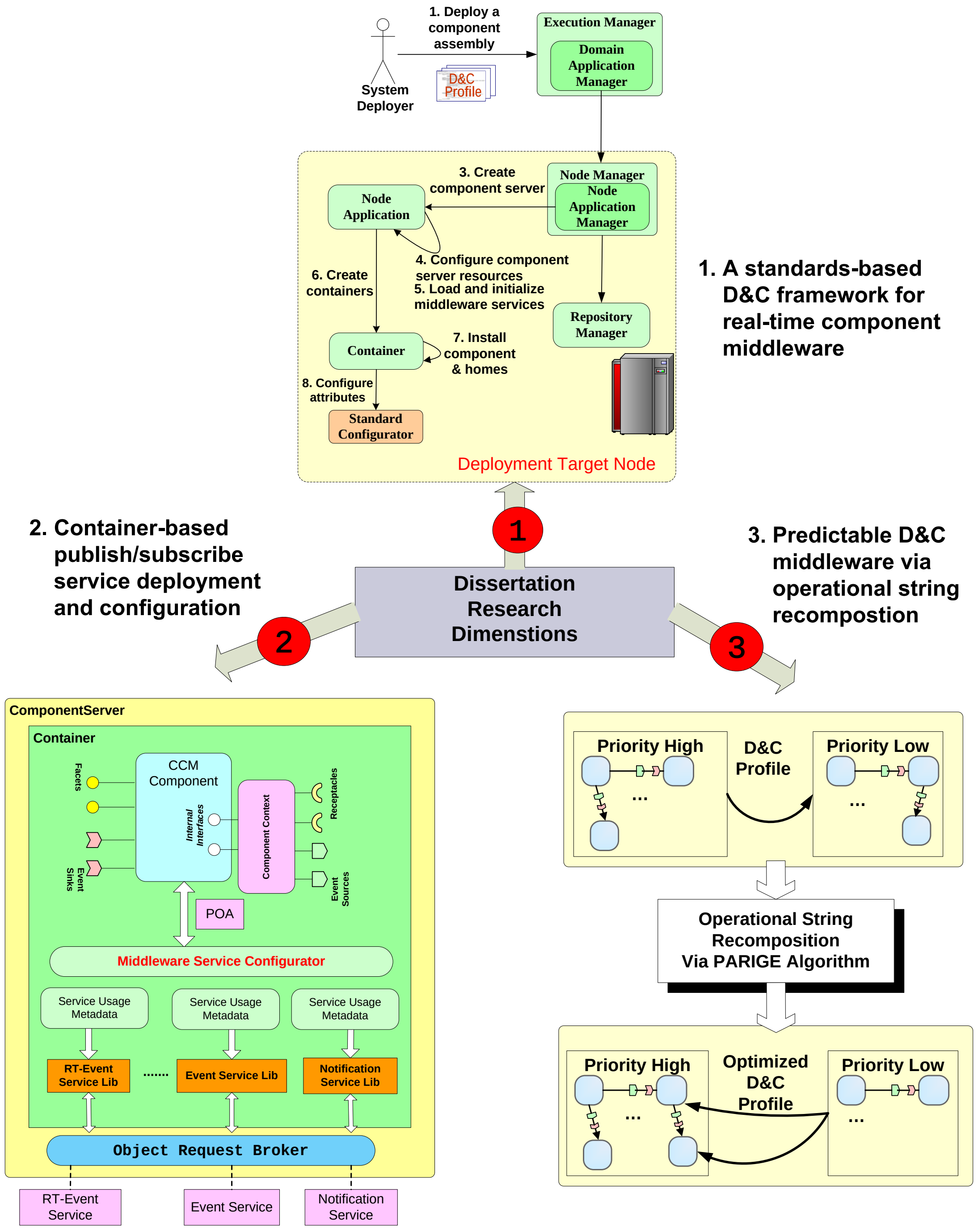


Figure I.4: Three Dimensions of Research

1. QoS-enabled Component Deployment and Configuration Technique. To effectively enforce real-time QoS requirements of component-based DRE systems based on particular deployment and configuration scenarios, we treat real-time QoS concerns as "extrinsic" concerns and decouple them from the "intrinsic" core business logic of components. The run-time D&C framework can then be used to automate the deployment and configuration of such systemic QoS concerns by treating them as an integral part of the D&C process. Since real-time QoS concern is highly configurable and can be affected by many real-time policies, our approach raises the abstraction level of these real-time QoS policies by defining them as first-class elements in the D&C middleware and allowing them to be specified declaratively via high-level configuration languages. The novelty of this approach stems from the *separation of concerns* [23, 120] technique it uses to decouple orthogonal non-functional system concerns from core functional concerns as part of the D&C process. These concerns can be manipulated at both initial deployment time and run-time to ensure system real-time QoS. Chapter III describes the approach for QoS-enabled component deployment and configuration in detail.

2. Publish/Subscribe Service Integration, Configuration and Deployment Technique. By analyzing the benefits and limitations of different design choices for integrating real-time publish/subscribe services into QoS-enabled component middleware, we take a container-managed integration approach for real-time publish/subscribe services and present a pattern language for its architectural design. This approach leverages the benefits of the component-oriented software development paradigm and promotes the reusability of components without sacrificing the performance of the system. In addition, by leveraging a *model-driven engineering* (MDE) tool called *Event QoS Aspect Language* (EQAL), this approach further simplifies the deployment and configuration of real-time publish/subscribe services. Chapter IV describes the approach for publish/subscribe service integration, configuration and deployment in detail.

3. D&C Predictability Assurance Technique. To address the challenge of lacking assurance of D&C service, we have developed a technique based on an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). The PARIGE algorithm analyzes the dependency relationships between operational strings, and removes all the dependencies causing priority inversions by promoting components and connections between them from higher priority operational strings to lower priority ones. By applying PARIGE, the D&C framework can avoid deployment priority inversions between operational strings when multiple operational strings need to be deployed at the same time dynamically. Chapter V describes this approach in detail.

I.5 Research Contributions

Our research on deployment and configuration techniques for component-based DRE systems has resulted in an improved D&C middleware platform with better system D&C *productivity* as well as better D&C *predictability* than conventional D&C technologies. The key research contributions of this dissertation are summarized in Table I.1.

The research contributions of this dissertation can be divided into three categories, which are described as follows:

1. **Automated D&C for Component-based DRE Systems** In our work on automated deployment and configuration techniques for QoS-enabled component based DRE systems, we describe how we design and develop the DAnCE D&C framework to separate various real-time QoS concerns from component implementations to promote component reuse. The DAnCE framework enables different real-time systemic concerns to be meta-programmable and declaratively specified through higher-level representation, *i.e.*, XML-based metadata, to simplify real-time QoS configuration and deployment. In addition, the DAnCE framework approach also automatically configures and manages the lifecycle of components and its resources as an integral part of the D&C process at both initial deployment phase and run-time phase.

2. **Integration, Configuration, and Deployment Techniques for Publish/Subscribe Services** In our work on provisioning real-time publish/subscribe services for component-based DRE systems, we developed a novel approach to integrate real-time publish/subscribe services within component middleware to address various publish/-subscribe service provisioning challenges. Our approach allows different publish/-subscribe services to be plugged interchangeably without sacrificing performance while significantly improving component reuse. Furthermore, to simplify the configuration of publish/subscribe services, we developed a MDE tool called Event QoS Aspect Language (EQAL), which simplifies the D&C of publish/subscribe services by enforcing both syntactic and semantic correctness of service configuration policies. Finally, we empirically demonstrate the benefits of the approach through a representative case study and compare the performance of our approach with conventional object-oriented middleware approaches.

3. **Predictable D&C for Component-based DRE Systems** In our work on ensuring the predictability of deployment and configuration for component-based DRE systems, we describe how D&C predictability of component-based systems can be affected by the complex dependency relationships among components and priorities by analyzing different dependency relationships. We then develop a multi-graph algorithm called "*partial priority inheritance via graph recomposition*" to avoid deployment priority inversion, hence improving D&C predictability. Finally, we empirically demonstrate the benefits and effectiveness of the approach and analyze its performance overhead through a representative DRE system case study.

I.6 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter II describes the research related to our work on deployment and configuration for component-based systems and points out the limitations in existing research; Chapter III describes the DAnCE toolchain, our approach to automate the deployment and configuration process for QoS-enabled component middleware-based DRE systems; Chapter IV describes the challenges and solution for integration, configuration, and deployment of real-time publish/subscribe services for QoS-enabled component-based DRE systems; Chapter V describes how we apply the PARIGE algorithm to the DAnCE tool chain to avoid priority inversion when deploying component-based DRE systems; and Chapter VI provides a summary of the research contributions, presents concluding remarks and outlines future research work.

Table I.1: Summary Of Research Contributions

Category	Benefits
Automated Deployment and Configuration Techniques for QoS-enabled Component Middleware	1. Describes how to separate real-time QoS concern from core component business logic to promote component reuse. 2. Enables these systemic concerns to be meta-programmable and declaratively specified through higher-level representation (*i.e.*, XML-based meta-data) to simplify the QoS configuration. 3. Automatically configures and manages the lifecycle of components and its resources as an integral part of the D&C process at both initial deployment phase and run-time phase.
Integration, Configuration, and Deployment Techniques for Publish/Subscribe Services	1. Evaluates the benefits and limitations of different architectural choices for integrating different publish/-subscribe services into component middleware. 2. Develops a novel pattern-oriented approach to integrate publish/subscribe services within component middleware to allow different publish/subscribe services to be plugged in for component-based DRE systems. 3. Designed a MDE-based design tool called Event QoS Aspect Language (EQAL) to simplify the D&C of different publish/subscribe services by enforce the syntactic and semantic correctness of the configuration. 4. Empirically demonstrates the benefits of the approach and its performance overhead.
Predictable D&C Service Techniques for Component-based DRE Systems	1. Describe how D&C predictability of component-based systems can be affected by their dependency relationships and priorities by analyzing different dependency relationships. 2. Develops a solution approach based on a multi-graph algorithm called PARIGE to improve D&C predictability. 3. Empirically demonstrates the benefits of the approach and its performance overhead through a representative DRE system case study.

CHAPTER II

EVALUATION OF ALTERNATE APPROACHES TO DEPLOYMENT AND CONFIGURATION

This section summarizes alternate approaches to D&C for component-based DRE systems. The goal in this chapter is to survey the existing approaches and present unresolved challenges. The subsequent chapters of this dissertation describe how the unresolved challenges are addressed.

II.1 QoS-enabled Component Deployment and Configuration

As component middleware becomes more pervasive, there has been an increase in research on technologies, platforms, and tools for deploying components effectively within distributed systems. This section describes related work in this field and presents unresolved challenges.

II.1.1 QoS-enabled Component Deployment and Configuration: Alternative Approaches

To structure the discussion, a taxonomy, *i.e.*, classification, is presented that categorizes related research across the following two dimensions shown in Figure II.1:

- **QoS Provisioning Time**, which determines when QoS provisioning techniques are applied, *i.e.*, system design/development-time, deployment-time or run-time.
- **Abstraction Level**, which determines at which abstraction level QoS provisioning techniques are applied, *i.e.*, programming language level, DOC middleware level or component middleware level.

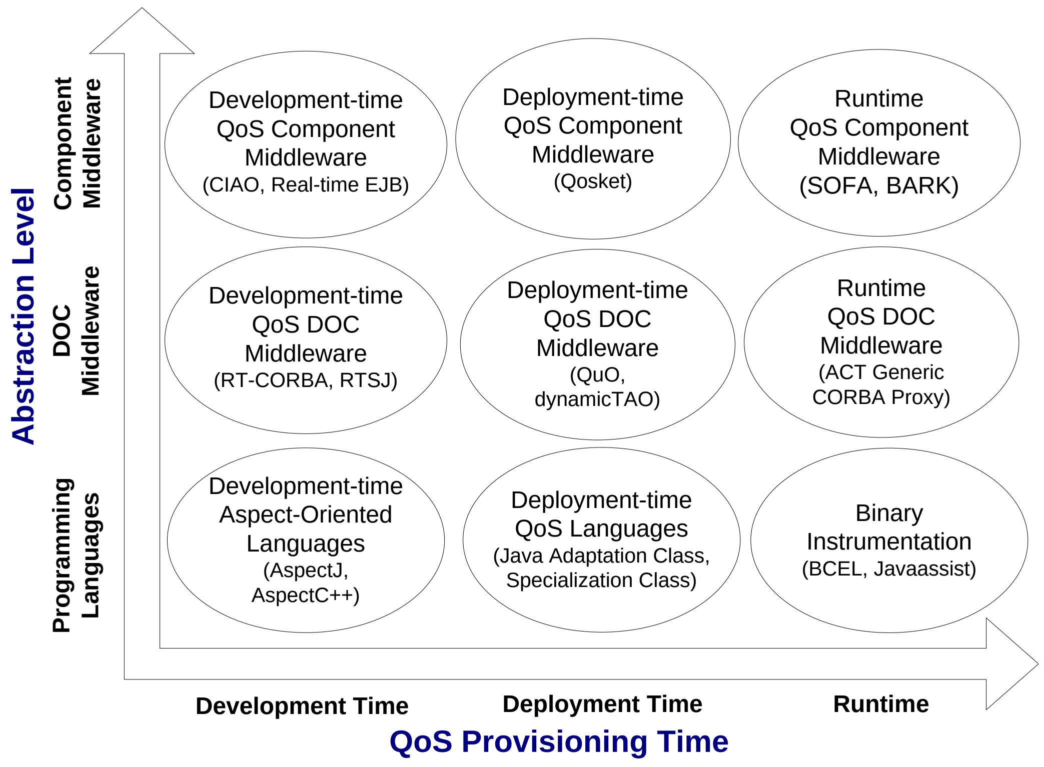

Abstraction Level
Component Middleware
DOC Middleware
Programming Languages
Development-time QoS Component Middleware (CIAO, Real-time EJB)
Deployment-time QoS Component Middleware (Qosket)
Runtime QoS Component Middleware (SOFA, BARK)
Development-time QoS DOC Middleware (RT-CORBA, RTSJ)
Deployment-time QoS DOC Middleware (QuO, dynamicTAO)
Runtime QoS DOC Middleware (ACT Generic CORBA Proxy)
Development-time Aspect-Oriented Languages (AspectJ, AspectC++)
Deployment-time QoS Languages (Java Adaptation Class, Specialization Class)
Binary Instrumentation (BCEL, Javaassist)
Development Time
Deployment Time
Runtime
QoS Provisioning Time

Figure II.1: Research Taxonomy and Research Evolution

Since this dissertation is focused on component-based systems, this section explores a representative sample of the research that has been applied to deploy and configure component-based DRE systems based on the dimension of QoS provisioning time, which can be summarized as follows:

1. **Design/development-time approaches**.

   Design-time QoS provisioning techniques allow system QoS settings to be bound statically at system development time only. Separation of concerns [22, 23, 39] have long been adopted in software engineering community to simplify software development and maintenance, while promoting software reuse. Researchers have been successfully using various separation of concerns techniques to enable the composition of QoS aspects into the application code by leveraging advanced programming language features. The key idea of development-time QoS provisioning techniques is to apply *aspect-oriented programming* (AOP) techniques [48] to weave QoS cross-cutting concerns into application business logic. Some AOP-enabled programming languages include AspectJ [57] and AspectC++ [116]. AOP techniques result in more modular code, lower development costs, and better real-time predictability. The methodology and benefits of applying AOP to address non-functional requirements such as distribution, real-time, and fault tolerance are also documented [27].

Design/development-time QoS provisioning approaches do not provide any dynamic capabilities because applications cannot modify its QoS settings once the initial settings are bound to the application. Therefore, they are mainly suited for *closed* DRE systems but not *open* DRE systems.

2. **Deployment-time approaches**.

   Deployment-time QoS provisioning techniques allow system QoS settings and changing rules to be bounded at deployment time in response to changes in its operating environment. Open DRE systems built upon deployment-time QoS provisioning techniques are self-contained and not able to support the addition of new application behaviors and changing rules once systems are deployed. With the increasing popularity of component-based software development, some deployment-time QoS provisioning techniques have been applied to component middleware.

   Researchers from BBN Technologies [66, 101] proposed to package QoS aspects into components known as "qoskets" for reuse because the limitations of the current implementations of adaptive QoS behaviors complicate their insertion into common application contexts and restrict reusability across applications. FCS/nORB [64] and ControlWare [125] enable real-time behavior to be controlled by adopting feedback control theory. Both Open ORB [8, 31] and dynamicTAO [49, 50] use reflective techniques in middleware to provide a greater degree of configurability and dynamic adaptability.

The key idea of deployment-time QoS provisioning techniques is to allow adaptation rules to be specified explicitly and allow these adaptation rules to be composed into the application through generative tools. Some of the adaptation rules are in the form of declarative QoS contracts, while others are in the form of control algorithms or dynamic scheduling mechanisms.

3. **Run-time approaches**. Run-time QoS provisioning techniques could allow both new application behaviors and adaptation rules to be introduced during run-time, even after systems are deployed. Generally speaking, run-time QoS provisioning supports the most powerful dynamic capabilities, but the system execution environments are also much more complex.

   Dynamic Aspect-Oriented Programming (Dynamic AOP) has been identified as one technique to address run-time QoS provisioning problem [6], focusing in particular on promoting separations of concerns where dynamic aspects involve plugging and unplugging aspects without stopping, and restarting a running system.

   PROSE [90] is a just-in-time weaver of aspects which allows to weave an application at run-time. Pointcuts and aspect advice are defined in an aspect class. Such an aspect can be woven at run-time without previously woven joinpoints. By using the debugger interface, the joinpoints can be set as breakpoints. When a breakpoint is reached, the PROSE engine will execute the appropriate aspect advice. JAsCo [123] is an aspect-oriented programming language originally tailored for component-based systems, in particular the EJB component model. This technique allows aspects to be defined for Java Beans. Other related research [13, 16, 41, 43, 58, 89] also rely on dynamic aspects to introduce QoS provisioning behavior at run-time.

From a system lifecycle perspective, decisions for provisioning system QoS for DRE systems can be made at multiple phases, *i.e.*, at design/development time, deployment time, or run-time. The run-time requirements are the most challenging since they occur dynamically and thus have the shortest time scales for decision-making. Moreover, open DRE system integrators and researchers have the least experience with developing, validating, and optimizing appropriate run-time solutions. Section II.1.2 describes the key unresolved challenges in related research that forms the basis for our research.

II.1.2 QoS-enabled Component Deployment and Configuration: Unresolved Challenges

The majority of existing approaches of QoS provisioning for component-based open DRE systems rely on dynamic AOP techniques, but unfortunately these approaches are all programming language dependent. Open DRE systems typically involve system-wide heterogeneity, *e.g.*, platform and language heterogeneity. Therefore, it is often not feasible to realize an aspect as a simple piece of code to be inserted always in the same fashion. Instead, such QoS concerns must be realized differently in different parts of the system, depending upon platform, language and the dynamics of program execution.

The following is a list of the unresolved challenges with QoS provisioning for component-based DRE systems:

1. **Eliminating complexities for deploying and configuring component server resources to meet real-time requirements.** Conventional middleware D&C platforms are manual and *ad hoc*, which makes them poorly suited for assembling DRE systems from pre-existing COTS components that must meet QoS requirements based on a specific deployment environment. In other words, pre-existing COTS components are not able to have different real-time behaviors based on different system computing resources or mission requirements. It is therefore hard to reuse pre-existing components for DRE systems, and the onus is on humans to manage such concerns manually via *ad hoc* techniques, such as modifying component source code,

which impedes productivity and quality. Furthermore, existing QoS-enabled middleware technologies and D&C technologies lack dynamic reconfiguration capabilities so they are not suitable for open DRE systems. The challenge ahead is to design and implement some form and degree of run-time reconfiguration capabilities while simultaneously ensuring system QoS.

2. **Lack of capability to activate, passivate, and deactivate component assemblies at run-time.** To manage shared resources in a DRE system effectively, components in an assembly need to be activated to become functional, passivated when they will not be accessed for an extended period of time, and deactivated when they are no longer needed. A key challenge is to coordinate these operations in a complete assembly, rather than in an individual component or node. For example, components in an assembly that collaborate by sending messages or events must be *preactivated* to configure the necessary environment and resources so that messages are exchanged in the intended fashion. In particular, all collaborating components in an assembly must be preactivated before any component is activated. Similarly, all collaborating components need to be passivated before any component is deactivated so that no component tries to communicate after its recipient has been deactivated.

   Chapter III describes the design and implementation of the DAnCE D&C framework in detail, which addresses the above challenges.

II.2 Deployment and Configuration of Publish/Subscribe Services

This section surveys literature on some available real-time publish/subscribe systems, both standards-based and proprietary, concentrating on the abstraction layers these publish/-subscribe mechanisms are based on and the QoS capabilities they support. Some of the prior work provides real-time QoS support for DRE systems. However, their QoS assurance mechanisms are based on the traditional object-oriented middleware layer, which hinders system reusability and maintainability and makes complex DRE systems hard to develop. On the other hand, some other related research takes advantage of the strengths of component middleware, but they are not yet suitable for DRE systems due to the limited real-time QoS support, which is the focus of our work.

II.2.1 Deployment and Configuration of Publish/Subscribe Services: Alternate Approaches

We first describe alternative approaches in standards-based publish/subscribe architectures, then present alternative approaches in proprietary publish/subscribe architectures.

II.2.1.1 Standards-based Publish/Subscribe Architectures for DRE Systems

The OMG Data Distribution Service. The OMG Data Distribution Service (DDS) specification [73] is a standard for QoS-enabled publish/subscribe communication aiming at mission-critical DRE systems. It is designed to provide (1) *location independence* via anonymous publish/subscribe protocols that enable communication between collocated or remote publishers and subscribers, (2) *scalability* by supporting large numbers of topics, data readers, and data writers, and (3) *platform portability and interoperability* via standard interfaces and transport protocols. Multiple implementations of DDS are now available ranging from high-end COTS products to open-source community-supported projects, such as OpenSplice [91] and TAO DDS [84]. The OMG DDS standard is based on object-oriented middleware rather than component middleware.

The CORBA Distributed Notification Service. The OMG has also issued a specification to build distributed versions of the Notification Service via its "Management of Event Domains Specification" [76]. This document describes how multiple instances of the Notification Service can be interconnected to avoid the excessive overhead and eliminate the single point of failure represented by the event channel object. This specification, however, does not incorporate any mechanisms to reduce event delivery based on filters. Similar to DDS, both the CORBA Notification Service and the CORBA Distributed Notification Service are based on object-oriented middleware rather than component middleware, and are a target of our integration approach.

The Java Message Service. The Java Message Service (JMS) [69] is a messaging standard that allows Java applications based on the J2EE standard to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous. JMS supports both object-oriented and component-based mechanisms for client connections and QoS configurations (such as transaction, fault tolerance and persistency). The object-oriented mechanism allows pure Java clients to play the role of event publisher or event subscribe and the component-based mechanism allows Java Message-Driven Beans (MDBs) to be activated through a metadata-based mechanism called *activation specification*. The richness of delivery semantics of JMS makes it a powerful pub/sub mechanism for enterprise business applications. However, since JMS incurs much larger memory footprint and only provides limited real-time QoS support, it is not yet suitable for DRE systems. Moreover, JMS-based J2EE applications are typically bundled with large, complex scripts to handle the configuration of their messaging infrastructure because there is no deployment and configuration tools that could automatically handle the services among a large number of application components.

II.2.1.2 Proprietary Publish/Subscribe Architectures for DRE Systems

Cadena Event Channel Framework. Cadena Event Communication Framework [115] includes a CORBA-based event channel which has been integrated into the OpenCCM [122] component middleware infrastructure. The framework implements a number of features of the event service middleware, such as event filtering and event correlation. Although this work comes close to our work, however, it does not address the problems such as event channel federation, real-time event scheduling and dispatching and periodic event processing, which are crucial for a number of mission-critical real-time applications. Our work, on the other hand, leverages the real-time event channels and QoS-enabled component middleware to provide the properties outlined above.

SIENA. SIENA [11, 12] is a notification service architecture for Internet-scale event distribution. The architecture is based on *content-based networking*, where a network of routers propagate packets based not on a specific destination address, but on the contents of the packet. The authors propose using an event format similar to the CORBA Notification Service, *i.e.*, a sequence of $(name, value)$ tuples. Using this format, consumers use a boolean predicate on the tuple values to describe the set of events they are interested in. The authors describe algorithms to reduce the use of network resources. For example, the authors propagate filtering information as close to the sources as possible; likewise, filtering constraints are combined and simplified to minimize the use of computation resources in the routers.

ECO. The Distributed Systems Group at Trinity College, Dublin has developed ECO ("Events, Constraints and Objects") [117]. The authors propose building programs out of cooperating objects that publish or subscribe to events as needed. Filtering, concurrency and timeliness constraints are expressed as constraints on the events that a particular object publishes or subscribes.

The authors propose extending general-purpose programming languages, such as C++ or Java, to include explicit declarations for events, as well as the types of events that a class can subscribe or publish. Naturally, this static publication or subscription can be further restricted at run-time. The authors propose using new language statements for this purpose.

Objects can add *Notify* constraints to limit the objects that they subscribe to. These conditions are evaluated at the source of the event and thus are limited to constraints on the event parameters or the source identity. Objects can also define *Pre* and *Post* constraints, which are evaluated on the destination object and can use the state of the receiving object to affect the event processing. *Pre* constraints are evaluated before the event is delivered and can determine if the event is dropped, enqueued or processed immediately.

CMU Real-time Publish/Subscribe. Rajkumar, *et al.*, describe a real-time publish/subscribe prototype developed at CMU/SEI [94]. Their Publish/Subscribe model is functionally similar to the CORBA RTES, though it defines its own programming APIs and communication protocols. The authors detail how real-time threads and adequate synchronization primitives can be used to implement the RT publish/subscribe model without undue priority inversions. However, the authors do not consider the fact that adequate synchronization primitives are a necessary condition to address unbounded priority inversions, but it is not a sufficient condition.

Section II.2.2 describes the key unresolved challenges in related research that forms the basis for our research.

II.2.2 Deployment and Configuration of Publish/Subscribe Services: Unresolved Challenges

Much has been written about real-time publish/subscribe systems, but little effort has been expended in documenting the patterns, optimizations and architectures required to design and implement QoS-enabled publish/subscribe models in component-based software architectures. Also, there is very little or no empirical evidence to support the performance and predictability claims of several of these systems when used in component-based systems, even when research concentrates on real-time applications.

The following is a list of key unresolved challenges with respect to deploying and configuring real-time publish/subscribe services when using standards-based component middleware:

1. **Integration Challenges.** Traditional object-oriented middleware (such as CORBA 2.x) provides DRE systems with access to various publish/subscribe middleware services (such as CORBA Event Service [75] and Real-time Event Service [35])

through the underlying Object Request Broker (ORB) and Portable Object Adapter (POA) [82]. Component-based middleware, such as Lightweight CORBA Component Model (CCM) [72] enables (1) reusability of components by implementing only application logic and (2) easier integration into different applications and run-time contexts. Component deployers thus need to support the integration of common middleware services into component-based systems for which no standard mechanisms yet exist. Although directly using object-oriented real-time publish/subscribe services is viable for small-scale DRE systems with a couple of components, however, there are enormous amount of complexities involved to develop large-scale, complex DRE systems, which might consist of hundreds or even thousands of components.

2. **Configuration Challenges.** For large-scale, complex DRE system which are composed of many components developed by different component vendors, there is no mechanism to centralize the QoS management end-to-end for the entire DRE system, since each component developed by a particular vendor only has a "local" view of the entire system. Furthermore, most publish/subscribe services based on DOC middleware (including the CORBA Event and Notification Services) do not validate QoS specifications automatically. It is hard, moreover, to *manually* validate QoS configurations for semantic compatibility. This process is particularly daunting for large-scale, mission-/safety-critical DRE systems, where the cost of human error is most egregious.

3. **Deployment Challenges.** Since different components are developed by different parties, to deploy these components into a common publish/subscribe architecture is usually very hard since there is no standards-based deployment tools that can understand all different configurations of all different components. As a result, DRE system deployers must custom-build their deployment and configuration solutions to marriage different component configurations together. Such customized solutions

are often tightly coupled with particular deployment settings and hard to evolve, because a lot of configurable options are only available at the component middleware infrastructure level, *i.e.*, container and component server level.

Chapter IV describes how we apply pattern-based system design and MDE techniques to address the above challenges.

II.3 QoS Assurance of D&C Service

As component middleware becomes more pervasive, there has been an increase in research on technologies, platforms, and tools to ensure components are deployed correctly, effectively and efficiently. This section describes related work in this field then presents unresolved challenges.

II.3.1 QoS Assurance of D&C Services: Alternative Approaches

We classify the related work into two categories. The first category includes formal models using various formal method techniques to ensure consistency and dependency correctness issues of software deployment. The second category includes related works that aims to provide architectural support to address various QoS issues of software deployment.

II.3.1.1 Dependency Management Approaches

Much prior research has been conducted on dependency management of software components. Most literature in this area analyzes deployment installability via formal models that explicitly express dependencies of software components or software packages. For example, [5] present a model to formalize deployment dependencies of software components. These dependencies are expressed in a logical language associated with a D&C framework that allows proving properties (such as whether the dependency is mandatory, optional, or negative) of the deployment plan.

Liu, *et al.* [62] define an *application buildbox* as a software deployed environment and defines a formal Labeled Transition System (LTS) on the buildbox with transitions for deployment operations that include build, install, ship, and update. Formal properties of the LTS are introduced, including version dependency and software component dependencies. This prior work, however, does not address deployment predictability, which makes it unsuited for DRE systems.

Rigole, *et al.* [97] present a strategy for deploying component-based systems incrementally to match the functionality of pervasive computing applications. This deployment strategy is accomplished by linking component composition models with task models at design-time, from which a run-time deployment plan can be deduced.

Some software frameworks have been developed to support software component deployment or redeployment. For example, Chen, *et al.* [13] and Kon, *et al.* [51] use framework-guided reconfiguration mechanisms for component-based distributed systems. These frameworks offer mechanisms to analyze dependencies between peer components to deal with reconfiguration consistency. However, this work does not consider how to improve the predictability of the deployment process, but instead focuses only on consistency.

II.3.1.2 Deployment Planning Approaches

Various component deployment techniques have been proposed to improve system QoS which are centered around a deployment planning algorithm. Typically, a deployment planning technique requires a goal to produce a deployment plan.

Some planning algorithms aim to optimize system resource usages, such as the approaches proposed in [38, 46]. Other approaches, such as [47], aim to resolve multiple constraints at the same time, such as constraints resulting from application semantic requirements, network resource limitations, and interactions between the two.

Some planning techniques can be applied not only at initial deployment phase, but also at run-time for redeployment. For example, Planit [1] manages the deployment and reconfiguration of a software system through a temporal planner. Given a model of the structure of a software system, the network upon which the system should be hosted, and a goal configuration, Planit can use the temporal planner to devise possible deployments of the system. Ivan, *et al.* [40] provide run-time support for dynamic component deployment in conjunction with planning policies, which steer the deployment to accommodate underlying running environment characteristics.

II.3.1.3 Architectural Support Approaches

Various architectural support approaches have been proposed to ensure different QoS aspects of D&C for component-based systems.

The OpenCCM (`corbaweb.lifl.fr/OpenCCM/`) Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. The OpenCCM DCI allows systems to be configured through XML-based metadata. Similarly, [93] proposes using an architecture descriptive language (ADL) that allows assembly-level activation of components and describes assembly hierarchically. The asynchronous and hierarchical techniques in this approach can enhance the parallelism among different nodes to enhance D&C performance.

The work reported in [100] proposes the use of the Globus Toolkit [26] to deploy CCM components on a computational grid to take advantage of high bandwidth wide-area networks. The work reported in [95] relies on pre-allocating resource (*e.g.*, pre-load components in .NET dynamic reconfiguration framework) to make the D&C process of components more predictable. However, this approach only concerns improving the response time within a single deployment unit, rather than of managing multiple deployment units at the same time.

The approach described in [56] uses a virtual machine (VM) technique that provides automated D&C of flexible VMs that can be configured to meet application needs and then subsequently cloned and dynamically instantiated to improve the predictability of deployments. This approach supports a graph-based model for the definition of customized VM configuration actions.

Subramonian, *et al.* [118] propose a technique for static component configuration and deployment, which enhances configurability by avoiding features that are not supported by key real-time platforms, while reducing run-time overhead and footprint. However, this technique must trade off component deployment flexility for performance, therefore it is not suitable for open DRE systems.

II.3.2 QoS Assurance of D&C Services: Unresolved Challenges

All the related work in the dimension of dependency management does not take the predictability issue into account, which makes them unsuited for DRE systems. Likewise, all the related work in QoS assurance for D&C dimension does not address the issue of D&C predictability but focusing on improving D&C performance.

The following is a list of key unresolved challenges with respect to ensuring the predictability of deployment and configuration of DRE systems:

1. **Avoid deployment priority inversion and its propagation effect.** For large-scale DRE systems, components can have very complex dependency relationships among each other, which can conflict with real-time QoS configuration of components, such as priority assignments. The conflict between component deployment priorities and their dependencies can cause deployment priority inversions because components with lower priority must be deployed before a higher priority component if the higher priority has a dependency to it. Moreover, when a circular dependency exists among operational strings, the central coordinated phased deployment technique will fail because the deployment of each operational string is a primitive task and cannot be interleaved with each other.

2. **Lack of mechanism to improve the overall utility of DRE systems while components are being deployed.** The dynamic nature of open DRE systems requires on-demand deployment of a number of components which cooperate with each other to ensure DRE systems are kept synchronized with changing mission goals or environmental changes. Since a number of components with different importance to the entire DRE system need to be deployed at the same time, the goal of D&C mechanisms is therefore to deploy these components in an effective way to improve the overall QoS of DRE systems in two different aspects.

   Chapter V describes how we apply the PARIGE algorithm to improve D&C predictability in detail.

II.4 Summary

This chapter provided a survey of work related to the research described in this dissertation. Chapter III describes in detail how the QoS provisioning capabilities of a QoS-enabled deployment and configuration engine resolves these limitations. Chapter IV describes the challenges with integration, configuration and deployment of publish/subscribe services in component-based systems in detail and explains how they are addressed by this dissertation. Chapter V describes how we address the predictability of D&C for large-scale DRE systems when a large number of components must be deployed at run-time.

CHAPTER III

TECHNIQUES FOR QOS-ENABLED COMPONENT DEPLOYMENT AND CONFIGURATION

In large-scale distributed real-time and embedded (DRE) systems, component middleware features can help make the software more flexible by separating application functionality from system lifecycle activities, such as component configuration and deployment. Conventional component middleware platforms, such as J2EE and .NET, is not well-suited for these types of DRE systems since they do not provide real-time QoS support. *QoS-enabled component middleware*, such as Component Integrated ACE ORB (CIAO) [124], Qedo [98], and PRiSm [114], have been developed to address these limitations by combining the flexibility of component middleware with the predictability of Real-Time CORBA.

QoS-enabled component middleware, however, also introduces new complexities that stem from the need to (1) deploy component assemblies into the appropriate DRE system target nodes, (2) activate and deactivate component assemblies automatically, (3) initialize and configure component server resources to enforce end-to-end QoS requirements of component assemblies, and (4) simplify the configuration, deployment, and management of common services used by applications and middleware. The lack of portable, reusable, and standard mechanisms to address these challenges is hindering the adoption of component middleware technologies for DRE systems.

To meet these challenges, we have developed the *Deployment and Configuration Engine* (DAnCE) (`www.dre.vanderbilt.edu/CIAO`), which is an open-source QoS-enabled middleware framework compliant with the OMG Deployment and Configuration specification [79] that enables the deployment of DRE system component assemblies by addressing various QoS-related concerns, such as collocation, memory constraints, and processor loading. The deployment and configuration of components in DAnCE, therefore, involves mapping known variations in the *application requirements space* (such as variations in QoS requirements) to known variations in the *software solution space* (such as configuring the underlying network, OS, middleware, and application parameters to satisfy the end-to-end QoS requirements).

To support effective deployment and configuration of component-based DRE systems, the key capabilities provided by DAnCE include:

- One-time parsing and storing of component configuration and deployment descriptions (which are represented as metadata in XML format) so that run-time parsing overhead is not incurred during component deployment.
- Automatic downloading of component packages so that the implementations can be changed seamlessly as components migrate from one node to another, even in a heterogeneous target domains.
- Automatic configuration of object request brokers (ORBs), containers, and component servers to (1) meet the desired QoS requirements and (2) reduce human operator mistakes introduced while configuring middleware and application components.
- Automatic connection[1] of component ports so that developers need not be concerned with these low-level details.

III.1 Deployment and Configuration Challenges in Component-based DRE Systems

To illustrate the deployment and configuration challenges in DRE systems, this section presents a case study of a representative component-based DRE system called the *inventory tracking system* (ITS) [71]. An ITS is a warehouse management infrastructure that monitors

[1]In the context of this dissertation, a *connection* refers to the high-level binding between an object reference and its target component, rather than a lower-level transport (*e.g.*, TCP) connection.

and controls the flow of goods 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). This section first provides an overview of the structure/-functionality of our ITS case study and then uses the case study to describe configuration and deployment challenges.

III.1.1 Overview of ITS

An ITS provides mechanisms for managing the storage and movement of goods in a timely and reliable manner. For example, an ITS should enable human operators to 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 [71], we have developed the ITS shown in Figure III.1 and deployed it using the DAnCE framework.

Figure III.1 shows how our ITS consists of the following three subsystems:

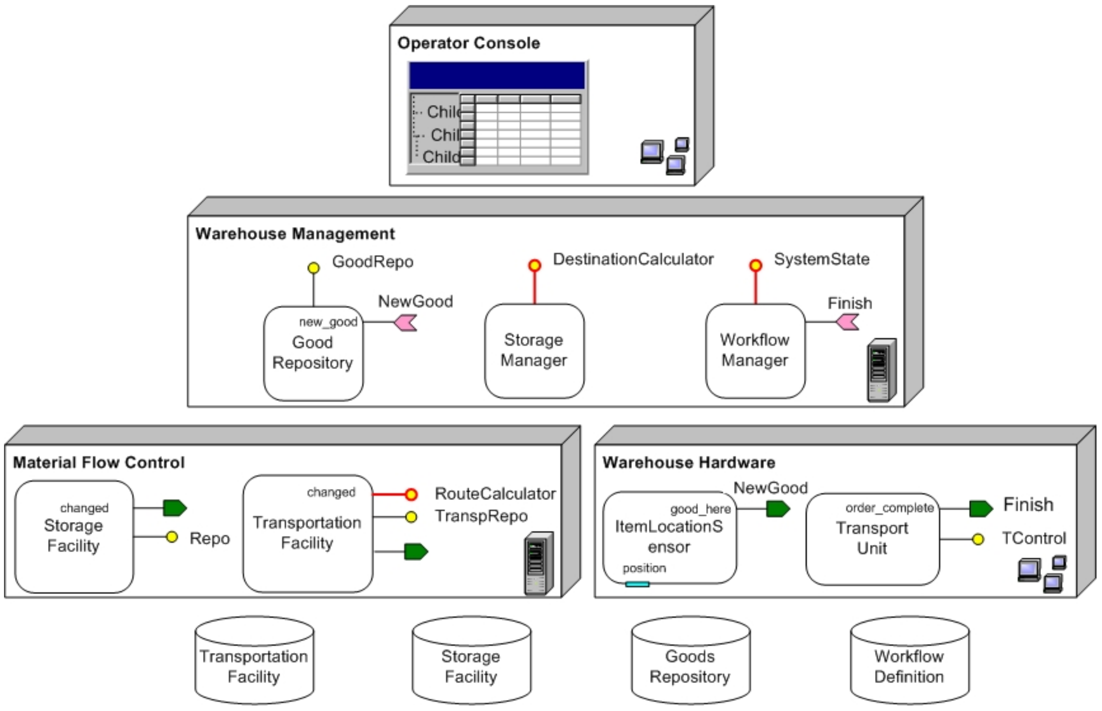


Figure III.1: Key Components in the ITS Case Study

- **Warehouse management**, whose high-level functionality and decision-making components calculate the destination locations of goods and delegate the remaining details to other ITS subsystems.
- **Material flow control**, which handles all the details (such as route calculation and transportation facility reservation) needed to transport goods to their destinations. The primary task of this subsystem is to execute the high-level decisions calculated by the warehouse management subsystem.
- **Warehouse hardware**, which deals with physical devices (such as sensors) and transportation units (such as conveyor belts, forklifts, and cranes).

After the ITS components comprising the ITS subsystems described above are developed, they must be configured and deployed to meet warehouse operating requirements. In our ITS case study, ∼200 components must be deployed into 26 physical nodes in the warehouse. We focus on a portion of this system to motivate key challenges DAnCE faced when deploying and configuring the ITS. Figure III.2 shows a subset of key component interactions in the ITS case study shown in Figure III.1. As shown in this figure, the `WorkflowManager` component of the material flow control subsystem is connected to the conveyor belt and crane transportation units of the warehouse hardware subsystem. We focus on the scenario where the `WorkflowManager` contacts the `ConveyorBelt` and `Crane` components using the `move_item()` operation to move an item from a *source* (such as a loading dock) to a *destination* (such as a warehouse storage location). The `move_item()` operation takes source and destination locations as its input arguments. When the item is moved to its destination successfully, the `ConveyorBelt` and the `Crane` inform the `WorkflowManager` via the `finish_mov()` event operation. `ConveyorBelt` and `Crane` components are also connected to various `ItemLocationSensor` components, which periodically inform the other components of the location of moving items.

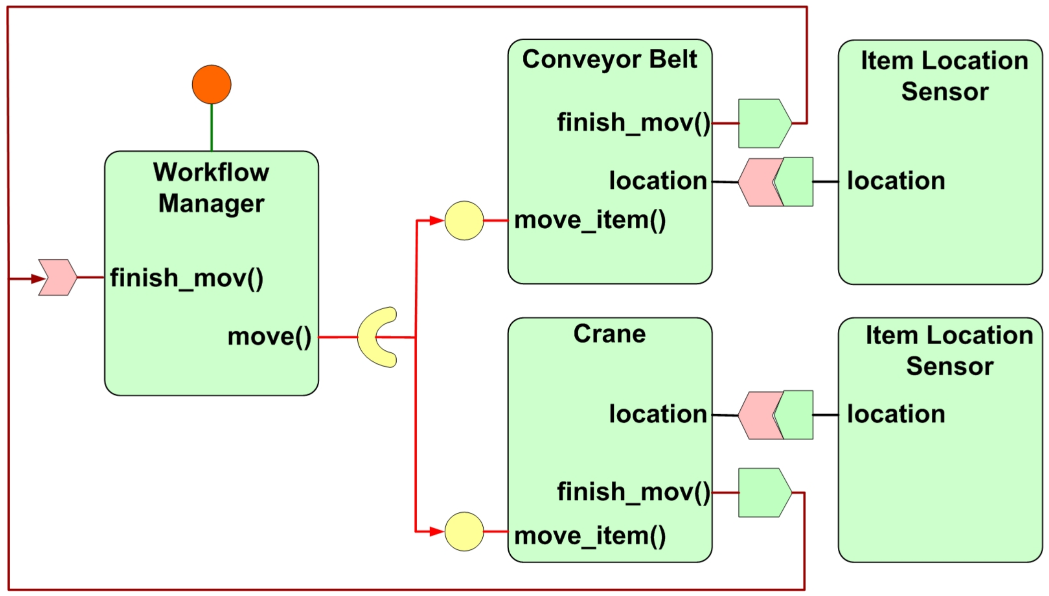


Figure III.2: Component Interactions in the ITS Case Study

III.1.2 Challenges in Configuring and Deploying ITS

Using the ITS case study described in Section III.1.1, we now illustrate the deployment and configuration challenges in component-based DRE systems.

Challenge 1: Efficiently storing and retrieving component implementations. Large-scale DRE systems need capabilities that enable application developers and deployment run-time tools to (1) upload component implementations to storage sites and/or (2) fetch component implementations from storage sites for installation. These capabilities should allow multiple implementations of a component written in different programming languages and run on different OS platforms. Moreover, it should be possible to pre-stage component implementations to avoid downloading selected implementations from central storage sites during the deployment process.

It is conceivable for an ITS `ConveyorBelt` component to have implementations for Linux in Java and Windows in C++, which will require that these implementations be fetched and deployed appropriately on a particular node in a small and bounded amount of time. Section III.2.2.1 describes how DAnCE addresses this challenge.

Challenge 2: Activation, passivation, and deactivation of component assemblies. To manage shared resources in a DRE system effectively, components in an assembly need to be activated to become functional, passivated when they will not be accessed for an extended period of time, and deactivated when they are no longer needed. A key challenge is to coordinate these operations in a complete assembly, rather than in an individual component or node to coordinate different components running on different nodes. For example, components in an assembly that collaborate by sending messages or events must be *preactivated* to configure the necessary environment and resources so that messages are exchanged in the intended fashion. In particular, all collaborating components in an assembly must be preactivated before any component is activated. Similarly, all collaborating components need to be passivated before any component is deactivated so that no component tries to communicate after its recipient has been deactivated.

For instance, when the `ConveyorBelt` component in Figure III.2 is being removed, the `WorkflowManager` component must already be passivated since otherwise it could continue to make `move_item()` invocations on the `ConveyorBelt`. Section III.2.2.2 describes how DAnCE addresses this challenge.

Challenge 3: Configuring NodeApplication component server resources. In large-scale DRE systems, QoS requirements (such as low latency and bounded jitter) are often important considerations during the deployment process since component (re)deployment may occur throughout the lifecycle of a large-scale system. To enforce these QoS requirements, component servers and containers must be configured in accordance with QoS properties, such as those defined in Real-Time CORBA [83]. Component deployment and configuration tools must therefore be able to (1) specify the middleware configurations needed to configure components, containers, and component servers and (2) set the QoS policy options provided by the underlying middleware into semantically consistent configurations.

For instance, in the ITS case study (Figure III.2), whenever a `ConveyorBelt` component's hardware fails, it should notify the `WorkflowManager` in real-time to minimize/-avoid damage. Likewise, ITS `ConveyorBelt` and `Crane` components may need to be collocated with the `WorkflowManager` in some assemblies to minimize latency. Section III.2.2.3 describes how DAnCE addresses this challenge.

III.2 The Design of DAnCE

This section describes the design of the DAnCE D&C framework based on the OMG's Deployment and Configuration (D&C) specification [79]. This specification standardizes many aspects of deployment and configuration for component-based distributed systems, including component configuration, component assembly, component packaging, package configuration, package deployment, and target domain resource management. These aspects are handled via a *data model* and a *run-time model*. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describe component assemblies and their configuration and deployment characteristics. The run-time model defines a set of managers that process the metadata described in the data model during system deployment. This section shows how the design and implementation of DAnCE has been tailored to address the D&C challenges of component-based DRE systems described in Section III.1.2.

III.2.1 The Structure and Functionality of DAnCE

The architecture of the Deployment and Configuration Engine (DAnCE) is shown in Figure III.3. This section describes how DAnCE provides a reusable middleware framework for deploying and configuring components in a distributed target environment, using the ITS case study in Section III.1.1 to motivate its key capabilities. DAnCE is built atop

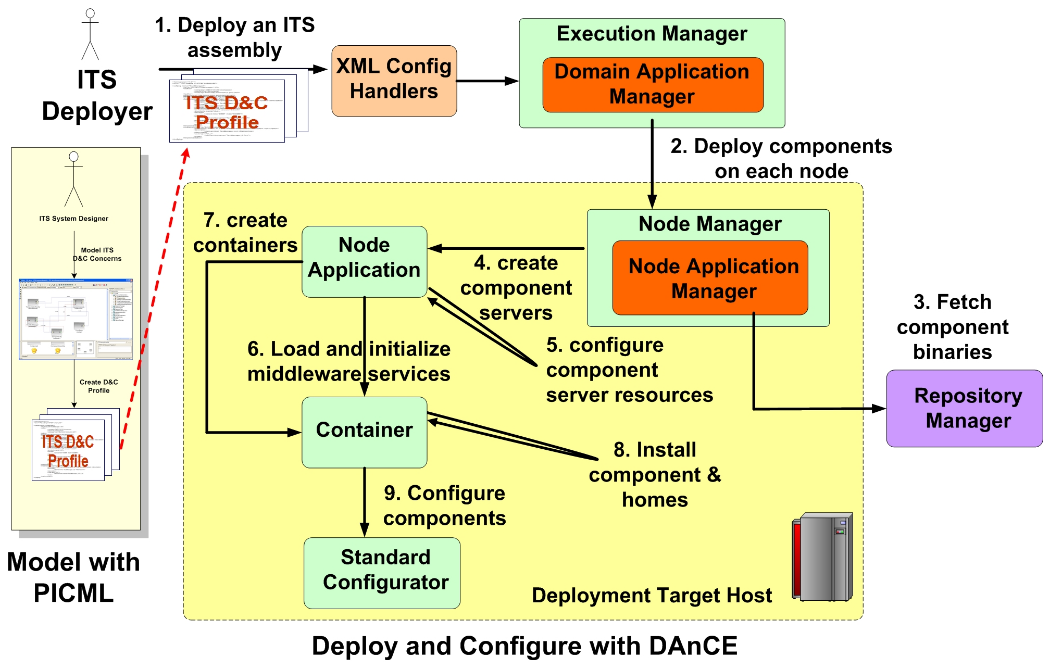


Figure III.3: Overview of DAnCE

The ACE ORB (TAO) [107] and CIAO [124], which makes it portable to most hardware and OS platforms in use today.

As shown in Figure III.3, an ITS deployer creates XML descriptors that convey application deployment and configuration metadata, using external model-driven engineering (MDE) tools [4] such as PICML [3]. PICML is a MDE tool that enables developers to define component interfaces, QoS parameters and software building rules, and also generates deployment and configuration metadata that facilitates system deployment. PIMCL generated metadata is compliant with the data model defined by the OMG D&C specification. To support additional deployment and configuration concerns not addressed by this specification, we enhanced the spec-defined data model by describing additional deployment concerns (such as real-time QoS requirements and middleware service configuration and deployment) discussed in Section III.2.2.

All the metadata to describe these concerns is captured in an XML file called the *deployment plan*, which describes (1) the DRE system component instances to deploy, (2) what properties of these components should be initialized, (3) what QoS policies these components must contain, (4) what middleware services the components use, and (5) how the components are connected to form component assemblies. The various entities of DAnCE shown in Figure III.3 are implemented as CORBA objects[2] that collaborate as follows:

ExecutionManager runs as a daemon and is used to manage the deployment process for one or more domains. In accordance with the D&C specification, DAnCE defines a *domain* as a target environment composed of *nodes*, *interconnects*, *bridges*, and *resources*. An `ExecutionManager` uses the *factory* and *finder* design patterns to manage a set of `DomainApplicationManagers`.

DomainApplicationManager manages the deployment of components within a single domain (to manage multiple domains, an `ExecutionManager` can coordinate with multiple `DomainApplicationManagers`). A `DomainApplicationManager` splits a deployment plan into multiple sub-plans, one for each node in a domain. In DAnCE, the `ExecutionManager` and `DomainApplicationManager` objects reside in the same daemon process to improve deployment performance by leveraging the collocation optimizations provided by TAO. If the target deployment environment has multiple domains, then multiple `DomainApplicationManager` objects will be activated inside the daemon process, one for each domain.

NodeManager runs as a daemon on each node and manages the deployment of all components that reside on that node, irrespective of which application they are associated with. Components are created by containers, which are hosted in component server processes called `NodeApplications`. The `NodeManager` creates the `NodeApplication-Manager`, which in turn creates the `NodeApplication` processes that host containers, thereby enhancing the reuse of components shared between applications on a node.

[2]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.

NodeApplicationManager is collocated with a `NodeManager` to manage the deployment of all components within a `NodeApplication` which is a server process that hosts a group of related components in a particular application. To differentiate deployments in a node, DAnCE's `DomainApplicationManager` uses the node's `NodeManager` to create a `NodeApplicationManager` for each deployment and sends it the metadata it needs to deploy components.

NodeApplication plays the role of a component server process that provisions the computing resources (*e.g.*, CPU, memory and network bandwidth) for the components it hosts. Based on metadata provided by other DAnCE managers in the deployment process, the `NodeApplication` creates the initial containers that provide an environment for creating and instantiating application components. Components in a node are thus deployed in one or more `NodeApplications` in accordance with a deployment plan.

RepositoryManager runs as a daemon dedicated to a domain and is used by (1) deployer agents to store component implementations and (2) DAnCE's `NodeApplicationManager` to fetch necessary component implementations on demand. Each `NodeApplicationManager` uses its `RepositoryManager` to search component implementation binaries (stored in the form of dynamic linking libraries) and fetches them into the local node's storage cache.

III.2.2 Applying DAnCE to Address DRE Systems D&C Challenges

The remainder of this section describes how (1) the DAnCE managers in Figure III.3 address key DRE systems D&C challenges described in Section III.1.2 and (2) our solutions are applied to the ITS case study presented in Section III.1.1.

III.2.2.1 Resolving Challenge 1: Storing and Retrieving Component Implementations via a Repository Manager.

DAnCE's `RepositoryManager` provides efficient mechanisms where applications can (1) store component implementations at any time during the system lifecycle and (2) retrieve different versions of implementations as components are (re)deployed on various types of nodes. As shown in Figure III.4, the `RepositoryManager` can also act as an HTTP client and download component implementations specified as *URLs* in a deployment plan. It caches these implementations in the local host where the `RepositoryManager` runs so they can be retrieved by `NodeApplicationManagers`.

Over a system's lifetime, a component could be migrated and redeployed on a node whose type is different than its earlier host(s), in which case a different component implementation must be provided. To support efficient deployment, DAnCE's `NodeApplic-ationManagers` periodically contact the `RepositoryManager` to download the latest implementations of designated components. When a component is redeployed, therefore, all its implementations can be cached locally on the target nodes, so downloading overhead need not be incurred during the deployment process.

DAnCE's `RepositoryManager` uses ZIP compression and file archiving mechanisms [126] to provide an efficient representation of the contents of a ZIP archive and (de)compress all the implementations in a packaged format. It uses CORBA operation invocations to transfer the ZIP-encoded component assembly packages to the node(s) in a domain that run `NodeApplicationManagers`. In the ITS case study, an initial deployment might write the `ConveyorBelt` component in Java and host the component on an Embedded Linux node. As the system runs, ITS developers could create a C++-based Win32 implementation of `ConveyorBelt` and submit it to DAnCE's `Repository-Manager`. At some point during the ITS lifecycle, the `ConveyorBelt` could be stopped at the current Linux node and moved to a Windows node. To execute that deployment request, DAnCE's `NodeApplicationManager` running on the Windows node could

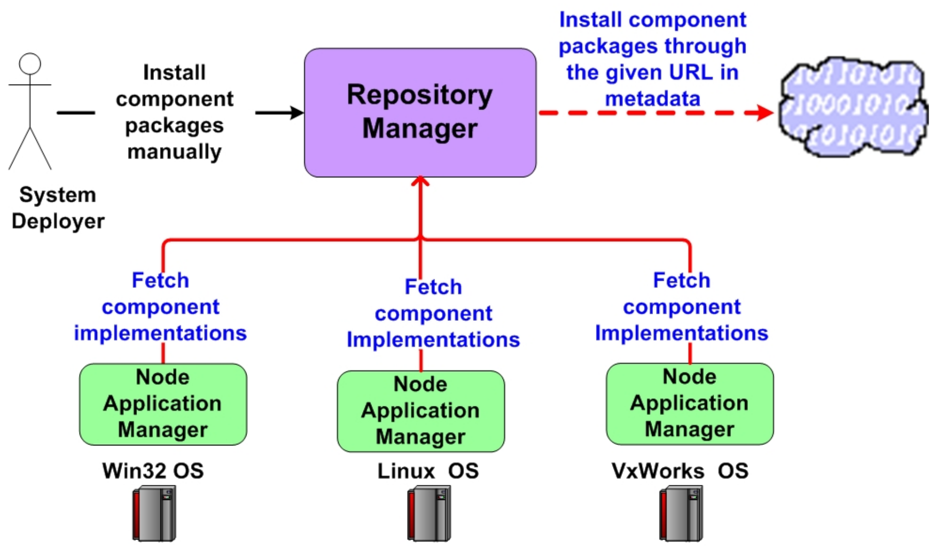

Install component packages manually
Repository Manager
Install component packages through the given URL in metadata
System Deployer
Fetch component implementations
Fetch component Implementations
Fetch component Implementations
Node Application Manager
Node Application Manager
Node Application Manager
Win32 OS
Linux OS
VxWorks OS

Figure III.4: Downloading implementations using the Repository Manager

contact the `RepositoryManager` to retrieve the Windows implementation of the `ConveyorBelt` component and deploy it. The `RepositoryManager` thus helps decouple when and how ITS component implementations are developed from when they are deployed.

III.2.2.2 Resolving Challenge 2: Using the DomainApplicationManager to Coordinate the Component Assembly Lifecycle.

The OMG D&C standard only defines how the initial D&C of a component assembly should be performed and how it can be undeployed, but does not define how the lifecycles of components should be managed. Therefore, during the lifecycle of the component assembly, DAnCE's `DomainApplicationManager` maintains PREACTIVE, ACTIVE, PASSIVE, and DEACTIVATED run-time state information on each component in the component assembly, as shown in Figure III.5. The PREACTIVE state indicates that the component has been created and has been provided its environment settings. The ACTIVE state indicates that the component has been activated with the underlying middleware. The PASSIVE state indicates that the component is idle and all its resources can be used by other components. The DEACTIVATED state means that the component has been deactivated and removed from the system. During the deployment process, DAnCE's `DomainApplic-`

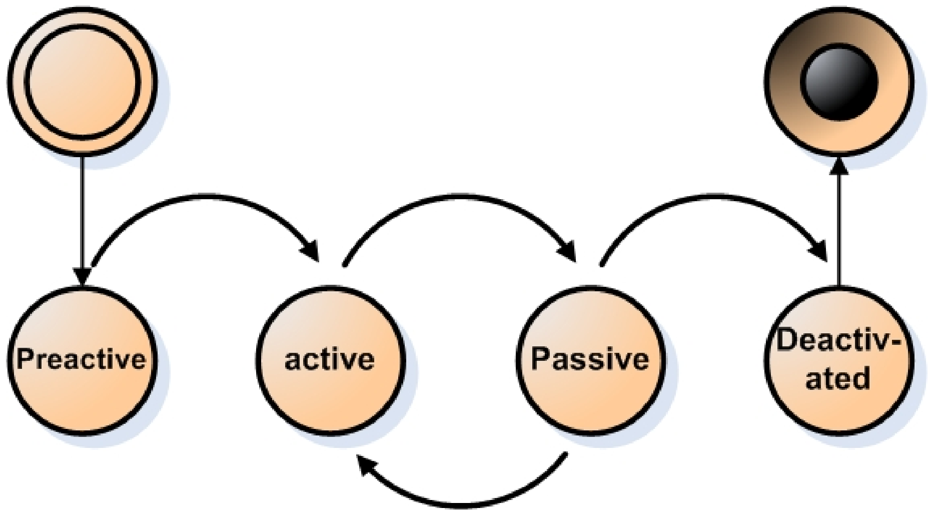


Figure III.5: Different States of a Component

`ationManager` ensures that components are not connected and activated until all the components are in the *preactive* state. Similarly, during assembly deactivation, DAnCE's `DomainApplicationManager` ensures that components in an assembly are deactivated only when all the components are in the *passive* state.

To ensure that a component's ongoing invocations are processed completely before it is passivated, all operation invocations on a component in CIAO are dispatched by the standard Lightweight CCM Portable Object Adapter (POA), which maintains a *dispatching table* that tracks how many requests are being processed by each component in a thread. CIAO uses standard POA reference counting and deactivation mechanisms [61] to keep track of the number of clients making invocations on a component. After a server thread finishes processing the invocation, it decrements the reference count in the dispatching table. Only when the count is zero, is the component passivated. CIAO therefore ensures that the system is always in a consistent state to ensure that no invocations are lost. To prevent new invocations from arriving at the component while it is being passivated, the container blocks new invocations for this component in the server ORB using standard CORBA portable interceptors [70].

In the ITS case study, DAnCE's `DomainApplicationManager` ensures that the `ItemLocationSensor` components do not make operation invocations on the `ConveyorBelt` components unless both are active. Similarly, during the deactivation of the `ConveyorBelt` component, the `DomainApplicationManager` ensures that `WorkflowManager` components are passivated, which ensures that all `move_item()` requests are handled properly. Finally, the `ConveyorBelt` component's POA ensures that all requests being processed by the component are dispatched before deactivating the component.

III.2.2.3 Resolving Challenge 3: Configuring NodeApplication Component Server Resources.

To enforce QoS requirements, DAnCE extends the OMG D&C [79] specification to define `NodeApplication` server resource configurations, which heavily influence end-to-end QoS behavior. Figure III.6 shows the different categories of server configurations that can be specified using the DAnCE *server resources XML schema*, which are related to system end-to-end QoS enforcement. In particular, each server resource specification can set the following options: (1) *ORB command-line options*, which control TAO's connection management models, protocol selection, and optimized request processing, (2) *ORB service configuration option*, 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.

As described in Section III.2.1, components are hosted in containers created by the *NodeApplication* process, which provides the run-time environment and resources for components to execute and communicate with other components in a component assembly.

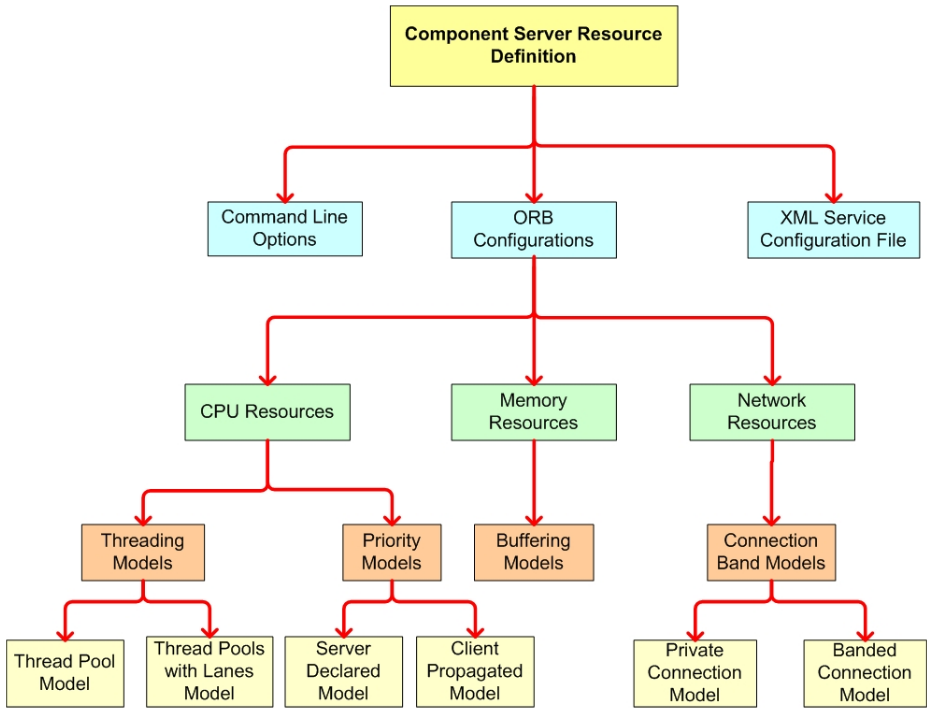
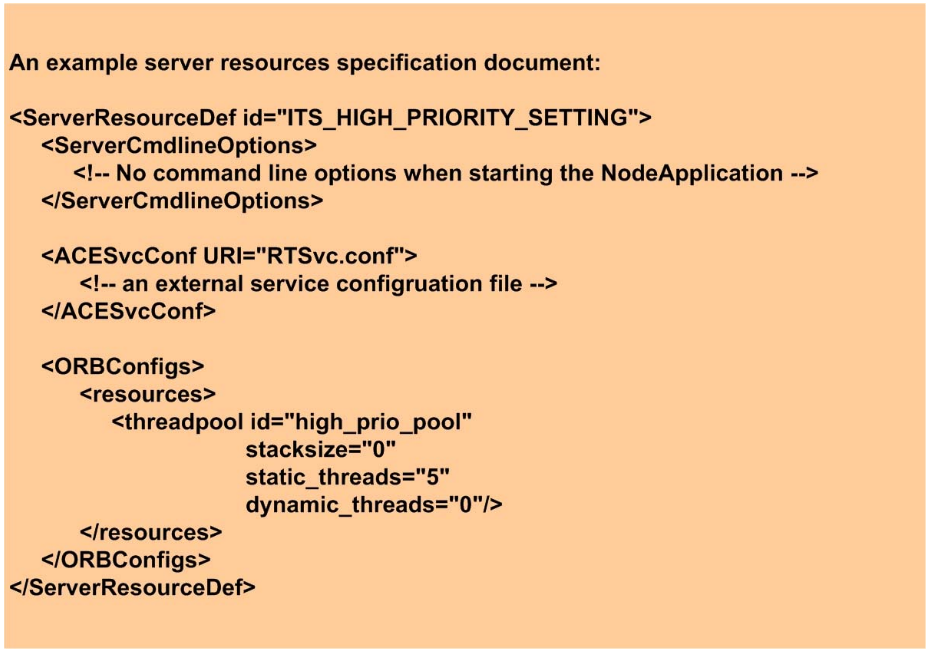


Figure III.6: Specifying RT-QoS requirements

The ORB configurations defined by the *server resources XML schema* are used to configure *NodeApplication* processes that host components, thereby providing the necessary resources for the components to operate. Since the deployment plan describes the components and the artifacts required to deploy the components, DAnCE extends the standard OMG D&C deployment plan to specify the server resource configuration options.

As shown in Figure III.3, `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 run-time overhead of parsing XML files repeatedly. The IDL data structure output of the `XMLConfiguration-Handler` is input to the `ExecutionManager`, which propagates the information to the `DomainApplicationManager` and `NodeApplicationManager`. The `Node-ApplicationManager` uses the server resource-related 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 priority-banded connection models.


An example server resources specification document:

```
<ServerResourceDef id="ITS_HIGH_PRIORITY_SETTING">
   <ServerCmdlineOptions>
      <!-- No command line options when starting the NodeApplication -->
   </ServerCmdlineOptions>

   <ACESvcConf URI="RTSvc.conf">
       <!-- an external service configruation file -->
   </ACESvcConf>

   <ORBConfigs>
       <resources>
          <threadpool id="high_prio_pool"
                      stacksize="0"
                      static_threads="5"
                      dynamic_threads="0"/>
       </resources>
   </ORBConfigs>
</ServerResourceDef>
```

Figure III.7: Example Server Resources Specification

ITS components, such as `ItemLocationSensor` and `WorkflowManager`, have stringent QoS requirements since they handle real-time item delivery activities. The server resource configurations for all nodes hosting these components are specified via an MDE tool. Figure III.7 shows an example XML document that specifies the server resource configurations defined by a system deployer. The `XMLConfigurationHandler` parses the descriptors produced by the MDE tool to notify the `NodeApplicationManager`. To honor all the specified configurations, the component servers hosting these components are configured with server-declared priority model with the highest CORBA priority, thread pools with preset static threads, as well as priority-banded connections.

III.3 Summary

Component middleware is intended to enhance the quality and productivity of software and software developers by elevating the level of abstraction used to develop distributed systems. Conventional middleware, however, generally lacks mechanisms to handle deployment concerns for distributed real-time and embedded (DRE) systems. This chapter described how we addressed these concerns in DAnCE, which is an open-source implementation of the OMG's D&C specification targeted for deploying and configuring DRE systems based on Lightweight CCM. DAnCE leverages MDE tools and QoS-enabled component middleware features to support (1) the efficient storage and retrieval of component implementations, (2) component activation, passivation, and removal semantics within component assemblies, (3) configuring QoS-related client/server resources, and (4) integrating common middleware services into applications.

CHAPTER IV

TECHNIQUES FOR DEPLOYMENT AND CONFIGURATION OF PUBLISH/SUBSCRIBE SERVICES

An increasing number of distributed real-time and embedded (DRE) systems require middleware support for real-time transfer of control and data among a large number of heterogeneous entities that coordinate with each other in a loosely coupled fashion. Examples of such systems include military systems like the Joint Battlespace Infosphere (JBI), telecommunications systems involving large-scale network monitoring and management, environmental emergency response systems requiring real-time coordination between various civilian emergency response units, and supervisory control and data acquisition (SCADA) systems requiring real-time robust control and data communication.

Perhaps the most critical middleware service for the types of DRE systems outlined above are asynchronous, event-based publish/subscribe services [9]. The publish/subscribe architecture is a powerful paradigm for event-based communication because it provides *anonymity*, by decoupling the interfaces between event publishers and subscribers; and *asynchronism*, by automatically notifying subscribers when a specified event is generated. These design principles reduce software dependencies and support the loose coupling requirements of these DRE systems.

Some widely used real-time publish/subscribe services for DRE systems based on *object-oriented* middleware include CORBA Real-Time Event Service (RTES) [35], CORBA Real-Time Notification Service (RTNS) [32] and OMG's Data Distribution Service (DDS) [73]. For example, the CORBA RTES is based on OMG's Real-Time CORBA [83] and provides low-latency/jitter event dispatching, support for periodic processing, dynamic client connection management, centralized event filtering, and efficient use of network and computational resources, which are well-suited for DRE systems with stringent QoS requirements.

DDS exploits topic-based anonymous publish/subscribe mechanisms for DRE systems running in a dynamic environment. Additionally, many of the requirements of the DRE systems can be met scalably and robustly via federations of real-time event channels across the communication network.

Recent trends [42, 63, 112] indicate that *QoS-enabled component middleware* platforms, such as CIAO [124], PRiSM [99], and Qedo [98], are increasingly used to develop and deploy next-generation DRE systems. The increasing use of QoS-enabled component middleware in DRE systems compounded by the need for real-time publish/subscribe services to support a large class of DRE systems requires the integration of the real-time publish/subscribe paradigm within QoS-enabled component middleware. Unfortunately, standards-based component middleware do not yet specify how publish/subscribe services can be robustly supported within component middleware. Moreover, to date there is a general lack of systematic studies that address these concerns.

Although performance evaluation metrics for real-time publish/subscribe object-oriented middleware services are available [105], there is a general lack of information and insights into the design and performance evaluation of real-time publish/subscribe services within QoS-enabled component middleware. In this chapter, we systematically evaluate the benefits and limitations of different design alternatives for integrating real-time publish/-subscribe services within QoS-enabled component middleware architectures. We describe how we applied pattern-driven design and meta-programming techniques in realizing the most promising choice among these alternatives, which is based on the container programming model. Our study shows that the container-managed real-time publish/subscribe services provide predictable and comparable performance when compared to their object-oriented counterparts, which provides key guidance in the suitability of real-time publish/-subscribe services in component technologies for DRE systems.

IV.1 Architectural Design Choices for Integrating Real-Time Publish/Subscribe Services

In this section we describe three different design choices for integrating real-time publish/-subscribe services within QoS-enabled component middleware. To make our discussions concrete, we describe these choices in the context of OMG's Lightweight CORBA Component Model (LwCCM) [80], which is an emerging component middleware standard for DRE systems and implemented by our CIAO QoS-enabled component middleware. It is worth noting that many of our discussions here are also applicable to other component models as well.

IV.1.1 Evaluating Publish/Subscribe Service Integration Design Choices

Before we delve into describing the design choices, we first provide an intuitive description of how CCM components in a QoS-enabled component middleware use the publish/-subscribe paradigm for event communication. Figure IV.1 illustrates how CCM components can publish and subscribe events through real-time event channels. As illustrated in the figure, QoS configurations for the event dispatching are available at three different scopes, *i.e.*, *channel scope*, *port scope*, and *event scope*, which should ideally be configured and integrated into the component middleware architecture in an intuitive manner. The goal of this chapter is to evaluate different design choices for integrating real-time publish/subscribe mechanisms within QoS-enabled component middleware.

Although directly using object-oriented real-time publish/subscribe services is viable for small-scale DRE systems with a small number of components, such an approach will not scale for complex DRE systems with hundreds or even thousands of components, since the OO-based approach requires imperative-based programming interfaces to implement DRE systems which makes components sensitive to changes. On the other hand, the CCM

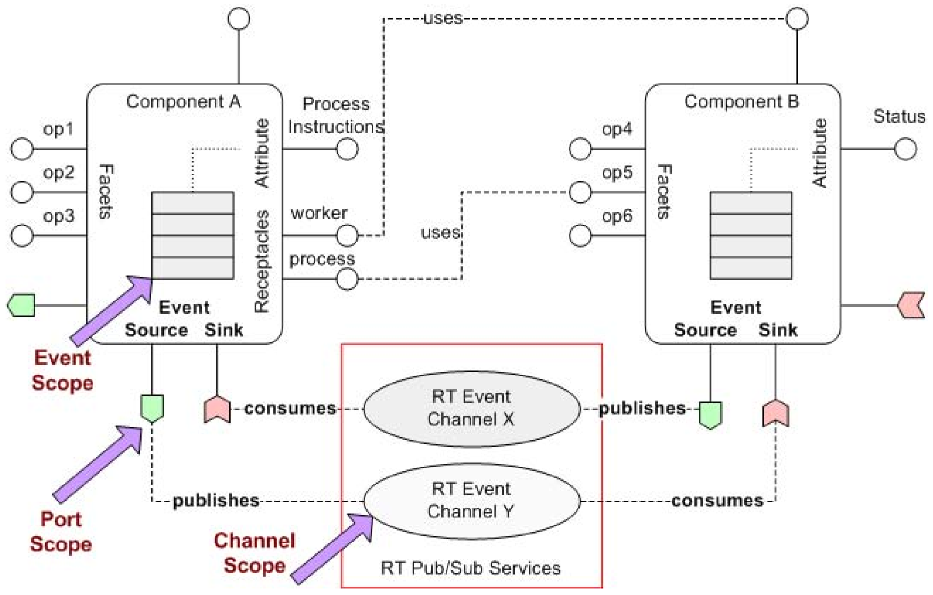


Figure IV.1: Using Publish/Subscribe Services in QoS-enabled Component Middleware

standard does not specify how publish/subscribe services can be composed within component middleware. This issue is further complicated by the need for *real-time* publish/-subscribe services to be integrated within QoS-enabled implementations of CCM, such as our Component Integrated ACE ORB (CIAO) [124].

Based on this information, this section describes three possible architectural choices for integrating real-time publish/subscribe services within LwCCM. Figure IV.2 illustrates different architectural choices for integrating publish/subscribe services. The three architectural choices include (1) *component-managed* – where the event channel can be represented as an application-level component, (2) *container-managed* – where the event channel can be encapsulated within the container, and (3) *component server-managed* – where the publish/subscribe service can reside within the component server. This section describes each architecture choice in detail and analyzes the advantages and disadvantages of each approach.

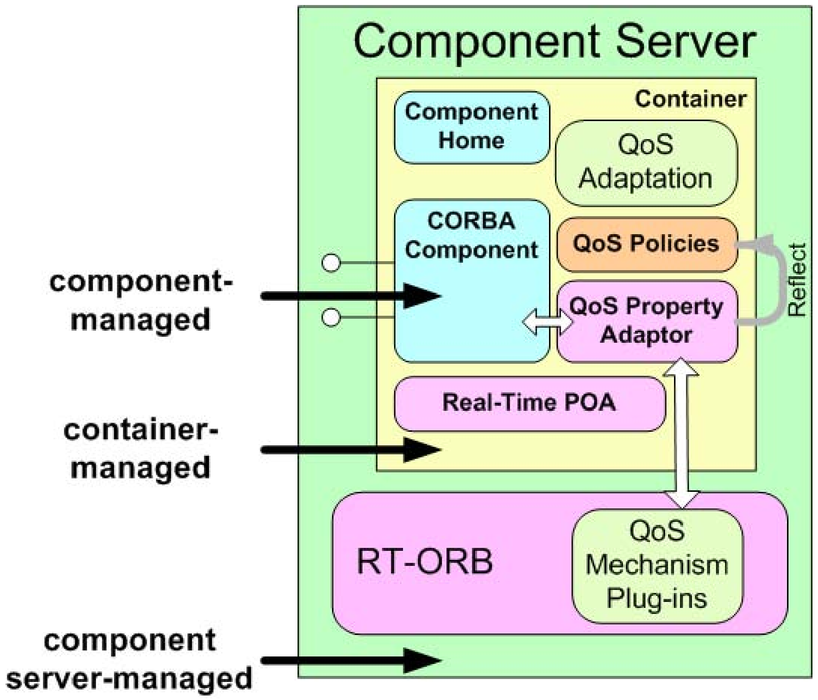


Figure IV.2: Architectural Choices for Integrating Publish/Subscribe Services in CCM

IV.1.1.1 Component-Managed Publish/Subscribe Services

Design: The first architecture choice for providing real-time publish/subscribe services in component middleware is to instantiate them as application-level CCM components as illustrated in Figure IV.3. In this architecture, the interfaces provided by the publish/-subscribe services are exposed as component facet ports. These ports contain methods to connect component event sources/sinks to the event channel, configure event service real-time properties, and push events.

Analysis: The primary advantage of this approach is its simplicity. The complexity needed to implement a publish/subscribe service component is rudimentary since the encapsulated service already implements the publish/subscribe service functionalities, making the full set of service features readily available to other components. Instantiating and deploying multiple publish/subscribe service components follows the same rules that apply to standard components.

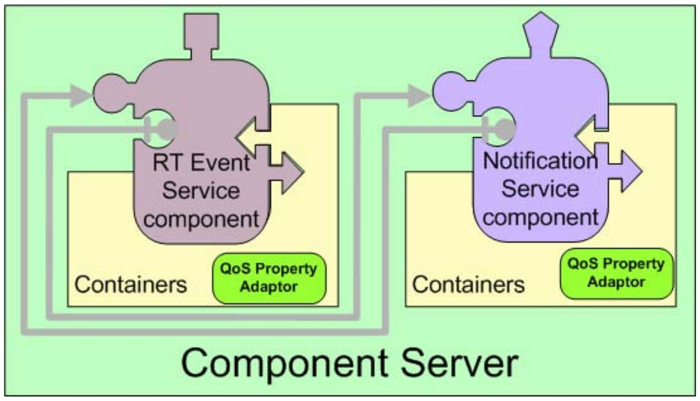


Figure IV.3: Publish/Subscribe Service as CCM Component

However, there are a number of disadvantages of using component-managed publish/-subscribe mechanisms. Generally speaking, the shortcomings of the *object-oriented* model still manifest in this architecture. First, the component glue-code, or servant, must manipulate publish/subscribe interfaces directly, which exposes low-level CORBA programmatic details thereby defeating the declarative approaches used by component middleware. Second, the component servant logic must encapsulate QoS and real-time properties, which inhibits the flexibility and reusability of components across different operating contexts and environments. Third, it is impossible to substitute or interchange different real-time publish/subscribe services without recompilation of components because the servant implementation within a component is tightly coupled with a specific type of publish/subscribe service. Finally, application-level components must now be responsible for managing the publish/subscribe lifecycles.

In conclusion, this architecture tightly couples the service provisioning behaviors into the component implementation thereby hampering the reusability and evolution of DRE systems. Additionally, the component-based publish/subscribe architecture conflicts with the standard CCM container programming model, which makes the container a mediator between application-level components and common middleware services. Ironically, in this case the publish/subscribe service itself is encapsulated within the component. Finally, this architecture results in a remote call to transmit an event to the publish/subscribe service component, which must be handled by the ORB and hence requires additional processing and levels of indirection. Given the number of unfavorable consequences of utilizing this architecture, it is not appropriate for the majority of component-based DRE systems, especially large-scale systems that require highly flexible and customizable QoS guarantees at a low cost.

IV.1.1.2 Container-Managed Publish/Subscribe Services

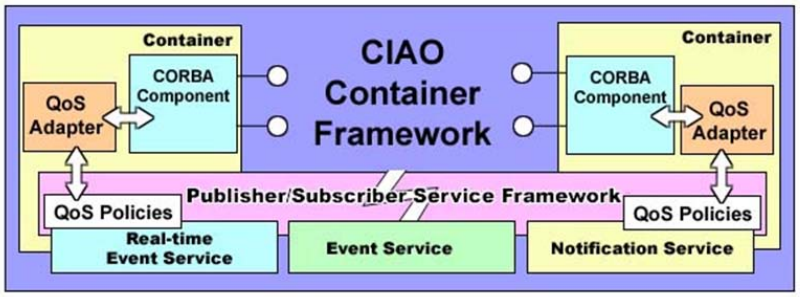


Figure IV.4: Publish/Subscribe Services Within Container

Design: Figure IV.4 depicts a second architecture for providing real-time publish/subscribe services in component middleware where a publish/subscribe service is encapsulated within the CCM container. In this architecture, the container is responsible for managing publish/-subscribe service lifecycles and their clients, initializing channels and gateways, connecting publishers and subscribers, configuring QoS and real-time properties, managing publisher and subscriber component servants, and setting up the federation among multiple event channels across different containers. In this design, the container exposes two distinct interfaces. One interface provides configuration methods and is invoked by the component deployment framework based on the properties specified in XML-based metadata descriptors that describe configuration decisions. The second interface provides a push method and is invoked by application-level components.

Analysis: There are many advantages to this architecture. First, since the publish/subscribe services are managed by the container, the business logic of application components are decoupled from the publish/subscribe service configuration. This decoupling enables real-time publish/subscribe service configurations and specifications to be validated and synthesized via high-level model-driven engineering (MDE) tools [24] prior to system deployment time, which increases the level of abstraction and automation of the DRE system development process. This separation of concerns maximizes the flexibility and reusability of components by allowing them to be reconfigured with different QoS properties and/or services as required by new and changing operating contexts without making any changes to the application component logic or glue-code thereby obviating the need for recompilation.

Second, this design reduces the memory footprint of individual components and preserves their lightweight nature. Although the component deployment framework is exposed to the implementation details of the real-time publish/subscribe services (since the deployment framework must instantiate and configure the channels) rather than component servant glue code, it is not important for the deployment framework to be as lightweight as the CCM components because the deployment framework is not part of the run-time system and does not consume resources after a DRE system is deployed.

Third, this architecture aligns with the CCM container programming model and defers publish/subscribe configuration-related decisions until deployment time, which allows additional optimizations to be incorporated depending on knowledge of the deployment context. For example, it may not be known until deployment time which network links have high latency or low reliability, yet this information is critical to determining the best possible real-time publish/subscribe service configuration.

The disadvantage of container-managed event channel architecture is the difficulty encountered in actually implementing it effectively and efficiently due to the complexity of the CCM container architecture and its programming model. There are a number of design challenges that arise when pursuing this design choice. We discuss in Section IV.2 how we resolve these design challenges based on our patterns-driven solutions.

IV.1.1.3 Component Server-Managed Publish/Subscribe Services

Design: The third alternative architecture for providing real-time publish/subscribe services in component middleware is to host them within the component server, which is similar to existing approaches of supporting services in object-oriented middleware. In this architecture, publish/subscribe services are still accessed and manipulated via the container. However, the component server-managed architecture is fundamentally different from the container-managed architecture in that the component server is a lower-level entity which hosts all the components, which in turn end up sharing the same publish/subscribe service.

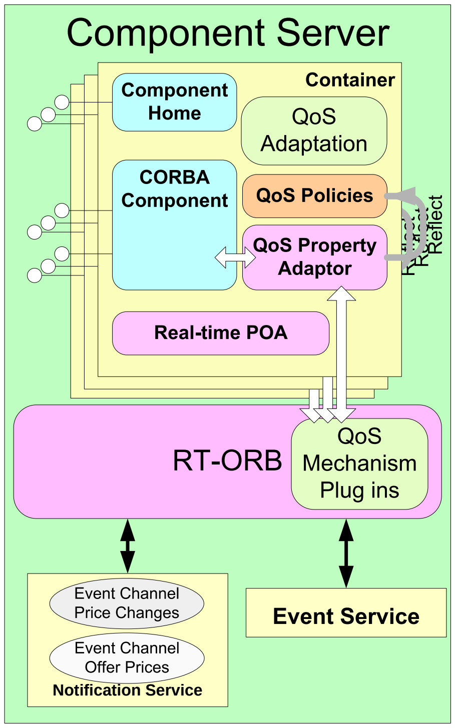


Figure IV.5: Publish/Subscribe Services Within Component Server

Analysis: The advantages present in the container-managed architecture are also applicable to the component server-managed architecture: components are still isolated from publish/subscribe services in such a way that they remain configurable after compilation, and push operations result in only local method invocations. However, the component server-managed architecture is more coarse-grained *i.e.*, a large number of components may be required to share a single service thereby affecting differentiated treatment to application components depending on their real-time needs.

For applications that require either multiple publish/subscribe services on a single host or those who wish to maximize component flexibility to allow for future enhancements or modifications, the component server-managed architecture may be too restrictive. On the other hand, for applications that do not require these capabilities, the component server-managed architecture results in a simpler configuration and deployment process, which reduces development effort. In the case of very large-scale DRE systems, the savings may be substantial if sharing is desired. However, for DRE systems that require partitioning and configuring the system capabilities based on priorities, load balancing and reliability, this coarse-grained approach is not suitable.

Summary: Based on our analysis of the benefits and limitations of each design choice, we have selected the *container-managed* architecture as our design choice to obtain additional guidance on its applicability and performance. The container-managed architecture provides the most flexible real-time publish/subscribe services while preserving the benefits of the CCM container programming model; hence, it is applicable for the widest range of DRE systems and especially useful for developing large-scale complex DRE systems.

Due to the complexity of the CCM container architecture, component programming model, and the associated D&C model, there are a number of challenges in the context of this design architecture. In Section IV.2, we show how this architecture can be implemented in a way that is very efficient, lightweight, and flexible enough to accommodate new services to be plugged in with little modification.

IV.2 Challenges and Solution of Integration, Configuration, and Deployment of Publish/Subscribe Services in Component Middleware

This section describes the design and implementation of a container-based approach to provision of real-time publish/subscribe services in CIAO, which draws on the combined strength of pattern-driven design [28], model-driven engineering (MDE) [103], and meta-programming techniques [17]. We divide this section into three parts. First, we discuss the integration design goals and our implementation strategies. We follow this by the configuration issues, which arise due to the complexities to ensure syntactic and semantic correctness of configuring various publish/subscribe services. Finally, we discuss deployment issues, which arise due to the declarative as opposed to imperative approaches used for deployment in component middleware.

IV.2.1 Addressing Integration Challenges of Publish/Subscribe Services in Component Middleware

Figure IV.6 gives an overview of the design of the container-managed real-time publish/-subscribe service architecture as outlined in Section IV.1. The fundamental objective of this design is to increase the efficiency and flexibility of large-scale DRE systems, while preserving the lightweight nature of CCM components and the CIAO middleware framework. The numbered bullets in this diagram depict the flow of control among different entities in the CIAO QoS-enabled component middleware architecture [124].

Design goal 1, which calls for providing a service-independent representation of real-time properties since different publish/subscribe services depend on different representations of real-time properties.

Solution approach → Adapter pattern: We apply the *adapter* pattern that converts service-specific representations of real-time properties into service-independent representations. The benefits of this design are twofold: (1) component developers need not concern themselves with peculiar configuration interfaces, and (2) no matter what changes occur to

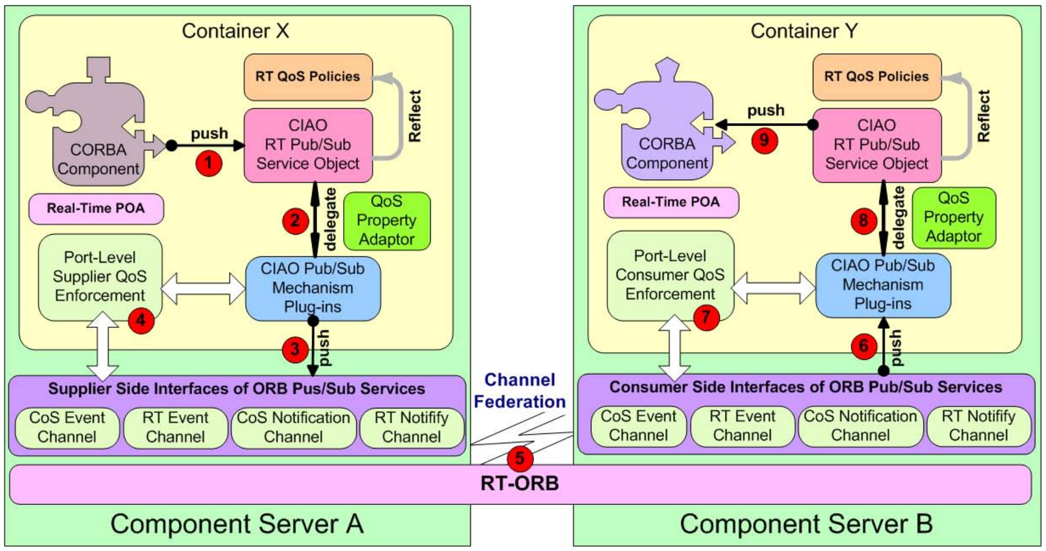


Figure IV.6: CIAO Publish/Subscribe Architecture

the underlying publish/subscribe services, the interface exposed to components does not change.

Design goal 2, which requires enhancing reuse and extensibility by allowing new publish/-subscribe services to be easily plugged-in.

Solution approach → Strategy pattern: This design goal is satisfied using the *strategy* pattern, which results in service implementations that are interchangeable from the container perspective. After object creation, the container has no knowledge of the actual algorithm being used, which enables fast operation delegations and simplifies container design.

Design goal 3, which emphasizes reduction in the memory footprint of the container by decoupling the creation of publish/subscribe service instances from their representation.

Solution approach → Builder pattern: The creation of most real-time publish/subscribe service instances is complex since a lot of objects must be instantiated and configured properly. A CIAO container defines a *builder* class that encapsulates the complexity, which results in finer control of the construction process, isolation of construction code, and the ability to vary the service configurations.

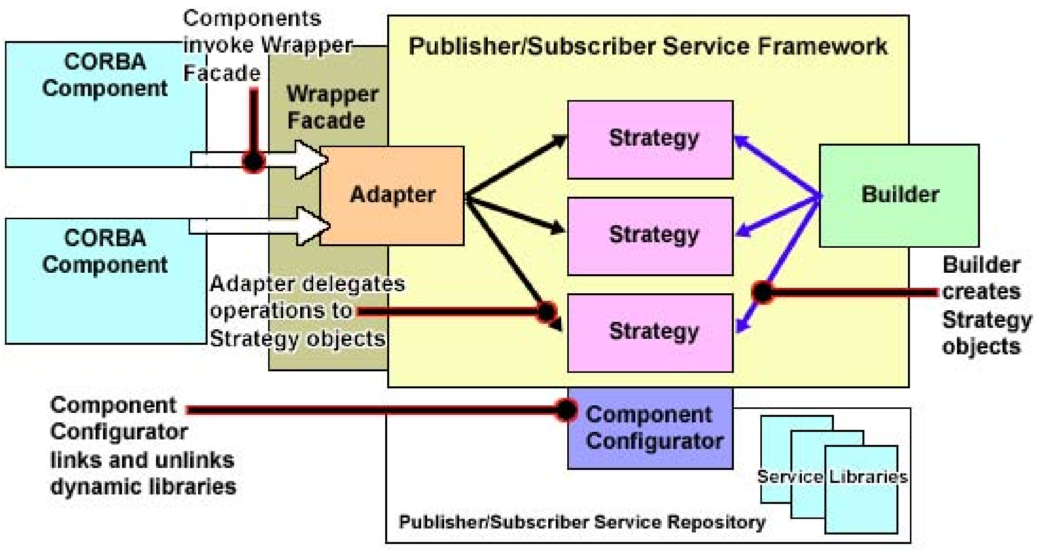

Components invoke Wrapper Facade
CORBA Component
CORBA Component
Publisher/Subscriber Service Framework
Wrapper Facade
Adapter
Strategy
Strategy
Strategy
Builder
Builder creates Strategy objects
Adapter delegates operations to Strategy objects
Component Configurator links and unlinks dynamic libraries
Component Configurator
Service Libraries
Publisher/Subscriber Service Repository

Figure IV.7: Pattern Interactions in the CIAO Publish/Subscribe Service Framework

Design Goal 4, which requires ensuring that the components incur only the cost of services that are required by deferring publish/subscribe service selection and configuration decisions until *run time* instead of *design time*.

Solution approach → Component Configurator pattern: In CIAO, a *component configurator* enables publish/subscribe service libraries to be loaded dynamically on-demand to avoid encumbering the application with unused services, while still allowing components to wait until deployment time to select a particular service. This mechanism provides the flexibility to initiate, suspend, resume, and terminate services.

Design Goal 5, which requires a component be able to access the full set of QoS features available in real-time publish/subscribe services by encapsulating service-specific QoS specification operations within a high-level interface.

Solution approach → Wrapper Facade pattern: The CIAO container framework implements a high-level configuration interface based on *wrapper facades* that forwards invocations to the corresponding service-specific operations for each publish/subscribe service. This design results in a concise and robust common programming interface capable of configuring the QoS features in multiple dissimilar publish/subscribe services.

IV.2.2 Addressing Configuration Challenges of Publish/Subscribe Services in Component Middleware

To derive benefits from QoS-enabled component middleware for event-based DRE systems, the complexities associated with configuring publish/subscribe services must be addressed. This section explains the context in which each challenge arises, identifies the specific problems, and then presents solution approaches that help resolve these challenges.

IV.2.2.1 Challenge 1: Configuring Publish/Subscribe Quality-of-Service

Context. *Configurability* is an important requirement for many publish/subscribe services developed using middleware. For example, various operating policies (such as threading and buffering strategy) of the CORBA publish/subscribe services can be customized programmatically via invocations on a configuration interface. The drawbacks with DOC middleware approaches to configurability, however, are (1) *reduced flexibility* due to tight coupling of application logic with crosscutting configuration and deployment concerns, such as publish/subscribe relationships and choice of various types of publish/subscribe services, such as the CORBA-based Event, Real-Time Event, and Notification Services, and (2) *impeded reuse* due to tight coupling of application logic with specific QoS properties, such as event latency thresholds and priorities.

In contrast, component middleware publish/subscribe services enhance flexibility and reuse by using meta-programming techniques (such as the XML descriptor files in CCM) to specify component configuration and deployment concerns. This approach enables QoS requirements to be specified *later* (*i.e.*, just before run-time deployment) in a system's lifecycle, rather than *earlier* (*i.e.*, during component development). For example, the configuration framework provided by the CIAO component middleware parses XML configuration files and make appropriate invocations on a publish/subscribe service configuration interface. This approach is useful for DRE systems that require custom QoS configurations for various target OS, network, and hardware platforms that have different capabilities and properties.

Problem. Conventional component middleware relies upon *ad hoc* techniques based on *manually* specifying the QoS requirements for DRE component systems. Unfortunately, configuring component middleware manually is hard [67] due to the number and complexity of operating policies, such as transaction and security properties, persistence and lifecycle management, and publish/subscribe QoS configurations. These policies exist at multiple layers of middleware and often employ non-standard legacy specification mechanisms, such as configuration files that use proprietary text-based formats.

Moreover, given component interoperability needs across various platforms (*e.g.*, CCM and J2EE) and the existence of multiple publish/subscribe services within individual platforms (*e.g.*, the CORBA Event Service and Notification Service), a component-based application may use several publish/subscribe services. To further complicate matters, certain combinations of policies are semantically invalid and can result in system failure. For example, if multiple levels of priorities for events are supported, a priority-based thread pool model should be used rather than a reactive threading model [104]. Care should be taken to ensure that lower level configurations support end-to-end priorities, *e.g.*, using Real-Time CORBA priority-banded connections [92].

Most publish/subscribe services based on DOC middleware (including the CORBA Event and Notification Services) do not validate QoS specifications automatically. Moreover, it is hard to *manually* validate QoS configurations for semantic compatibility. This process is particularly daunting for large-scale, mission-/safety-critical DRE systems.

Solution approach → Develop MDE tools to create publish/subscribe service configuration models. MDE tools can help application developers create QoS specifications for

DRE systems more rapidly and correctly by automatically generating configuration descriptor files and enforcing constraints among publish/subscribe policies via model checkers [36]. These benefits are particularly important when component applications are maintained and evolved over an extended period of time since (1) QoS configurations can be modified more easily to reflect changing OS, network, and hardware platforms and (2) QoS configurations for system enhancements can be checked systematically for compatibility with legacy specifications.

To attain these benefits, we developed the *Event QoS Aspect Language* (EQAL), which is an MDE tool that models configurations for three CORBA-based publish/subscribe services: (1) the Event Service [75], (2) Real-Time Event Service [35, 105], and (3) Notification Service [77]. EQAL informs users if invalid combinations of QoS policies are specified. After publish/subscribe QoS models are complete and validated, EQAL can also synthesize the XML configuration files used by the underlying component middleware to configure itself. Section IV.2.2.3 gives a detailed description about how we address this challenges in the context of the EQAL MDE tool.

IV.2.2.2 Challenge 2: Configuring Publish/Subscribe Services in Target Networks

Context. *Scalability* is another important requirement for many publish/subscribe systems. Large-scale publish/subscribe systems consist of many components and event channels distributed across network boundaries and possibly different administrative domains, and each event channel may have many consumers. Naive implementations of publish/subscribe services send a separate event across the network for each remote consumer, which can transmit the same data multiple times (often to the same target host) and incur network and host overhead that is excessive for many resource-constrained DRE applications. As the number of channels and/or consumers grows, these types of publish/subscribe services can become a bottleneck.

To minimize the overhead of publish/subscribe services, multiple event channels can be linked together to form *federated* configurations [88, 105], where event channels are assigned to particular hosts and events received by one channel are propagated automatically to other channels in the federation. Figure IV.8 illustrates how CIAO's publish/subscribe services support federated event channels. In CIAO's federated publish/subscribe services,

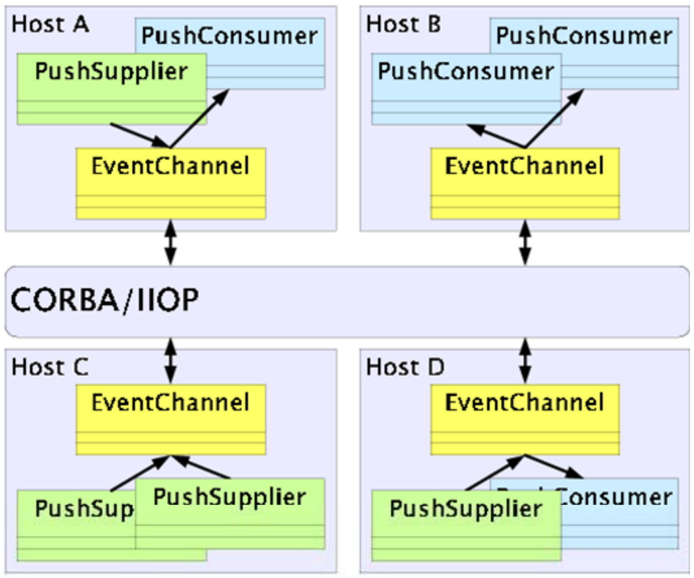


Figure IV.8: Federated Event Channels in CIAO

suppliers and consumers that are collocated on the same host connect to a local event channel. Each local event channel communicates with other event channels when events sent by suppliers are destined for consumers on remote hosts. This design reduces latency in large-scale DRE systems when consumers and suppliers exhibit *locality-of-reference*, *i.e.*, where event consumers are on the same host as event suppliers. In such cases, only local C++ method calls are needed instead of remote CORBA operation calls. Moreover, if multiple remote consumers are interested in the same event, only one message is sent to each remote event channel, thereby reducing network utilization.

In CIAO's federated publish/subscribe services, event channel *gateways* are used to mediate the communication between remote event channels, while suppliers and consumers communicate with each other via local event channels. Each gateway is a CORBA component that connects to the local event channel as a supplier and connects to the remote event channel as a consumer. CIAO supports three types of event channel gateways: CORBA Internet Inter-ORB Protocol (IIOP), User Datagram Protocol (UDP), and IP multicast. In CIAO's federated publish/subscribe services, application developers need not write tedious and error-prone code manually to perform bookkeeping operations, such as creating and initializing gateways that federate event channels.

CCM deployment tools install individual components and assemblies of components on target sites, which are normally a set of hosts on a network. Similarly, event channels must be assigned to hosts in the target network. CIAO's federated publish/subscribe services are integrated via its DAnCE component deployment tool. The input to DAnCE is an XML data file that (1) specifies the event channel deployment sites and (2) automatically creates and initializes the event channel gateways at the appropriate sites.

Problem. Although the DAnCE CCM deployment tool provided by CIAO shields application developers from having to write bookkeeping code for its federated publish/subscribe services, application developers still must *hand-craft* federation deployment descriptor metadata using an XML schema based on the OMG Deployment and Configuration specification [79]. Hand-crafting descriptor metadata involves determining information about the type of federations (*i.e.*, CORBA IIOP, UDP, or IP multicast), identifying remote event channels, and identifying local event channels. Moreover, the metadata must address the following deployment requirements: (1) each host could have its own event channels, event consumers, event suppliers, and event channel gateways, (2) each event consumer and event supplier only communicates with an event channel collocated in the same host, (3) event channels distributed across network boundaries are connected through event channel gateways, (4) each connection between an event channel and an event supplier should be uniquely identified, (5) each connection between an event channel and an event consumer should be identified by using existing events, as defined in step (4), and (6) each connection between an event channel and an event channel gateway should also be identified by using existing events, as defined in step (4).

Experience has shown [112, 113] that it is hard for DRE developers to keep track of many complex dependencies when deploying federated publish/subscribe services. Without tool support, the effort required to deploy a federation involves hand-crafting deployment descriptor metadata in an *ad hoc* fashion. Since large-scale DRE systems may involve many different types of events and event channels, *ad hoc* ways of writing metadata to deploy publish/subscribe services are tedious and error-prone. Addressing this challenge requires techniques that can analyze, validate, and verify the correctness and robustness of federated event channel deployments.

Solution approach → Develop MDE tools to deploy event channel federations in a visual, intuitive way. MDE tools can synthesize the metadata for deploying a federated publish/subscribe service from *models* of the interactions among different event-related components (*e.g.*, event suppliers, event consumers, event channels, and various types of event channel gateways). MDE tools can also generate the metadata needed to deploy federated publish/subscribe services that are syntactically and semantically valid. Section IV.2.2.3 gives a detailed description about how we address these challenges in the context of our EQAL MDE tool.

IV.2.2.3 Design of Event QoS Aspect Language (EQAL)

EQAL is developed using the Generic Modeling Environment (GME) [60], which is a generative technology for creating domain-specific modeling languages and tools [45]. GME can be programmed via *metamodels* and *model interpreters*. Metamodels define modeling languages (called *paradigms*) that specify the syntax and semantics of the modeling element types, their properties and relationships, and presentation abstractions defined by a domain-specific modeling language (DSML). Model interpreters can traverse a paradigm's modeling elements and perform various actions, such as analyzing model properties and generating code.

The EQAL paradigm in GME consists of the two complementary entities described below:

• **EQAL metamodel** , which defines a modeling paradigm in which engineers specify the desired publish/subscribe service (*e.g.*, the CORBA Event Service, Notification Service, or Real-Time Event Service) and the configuration of that service for each component event connection. Based on application needs, engineers can also specify how event channels are assigned to different hosts and whether/how they must be linked together to form federations.

To address the publish/subscribe service configuration challenges describe above, the EQAL *configuration paradigm* in the EQAL metamodel specifies publish/subscribe QoS configurations, parameters, and constraints. For example, the EQAL metamodel contains a distinct set of modeling constructs for each publish/subscribe service supported by CIAO. Example policies and strategies that can be modeled include filtering, correlation, timeouts, locking, disconnect control, and priority.

• **EQAL model interpreters** that can (1) validate configuration and deployment models and (2) synthesize text-based middleware publish/subscribe service and federation service configuration- and deployment-specific descriptor files from models of a given component assembly. Engineers can build publish/subscribe service configurations for component applications using the EQAL modeling paradigm and its model interpreters.

Dependencies among publish/subscribe QoS policies, strategies, and configurations can be complex. Ensuring coherency among policies and configurations is therefore a non-trivial source of complexity in component middleware [114]. During the modeling phase, EQAL ensures that dependencies between configuration parameters are enforced by declaring constraints on the contexts in which individual options are valid, *e.g.*, priority-based thread allocation policies are only valid with component event connections that have assigned priorities. EQAL can then automatically validate configurations and notify users of incompatible QoS properties during model validation, rather than at component deployment- and run-time.

To ensure semantically consistent configurations, violation of constraint rules should be detected early in the modeling phase rather than later in the component deployment phase. To support this capability, EQAL provides a constraint model checker that validates the syntactic and semantic compatibility of event channel configurations to ensure the proper functioning of publish/subscribe services. EQAL's model checker is developed based on GME's constraint manager, which is a lightweight model checker that implements the standard OMG Object Constraint Language (OCL) specification [81].

EQAL's model interpreters perform the following two distinct configuration aspects:

• **XML descriptor generation.** EQAL contains an interpreter that synthesizes XML descriptors used by the DAnCE component deployment framework to indicate the QoS requirements of individual component event connections. Since the CCM specification does not explicitly address the mechanisms for ensuring component QoS properties, the EQAL-generated descriptors are based on a schema developed for the Boeing Bold Stroke project for their Prism [114] extensions to CCM (the XML descriptors remain compliant with the CCM specifications, however). Boeing's Bold Stroke schema has been carefully crafted, refined, tested, and optimized in the context of production DRE avionics mission computing systems [112, 113].

EQAL generates the XML descriptors for one service at a time. To complete the interpretation process, EQAL makes multiple passes through the model hierarchy, corresponding to each different type of publish/subscribe service, until all the service connections are configured. To simplify the interpreter implementation, EQAL [25] uses the Visitor pattern [28], which utilizes a double-dispatch mechanism to apply file-generation operations to different types of modeling elements.

• **Service configuration file generation.** EQAL also contains an interpreter that generates event channel service configuration files, called `svc.conf` files, that are used by the underlying publish/subscribe services to select the appropriate behaviors of event channel resource factories [88]. These factories are responsible for creating many strategy objects that control the behavior of event channels. CIAO supports many (*i.e.*, more than 40) options/policies for different types of publish/subscribe services, which increases the complexity for application developers who must consider numerous design choices when configuring the publish/subscribe services. Interactions between event channel policies are complex due to the possibility of incompatible groupings of options, making hand-crafting these files hard, *e.g.*, priority-based thread allocation policies are only valid with component event connections that have assigned priorities.

Much of the complexity associated with validating event channel QoS configurations is accomplished by EQAL's modeling constraints and GME's lightweight model checker. These constraints prevent application developers from specifying inconsistent or invalid combinations of policies. After a set of policy settings is validated via modeling constraints, EQAL generates an *event channel descriptor* (`.ecd`) file that contains valid combinations of policy settings chosen for a particular service configuration.

To address the publish/subscribe federation service deployment challenges, the EQAL *deployment paradigm* specifies how components and event channels are assigned to hosts on a target network. To address the scalability problem in any large-scale event-based architecture, CIAO provides publish/subscribe services that support event channel federation. With CIAO's publish/subscribe services, an event channel federation can be implemented via CORBA gateways. Application developers can configure the location of the gateways to utilize network resources effectively.

For example, collocating a gateway with its consumer event channel (*i.e.*, the one it connects to as a supplier) eliminates the need to transmit events that have no consumer event channel subscribers. Application developers can also choose different types of gateways based on different application deployment scenarios with different networking and computing resources. These deployment decisions have no coupling with, or bearing on, component application logic. The same set of components can therefore be reused and deployed into different scenarios without modifying application code manually.

The EQAL modeling paradigm allows three types of federation (*i.e.*, CORBA IIOP, UDP, or IP multicast) to be configured in a deployment. For event channel federation models, the EQAL metamodel defines two levels of syntactic elements:

- The **outer-level**, which contains the host elements as basic building blocks and allows users to define the hosts present in the DRE system and
- The **inner-level**, which represents a host containing a set of elements (including event channels, CORBA IIOP gateways, UDP senders and receivers, IP multicast senders and receivers, and event type references) that allow users to configure the deployment of these artifacts inside a host.

These two levels are associated with each other via *link parts*, which act as connection points between two different views of a model (such as adjacent layers of a hierarchical model) to indicate some form of association, relationship, or dataflow between two or more models. The inner-level elements are exposed to the outer-level in the form of link parts from the outside view, which can be used to connect them to form a federation.

Figure IV.9 is a screenshot that illustrates how we used EQAL to model the outer-level view of CIAO federated publish/subscribe service in a real-time avionics mission computing application. This figure shows the outer-level model of the deployment of the federated publish/subscribe service, which includes nine physically distributed locations that host CCM components. Figure IV.10 shows the inner-level of the federation configurations, which establish four CORBA gateways in the track center module to form a federation that reduces network traffic.

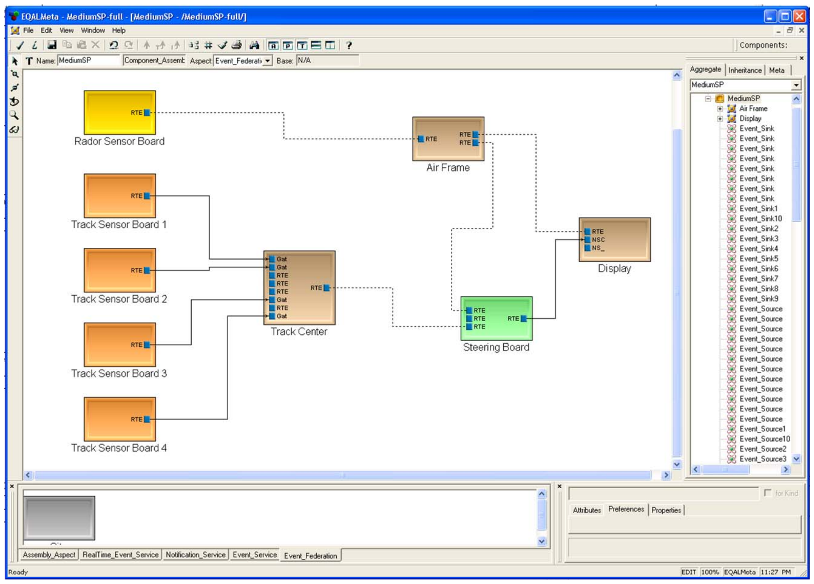


Figure IV.9: Outer-level View of an EQAL Deployment Model for a Real-Time Avionics System

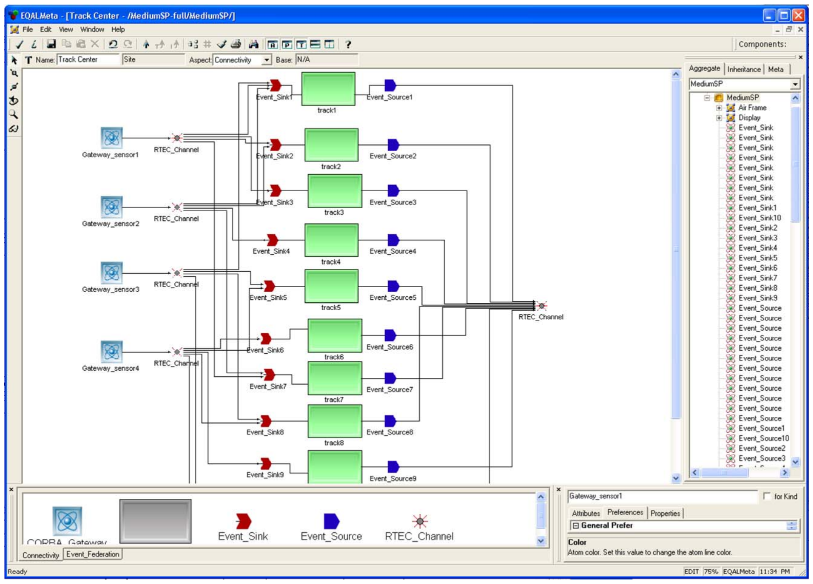


Figure IV.10: Inner-level View of an EQAL Deployment Model for a Real-Time Avionics System

To ensure the validity of event channel federation models during the deployment phase, each event channel's configurations and settings must be model-checked to ensure that they are consistent with the federation types. For example, IP multicast uses the Observer pattern [28] capabilities of CIAO's event channels. When a user chooses IP multicast as the type of event channel federation, this observer functionality must be enabled for IP multicast to work properly. These constraints can be checked automatically using EQAL.

IV.2.2.4 Empirical Evaluation of EQAL

We applied EQAL MDE tool to the OEP's *Medium-sized* (MediumSP) scenario, which is a product scenario in the DARPA PCES OEP [85]. The MediumSP scenario is a representative real-time avionics mission computing system that employs event-driven data flow and control [112, 113]. This scenario consists of 50+ components with complex event dependencies that control embedded sensors and perform calculations to maintain displays. In this type of mission-critical DRE system, reliability and stringent QoS assurance are essential.

The assembly of 50+ components for the MediumSP scenario requires a complicated *component assembly* file that stores the connection information between component ports as XML descriptors, partitions for process collocation, and interrelationships with other descriptors (*e.g.*, the relationship between the interface definitions and component implementations) whose details are spread across other assembly files, such as the *implementation artifact descriptor* (`.iad`) file. EQAL shields DRE system developers from these low-level details by ensuring that all this metadata and dependencies are captured appropriately in various descriptor files it generates in conjunction with other CoSMIC tools, such as PICML [30].

Every component requires two descriptor files: (1) the *software package descriptor* for the component, which contains general information about the software (such as author, description, license information, and dependencies on other software packages), followed by one or more sections describing implementations of that software, and (2) the *servant software descriptor*, which CIAO deployment tools use to load the desired servant library. For ∼50 components, ∼100 files are therefore required. Once again, EQAL shields DRE system developer from these low-level details by generating these files automatically, thereby ensuring that all interdependencies are captured appropriately in the descriptor files.

For the publish/subscribe service in the MediumSP component assembly, a *component property descriptor* (CPF file) is generated for each component event port, an *event channel descriptor* (ECD file) will be generated for each real-time event channel filter, and CIAO's service configuration file (`svc.conf`) file) will be generated for each event channel configuration. As a result, 12 ECD files, 53 CPD files, and 1 `svc.conf` file are generated by EQAL. Figure IV.11 summarizes the lines of code saved by not having to hand-craft these files.

File Type	# of Files	Average # Lines/File	Total Lines
CAD	1	750	750
CSD	50	46	2300
SSD	50	43	2150
CONF	3	6	18
ECD	12	8.58	103
CPF	53	43	2279
Total	**169**	**44.97**	**7600**

CAD: Component Assembly Descriptor	**CONF: Service Configuration File**
CSD: Software Package Descriptor	**ECD: Event Channel Descriptor**
SSD: Servant Software Descriptor	**CPF: Component Property File**

Figure IV.11: Amount of Code Reduction for Metadata by EQAL in Bold Stroke MediumSP Scenario

Each component, event service, and their servants are distinguished via a unique identifier (called a UUID) within the descriptor files mentioned above. Moreover, it is necessary to ensure that when referring to a specific component or event service, the same UUID is referenced across the different descriptor files. This requirement can yield accidental complexities when descriptor files are hand-crafted manually. In the MediumSP scenario, this results in $\sim$100 UUIDs that are referred to across the $\sim$100 descriptor files. EQAL's generative tools eliminate these accidental complexities by synthesizing the proper UUID references in the descriptor files.

IV.2.3 Addressing Deployment Challenges of Publish/Subscribe Services in Component Middleware

Component-based DRE systems require real-time publish/subscribe services to be deployed and configured onto the target execution environment. Common D&C concerns include (1) choices of publish/subscribe services and their bindings to the CCM components (2) process-collocation strategies between CCM components and publish/subscribe services, (3) host-collocation strategy between CCM components and these services, (4) real-time properties on event channels, event ports, and individual events, (5) choices of event channel federation strategies, such as using IIOP based CORBA gateways or UDP based unicast or multicast. To further simplify the DRE system D&C tasks, the real-time publish/subscribe service should ideally be automatically deployed and configured within the container and bound to components as an integral part of the standardized D&C process as defined by the OMG Deployment and Configuration (D&C) [86], and even using the same set of D&C tools based on the above standard.

Although the container-based publish/subscribe service integration approach decouples the functional aspect of CCM components from their publish/subscribe QoS requirements, there is a lack of a mechanism to automate and orchestrate the deployment and configuration process of publish/subscribe services. This section describes how we have employed meta-programming techniques to automate the D&C process of real-time publish/subscribe services for DRE systems. Our solution is based on the OMG Deployment and Configuration (D&C) specification and consists of two complementary abstraction models, one called the *data model* based on XML representation and one called the *run-time model* based on CORBA 2.x IDL.

Data Model. The data model uses XML descriptors to describe the real-time publish/-subscribe service configurations. In order to make our data model compatible with the OMG standard D&C model, we decouple DRE system publish/subscribe service configuration concerns from standards-based component assembly, and then capture these concerns using a different set of XML descriptors. These descriptors are based on our proprietary XML schema called *CIAO Events Descriptors*, which define a rich set of elements called *policies* that capture different D&C concerns of dissimilar real-time publish/subscribe services. Figure IV.12 shows the policies available for the CORBA Real-Time Event Service (RTES) that can be specified based on this *data model*.

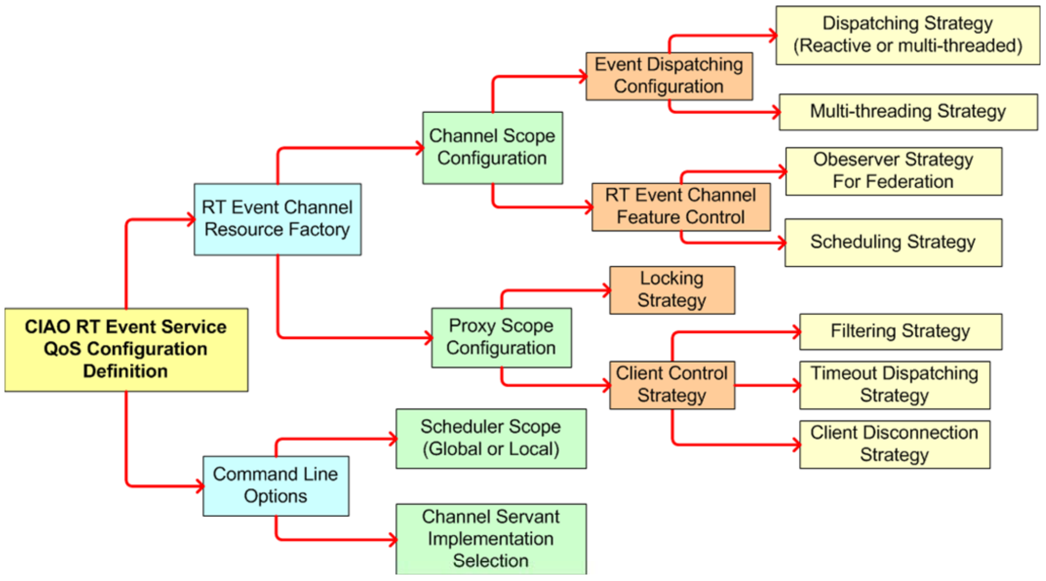

CIAO RT Event Service QoS Configuration Definition
RT Event Channel Resource Factory
Command Line Options
Channel Scope Configuration
Proxy Scope Configuration
Scheduler Scope (Global or Local)
Channel Servant Implementation Selection
Event Dispatching Configuration
RT Event Channel Feature Control
Locking Strategy
Client Control Strategy
Dispatching Strategy (Reactive or multi-threaded)
Multi-threading Strategy
Obeserver Strategy For Federation
Scheduling Strategy
Filtering Strategy
Timeout Dispatching Strategy
Client Disconnection Strategy

Figure IV.12: RTES QoS Configuration Dimensions in Data Model

On the other hand, system functional aspects are captured through a set of standards-based *Component Deployment Plan Descriptors*, which describe the interaction among a set of CCM components. The *Component Deployment Plan Descriptors* can refer to any *CIAO Events Descriptors* and any elements defined in them through two generic, standards-based XML elements called `deployRequirement` and `InfoProperty`. The `Info-Property` element specifies which *CIAO Events Descriptor* files to use within this deployment plan, and the `deployRequirement` elements can specify which policies to be associated with which entities in the deployment plan, including *components*, *connections* and *ports*. Figure IV.13 shows an example where we associate a CCM event sink port with a particular event filter, which is defined in a separate XML file.

```
<connection>
  ...
  <deployRequirement>
    <resourceType>EventFilter</resourceType>
    <name>source_filter_id_01</name>
    <property>
      <name>EventFilter</name>
      <value>
          <type>
            <kind>tk_string</kind>
          </type>
          <value>
            <string>source_filter_id_01</string>
          </value>
      </value>
    </property>
  </deployRequirement>
  ...
</connection>
```

Figure IV.13: Example QoS Configuration for a CCM Connection

This declarative approach offers a much more powerful and flexible reconfiguration mechanism than traditional object-oriented approaches. For example, when some deployment and configuration concerns of publish/subscribe services in an existing DRE system need to be modified to accommodate a different set of system deployment and configuration requirements, *e.g.*, due to the changes of target infrastructure resources, operating conditions or mission goals, a system deployer can easily modify the *Component Deployment Plan Descriptors* by referring to another set of publish/subscribe configuration policy elements because all the XML-based configuration elements in *CIAO Events Descriptors* can be predefined and reused through the *lazy instantiation* [65] idiom.

```
/// Create one CIAOEventService object in
/// the container, which will be used to mediate
/// the communication of CCM events
module Deployment
{
  /// Extension interface pattern
  interface CIAOContainer : Container
  {
    ...
    readonly attribute
      ::Deployment::Properties properties;

    // installs event service
    CIAO::CIAOEventService install_es (
        in CIAO::EventServiceDeploymentDescription
        es_info)
      raises (InstallationFailure);
    ...
 };
```

Figure IV.14: IDL for CIAO Pub/Sub Service Deployment and Configuration

Run-time Model. To deploy the data model as described above into the target environment, we extend the standards-based *run-time model* of the OMG D&C Specification as a set of CORBA 2.x IDL interfaces to compose real-time publish/subscribe QoS concerns into the system functional concerns. Our extended run-time model applies the *Extension Interface* pattern [108] to make our implementation capable of handling various publish/-subscribe service QoS management yet are strictly compatible to the standardized interfaces. Figure IV.14 shows part of the CORBA 2.x IDL interfaces of the run-time model.

The D&C framework and tools we developed based on this run-time model are integrated as part of DAnCE, which consists of a set of daemon processes plus a utility program called *plan launcher*. The daemon processes include a global-level daemon process called the `ExecutionManager`, which acts as the central portal for the the deployment and configuration of different DRE systems within a particular domain. Also, there is another type of daemon process called `NodeManager`, which serves the deployment and configuration within an individual node.

All daemon processes can be deployed onto a set of distributed nodes and then cooperatively deploy the publish/subscribe services as an integral part of the standards-based deployment process, all driven by the *plan launcher* utility. The numbered bullets show the flow of control among different DAnCE entities, starting from a system deployer using the *plan launcher* utility to invoke the service on the `ExecutionManager`, which conforms to the standardized D&C process. The design goals and implementation of the meta-programmable architecture are summarized in Table IV.1.

To design and implement the meta-programmable architecture for publish/subscribe services for DRE systems, we address the following design goals:

Design Goal 1: Eliminate the need for manually writing code to bridge the CCM Eventtype with ORB publish/subscribe service event types.

Solution approach → Component Implementation Definition Language (CIDL) Compiler: We modified the CIAO CIDL compiler so it can automatically generate the servant code for each CCM component port, which handles many low-level tedious and error-prone details, such as registering a CCM Event valuetype factory with the ORB, marshaling and demarshaling different event types, and converting a CCM event type between publish/-service typed events. Moreover, to reduce the memory footprint of CCM components, we allow some of such code to be suppressed or conditionally generated through CIDL compiler command line options.

Design Goal 2: Eliminate the need for manually writing code to identify the event publishers and subscribers, and set up QoS configurations based on them, such as constructing event filters based on the event source ID.

Design Goal 1: Eliminate the need for manually writing code to bridge the CCM event type with ORB publish/subscribe service event types.	**Solution approach → Automatic Code Generation:** We enhanced the CIAO CIDL compiler to automatically generate the necessary servant code for each CCM component port, which handles many low-level tedious and error-prone details, such as registering CCM Event valuetype factory with the ORB, marshaling and demarshaling different event types, and converting CCM Event-type between publish/service typed events.
Design Goal 2: Eliminate the need for manually writing code to identify the event publishers and subscribers which are necessary for certain QoS configurations such as constructing event filters.	**Solution approach → DAnCE Orchestration:** When DAnCE performs deployment, it automatically generates unique identifiers for every event publisher and subscriber, and maps them to the specific event types of the corresponding object-oriented services. This will eliminate the dependencies between the event filters we constructed and the event sources we declared.
Design Goal 3: Eliminate the need for manually writing code to set up event channel federations, which involves tedious and error-prone details such as instantiating gateway objects, activating gateway endpoints, and binding them with event channels.	**Solution approach → Service-based Event Channel Federation** We developed a reusable *Event Channel Federation Service* within the CIAO-container framework to allow different event channel federation mechanisms to be pluggable. For example, a UDP Unicast based federation mechanism can be replaced easily with UDP Multicast based federation or CORBA IIOP based federation.

Table IV.1: Meta-programming Design Goals and Solutions

Solution approach → DAnCE: Since DAnCE has a global view of the entire system, it automatically generates unique identifiers for every event publisher and every event subscriber based on the component instance ID and event port ID string pair. This will eliminate the the dependencies between the event filters we constructed and the event sources we declared.

Design Goal 3: Eliminate the need for manually writing code to set up event channel federations, which involves a lot of tedious and error-prone details such as instantiating gateway objects, activating the endpoints of these objects, and binding them with event channels.

Solution approach → Service-based Event Channel Federation Mechanism We develop a reusable *Event Channel Federation Service* within the CIAO-container framework, which allows different event channel federation mechanisms to be pluggable based on specific target running environment and system requirements. For example, UDP Unicast based federation mechanism can easily be replaced with UDP Multicast based federation or CORBA IIOP based federation by configuring the descriptors based on the enhanced *data model*.

IV.3 Empirical Performance Evaluation

The success of QoS-enabled component middleware technologies to develop and deploy DRE systems depends on the real-time performance of the publish/subscribe mechanisms supported by them. This section provides empirical results for the container-managed CORBA real-time event service (RTES) integrated within our CIAO QoS-enabled component middleware. We choose RTES as a vehicle to evaluate our design because it is a mature real-time publish/subscribe implementation based on real-time CORBA and has been widely used in many DRE systems [87].

Our performance evaluation of container-managed RTES focuses on answering two key questions: (1) how well does the performance of container-managed RTES in CIAO middleware compare with that of the widely used CORBA 2.x RTES implementation in TAO object-oriented middleware, which is used as a baseline for RTES performance, and (2) how well does the container-managed RTES in CIAO scale with the different number of publishers and subscribers under different QoS configurations. We illustrate how the flexible configuration capabilities of CIAO's container-managed RTES can provide desired event delivery QoS without modifications to the component implementations.

IV.3.1 Experimental Testbed

All our benchmarks were conducted on ISISlab (`www.isislab.vanderbilt.edu`), which is a testbed of computers and network switches powered by Emulab software suite that can be arranged in many configurations. ISISlab consists of 6 Cisco 3750G-24TS switches, 1 Cisco 3750G-48TS switch, 4 IBM Blade Centers each consisting of 14 blades (for a total of 56 blades), 4 gigabit network IO modules and 1 management module. Each blade has two 2.8 GHz Xeon CPUs, 1GB of RAM, 40GB HDD, and 4 independent Gbps network interfaces. The underlying hardware used by ISISlab can be configured to provide a virtual network topology and configure various parameters of that network including link bandwidth capacities, node characteristics for use in routing, traffic shaping or traffic generation, and link error rates.

In our tests, we used up to 5 blade nodes. Each blade ran Fedora Core 4 Linux, version 2.6.16-1.2108_FC4smp. All our benchmark applications were run in the Linux real-time scheduling class to minimize extraneous sources of memory, CPU, and network load. For each test, we run the iteration at least 10,000 times.

IV.3.2 Experiment Approach

The most important metric for any publish/subscribe service is the *event latency*, *i.e.*, the time elapsed from when a publisher sends an event until the last subscriber interested in the event receives it. Compared with measuring the latency in the event delivery within a single node, measuring the latency for a distributed case where multiple nodes are involved is hard since events are delivered via a uni-directional flow of communication from publishers to subscribers. Event delivery time and jitter is comparable to the network propagation delay, because a distributed clock precision is bounded by the jitter [52]. Measuring the latency of event propagation, where event publisher and event subscriber are deployed in different nodes, is impossible. Fortunately, the latency for the centralized configuration can be measured directly using a subscriber located in the same node as the publisher and measuring the roundtrip delay as shown in Figure IV.15.

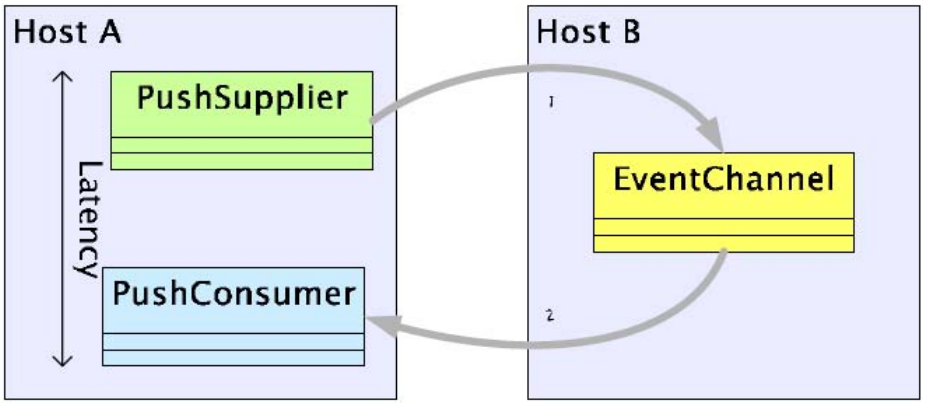


Figure IV.15: Experimental Setup to Measure Event Latency

The next section presents our experimental results with different event service QoS settings, illustrating their effects on the provisioning of event communication QoS for different DRE system deployment scenarios, including one-to-one, one-to-many, collocated deployment, distributed deployment, as well as varying event payload size. For each test, we run the iteration for 10,000 times, then measure the average latency.

IV.3.3 Comparing Performance of CIAO's Container-Managed RTES and TAO's RTES

In this test we measure the end-to-end latency introduced between publishers and subscribers. In order to measure the performance of CIAO's container-managed RTES, both the publishers and subscribers are developed as reusable CCM components, which can be deployed by DAnCE for different test cases. On the other hand, for TAO's RTES both the publishers and subscribers are developed in the form of CORBA objects. To ensure an accurate comparison between the CIAO container-managed RTES and TAO's RTES implementations, we designed both tests using the same set of QoS configuration settings on publishers, subscribers and event channels. The subscriber does nothing with the events it receives other than storing the data in a preallocated array. Both tests are configured to measure the end-to-end publish/subscribe latency using the IIOP communication mechanism, which uses point-to-multipoint event delivery rather than IP multicast. We measure the end-to-end latency based on different event payload size as well as increasing number of subscribers.

In many real-world DRE systems, it is common that multiple components are deployed within the same component server but still communicate with each other through CCM ports, *i.e.*, facets, receptacles, event sources, and event sinks. As a result, it is important to measure the performance of process-collocated cases. Our performance evaluation in this dimension is described below.

Latency Results for Process Collocated Event Processing. In this test all the event publishers, subscribers, and the event channel are collocated in the same process, which eliminates effects of ORB remote communication overhead. In the container-managed RTES case, both the publisher and subscriber components are deployed into the same container which in turn is hosted in a single CIAO component server. The end-to-end latency is determined by the publisher sending out the timestamp right before the `push` call and subsequently the subscribers calculating the difference between the timestamp at the publisher side and the subscriber side. We also use variable-sized octet sequence as the payload so that we can easily control the volume of the payload.

One important fact worth mentioning concerns the event data type we used in TAO's RTES benchmarking test. Since all the event source and event sink ports of CCM components are defined as `Eventtype`, which is a specialized `valuetype`, it is unavoidable to eliminate the additional overhead incurred due to marshaling/demarshaling of such a data type. To ensure a fair comparison between the performance of TAO's RTES and CIAO's container-managed RTES, we send `valuetype` data in both test cases, and make the octet sequence payload as the member of this data type.

Figure IV.16 shows the latency results for *point-to-point* (*i.e.*, one-to-one) configuration.

In this test, there is only one publisher and one subscriber both of which are collocated within the same event channel in a single OS process.

Figure IV.17 shows the latency results for the *one-to-many* case. In this test, we increase both the number of subscribers and the payload size to see how the end-to-end latency is affected. For the one-to-many case, the measured end-to-end latency is determined by the publisher sending out the timestamp then calculating the difference between the timestamp at the publisher side and the *last* subscriber that receives it.

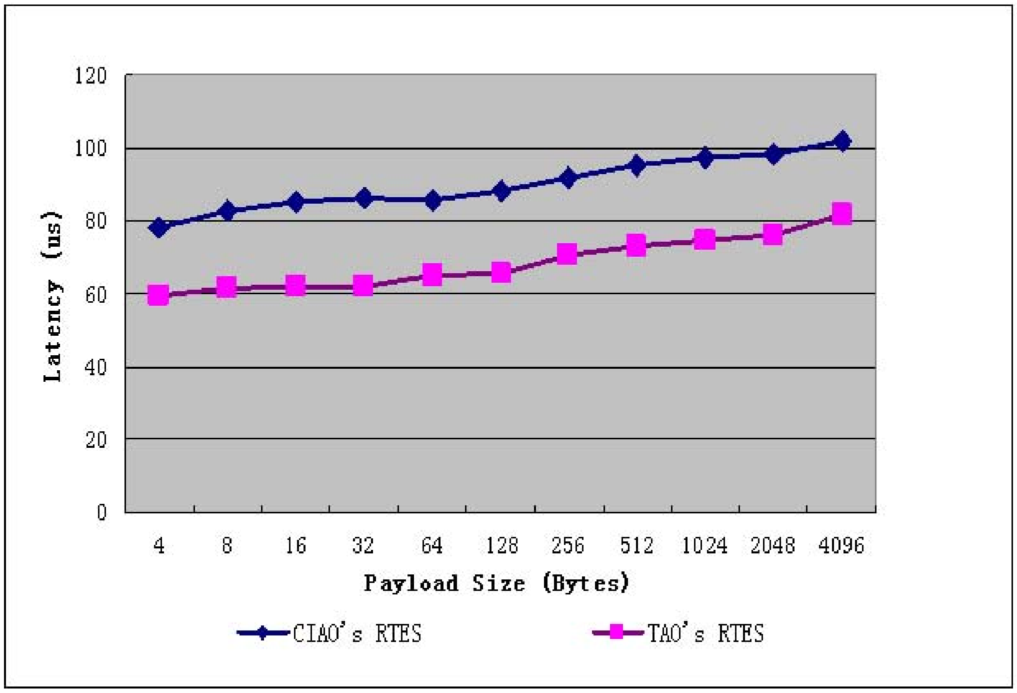


Figure IV.16: Collocated Point-to-Point Latency

Analysis. Our collocation experimental results indicate that the event dispatching overhead in CIAO's container-managed RTES incurs about 20$\sim$25% latency performance overhead consistently over the TAO's RTES implementation. This overhead is primarily due to two reasons: (1) in contrast to programming with CORBA RTES, where the publisher *CORBA objects* and the subscriber *CORBA objects* can directly interact with the event channel as

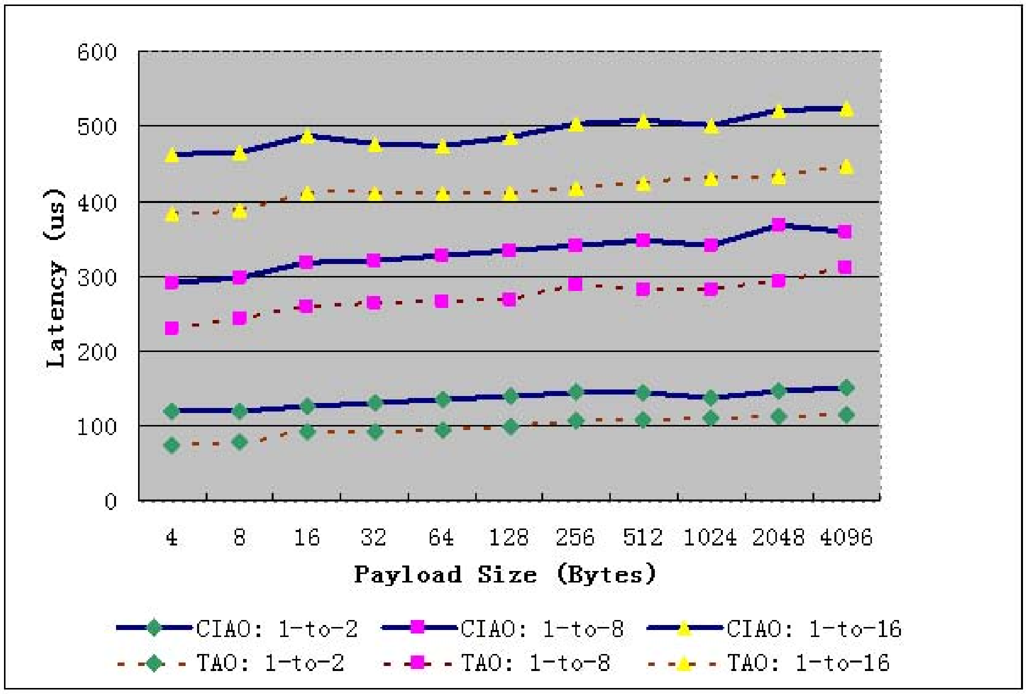


Figure IV.17: Collocated One-to-Many Latency

RTES clients, the OMG CCM standard introduces multiple indirections involving *component executors* that must communicate through their individual *contexts*, which in turn interact with the *component servants* and CORBA RTES, and (2) the additional indirection introduced with the higher-level container mediation interface as described in Section IV.2.

While it is inevitable to avoid the overhead caused by the indirection defined in the OMG CCM standard without breaking standards-based interfaces, we applied process-collocation optimization to CIAO's implementation to improve the performance and predictability of collocated component communication. The process-collocation optimization we conducted improves the performance and predictability for objects that reside in the same address space as the servant implementation, while maintaining locality transparency.

Our process-collocated experimental results demonstrate that the performance optimizations of object-oriented real-time event dispatching are preserved within a container-based solution. It is also interesting to observe that as the number of subscribers increase, the increase in latency is less than linear in both TAO's CORBA RTES implementation and

CIAO's container-managed RTES implementation, due in large part to the Handle/Body idiom used to optimize the processing of CORBA `Any` data types in TAO's RTES implementation [105], which is inherited by CIAO RTES. This idiom presents multiple logical copies of the same data while sharing the same physical copy.

CIAO's container-managed RTES implementation still preserves the performance of this optimization, which makes our solution scalable and hence suitable for large-scale DRE systems where there exist a large number of subscribers.

Latency Results for Remote Event Processing. [1] In this test, we run the experiment by creating two different processes within a single node to allow the events to be sent remotely. The event publisher and the event channel are collocated in one process, while the subscribers are in the other process. Figure IV.18 and Figure IV.19 show the latency results of *point-to-point* and *one-to-many* configurations, respectively.

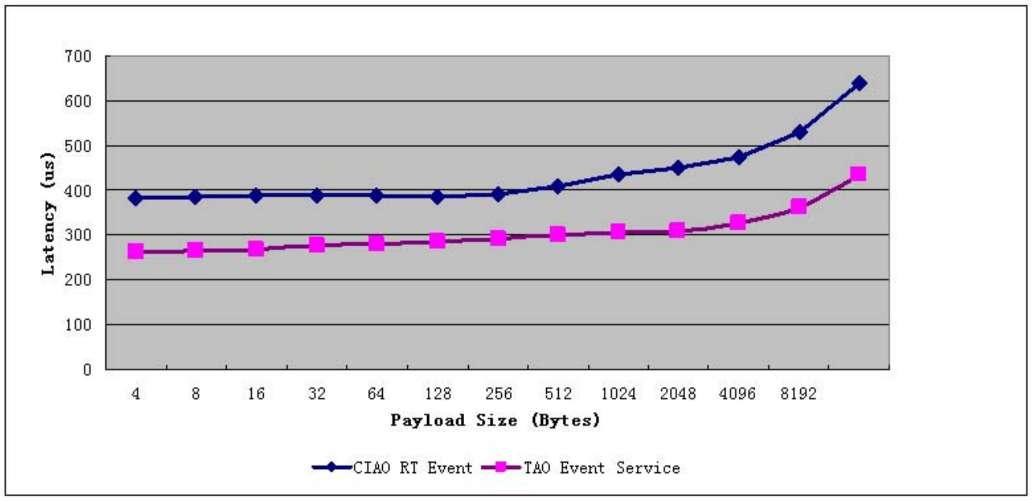


Figure IV.18: Two Process Point-to-Point Latency

Analysis. The results indicate that the remote event processing latency in both CIAO's container-managed RTES and TAO's RTES are much higher than that of process-collocated cases (both incurring about 6∼10 times overhead) due to the process boundary crossing,

[1]In this test configuration, a "remote" event is one intended for a subscriber located in the other process.

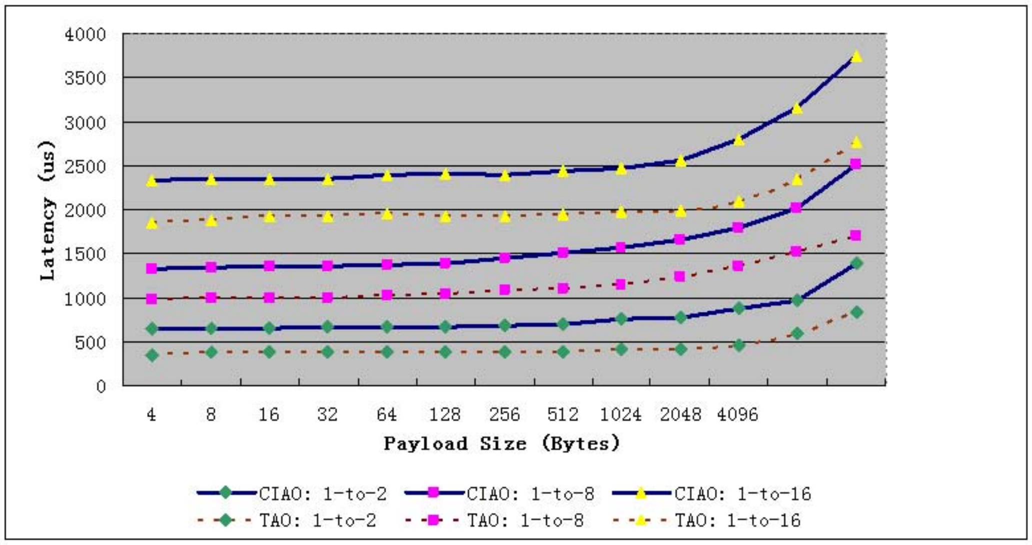


Figure IV.19: Two Process One-to-Many Latency

though they are close to the performance of a remote operation invocation. With increasing number of event subscribers and payload size, the results still show that the event dispatching performance in container-managed RTES in CIAO consistently has about 70∼80% performance of TAO's RTES in all test cases. In conjunction with the results of process-collocated tests, these results further confirm that the CIAO container-based RTES solution has predictable performance which is comparable with TAO's RTES, and is thus suitable for DRE systems.

IV.3.4 Evaluating Scalability of CIAO's Container-Managed RTES

Another important characteristic of real-time publish/subscribe services is scalability. As discussed in Section IV.2, CIAO's container-managed RTES provides support for both single event channel dispatching as well as federated event channels across multiple containers. To evaluate scalability in this test we measure the throughput of CIAO RTES' event dispatching under different configuration settings. Our goal is to demonstrate the efficiency of remote event processing using federated event channels where there are multiple remote subscribers or multiple distributed nodes. As a result, we split our tests into two parts, one with multiple subscribers on the same node but different processes, and one with multiple subscribers distributed on different nodes, and observe how different configurations can affect scalability.

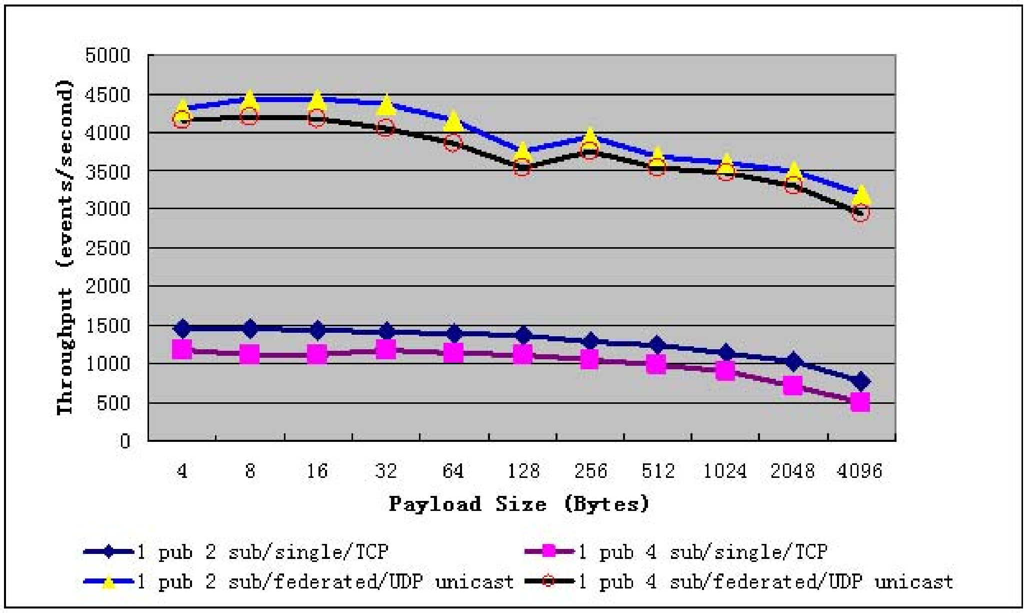


Figure IV.20: Host Collocated Throughput Measurement: All Subscriber on a Different Component Server

Scalability Results for Host Collocated Remote Event Processing. This test measures the throughput of CIAO's container-managed RTES where all the subscriber components are hosted in a remote container, while the publisher component is hosted in another container. Both containers are hosted in different CIAO component server processes but on the same node. We send a fixed number of events on the publisher side and measure the rate of the number of events dispatched per second. Figure IV.20 shows the throughput results for this configuration. The bottom two curves show the throughput results when there is no container-managed event channel configured on the subscriber side, and all the subscriber components are directly connected to the real-time event channel hosted in the publisher side container. The upper two curves show the throughput results where both

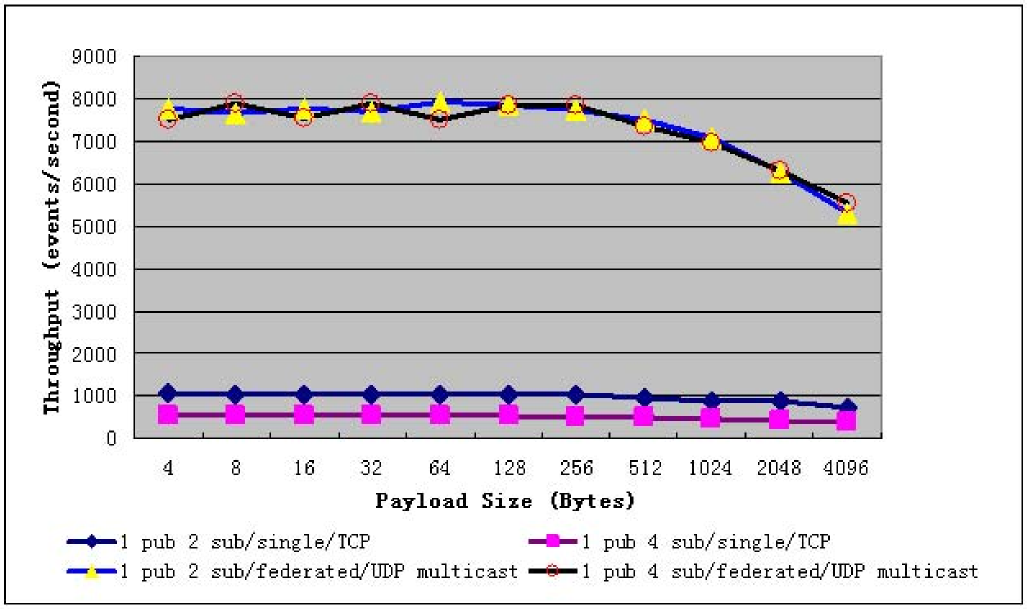


Figure IV.21: Distributed Throughput Measurement: Each Subscriber on a Different Node

containers have a real-time event channel configured locally and they are federated through UDP unicast gateway objects [88].

Analysis. For many DRE systems it is quite common that there are many subscriber components collocated in a single process but remote to the publisher components. Our results indicate that a federated group of event channels can improve the throughput performance dramatically because publishers and subscribers connect only to their local event channel, while event channel instances talk to each other via the CORBA bus. When multiple subscribers are collocated in the same process, instead of making multiple remote calls (one for each subscriber), only one remote call is necessary from the publisher process to the subscriber process. This configuration improves the total throughput by reducing the average latency for all the subscribers in the system because subscribers and publishers exhibit locality of reference, *i.e.*, most subscribers for any event are in the same domain as the publisher generating the event. We can imagine that in a networked environment, when multiple remote subscribers are interested in the same event only one message is sent to each remote event channel thereby also minimizing the waste of network resources.

Scalability Results for Distributed Event Processing. This test measures the throughput of CIAO's container-managed RTES when the subscriber components are hosted in different containers on different physical nodes, while the publisher component is hosted in another remote node. Again, we send a fixed number of events on the publisher side, and measure the dispatching rate of number of events per second. Figure IV.21 shows the throughput results for this configuration. The bottom two curves show the throughput results when there is no container-managed event channel configured on the subscriber side, and all the subscriber components are directly connected to the real-time event channel hosted in the publisher's side container through TCP-based IIOP protocol. The upper two curves show the throughput results where all the containers have their own real-time event channel configured locally and they are federated through UDP multicast gateway objects [88].

Analysis. The results indicate that the throughput performance improves substantially due to the UDP multicast federation settings among different container-managed event channels distributed across different nodes. Using a multicast protocol can avoid duplicate network traffic, and offload event dispatching workload from the CPU to the network infrastructure. The use of multicast is ideal for the scenario where many subscribers are distributed across different nodes in a networked environment since increasing the number of nodes distributed across the network has little effect on UDP multicast. Since currently TAO's RTES only supports unreliable multicast, we are working on a reliable multicast solution within TAO's RTES design so components using CIAO's container-managed RTES can take advantage of this feature.

IV.4 Summary

R&D over the past decade on object-oriented standards-based middleware, such as real-time CORBA and various real-time publish/subscribe services built atop it, has demonstrated its effectiveness in developing and supporting QoS requirements of DRE systems. However, building flexible and robust software for large-scale DRE systems with such an approach is still challenging because the need for determinism and predictability often results in tightly-coupled designs. For instance, conventional mission-critical applications built with object-oriented middleware consist of closely integrated responsibilities; for each component, application developers still have to write code to handle multiple aspects, such as real-time event dispatching, scheduling, connection management and periodic event processing. Tight coupling often yields inflexibility and thus can substantially increase the effort and cost of integrating new and improved system features.

Component middleware has already received widespread acceptance in the enterprise business and desktop application domains. However, developers of DRE systems have encountered limitations with the available component middleware platforms, such as the CCM and J2EE. In particular, component middleware platforms lack standards-based real-time publish/subscribe communication mechanisms that support key QoS requirements of DRE systems, such as low latency, bounded jitter, and end-to-end event propagation. This chapter provides guidance to establish the feasibility of integrating mature object-oriented real-time publish/subscribe services in QoS-enabled component middleware architectures. In particular, our results indicate that the approach of using container-managed real-time publish/subscribe service not only provides the most flexible way to developing DRE systems by taking advantage of standards-based component models, and the deployment and configuration model, but also provide predictable end-to-end performance and scalability.

CHAPTER V

TECHNIQUES FOR ENSURING PREDICTABILITY OF DEPLOYMENT AND CONFIGURATION

Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources or QoS requirements is an important and challenging R&D problem. Systems with such characteristics are called *open* DRE systems [29] since they operate in an open environment and must be prepared to accommodate changing operating conditions or requirements, such as power levels, CPU/network bandwidth or mission modes. Examples of open DRE systems include shipboard computing environment [109], and intelligence, surveillance and reconnaissance systems [111].

Open DRE system are often large and complex, *e.g.*, a shipboard computing system may consist of thousands of software components that run a wide range of missions, such as ship navigation, ship structural health monitoring, vision-based object tracking and object characterization. To manage the overall complexity of such systems, the missions are often decomposed into many domain-related tasks that can be modeled as *operational strings* [59], which are assemblies of software components that capture the partial order and workflow of a set of executing software capabilities for particular domain tasks.

Operational strings have the following properties that make them useful building blocks for open DRE systems:

- **Distributable**, *i.e.*, operational strings can be deployed onto multiple nodes of the target running environment, and different components in operational strings can communicate remotely with each other.

- **Concurrent**, *i.e.*, operational strings can run concurrently in the same target environment and share many system resources, such as CPU, memory, and network bandwidth.

- **On-demand**, *i.e.*, operational strings can be dynamically populated at system run-time and then deployed into the target running environment on-demand to accommodate changing mission goals.

- **Cooperative**, *i.e.*, to achieve certain mission goals different operational strings can cooperate with each other through their *ports*, which delegate to the ports of monolithic components that consist of the operational strings.

- **Prioritized**, *i.e.*, different operational strings can be assigned with different priorities by system architects or online planners to reflect their importance to certain mission tasks or to the overall system.

The dynamic nature of open DRE systems requires the deployment and configuration (D&C) of scores of operational strings at run-time to ensure that executing systems keep in sync with changing mission goals and resource availability. The run-time management of operational strings in DRE systems is hard since the D&C framework must be scalable and predicable. It is therefore essential that D&C frameworks be able to dynamically deploy and configure operational strings in a timely and predictable manner.

In complex DRE systems, many operational strings may be deployed dynamically, *e.g.*, in response to mission mode or environmental changes. If dependencies exist among these operational strings, *deployment priority inversions* can occur at run-time. A deployment priority inversion occurs when a higher priority operational string cannot be deployed before lower priority operational string(s) because of the dependencies between them. Existing D&C frameworks [20, 21, 93] only consider the dependency between operational strings and ignore their priorities, which can cause unbounded deployment priority inversions for DRE systems.

Solution Approach → Partial Priority Inheritance via Graph Recomposition.

To address the challenges of open DRE systems described above, we developed a technique based on an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm analyzes the dependencies between operational strings and removes all the dependencies from higher priority operational strings to lower priority ones by promoting[1] components from lower priority operational strings to higher priority ones. By applying our technique, a D&C framework can avoid potential priority inversions when multiple operational strings are deployed at run-time.

Figure V.1 shows the three three main steps of our approach:

- Step 1 converts a deployment descriptor (which contains metadata describing a set of operational strings) into an in-memory *directed graph* representation. Each vertex in the graph represents a component in the operational string and each edge represents a connection between two components.

- Since a deployment plan may have multiple operational strings with different priorities having dependencies among each other, step 2 analyzes the dependency relationship between all the operational strings by perform a graph-based algorithm called *partial priority inheritance via graph recomposition*. This algorithm removes all the priority inverted dependencies between operational strings by promoting component(s) from the lower priority operational string to the higher priority string.

- After graphs are recomposed, step 3 converts them back to deployment descriptor format and fed to the D&C framework for deployment. For the operational strings with dependencies with each other, the D&C framework can then deploy the operational strings from the highest priority to the lowest priority.

When a DRE system has many operational strings with complex dependencies it is

[1]In the context of this paper, *promoting* a component means that before this component is deployed it is temporarily moved from a lower priority operational string to a higher priority operational string for deployment purpose only.

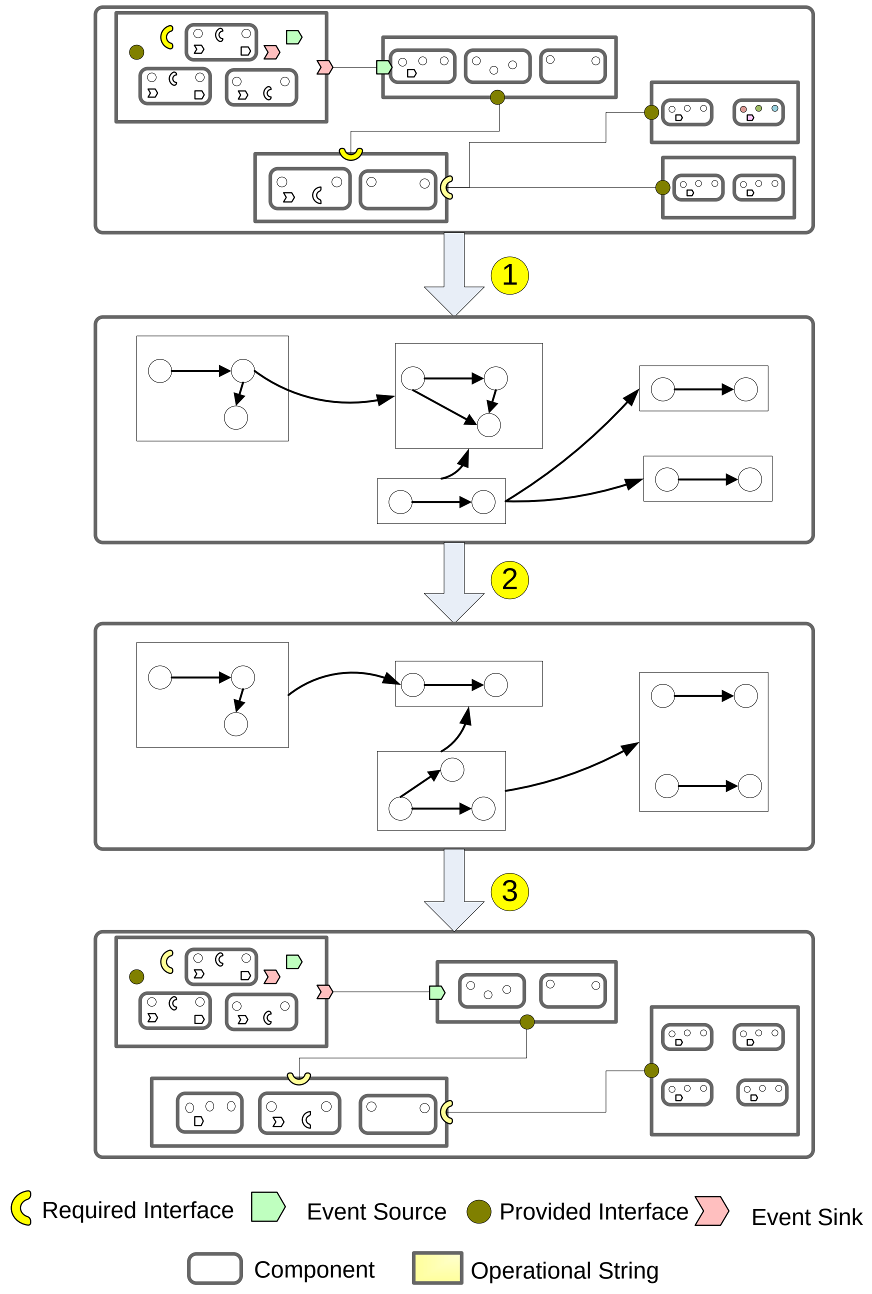


Figure V.1: Overview of Solution Approach

hard to determine manually which components in which operational strings should be promoted and which priority to promote to. This chapter therefore makes the following three contributions to the research on D&C for component-based DRE systems:

- Analyze dependency relationships among operational strings to determine how each relationship can affect deployment predictability.
- Present a multi-graph algorithm called "partial priority inheritance via graph recomposition" to avoid deployment priority inversion.
- Empirically evaluate the multi-graph algorithm to determine how effective it is on a representative DRE system.

V.1 Challenges of Ensuring Predictability of D&C

This section describes different configurations of operational strings in DRE systems that can cause deployment priority inversion to occur due to the dependencies among the strings. To make our discussion concrete, we use NASA's Magnetospheric Multi-Scale (MMS) mission system [119] as a case study. We first present the case study and then identify key challenges that must be addressed to ensure D&C predictability for the case study.

V.1.1 Overview of NASA MMS Mission System

The NASA Earth Science Enterprise's MMS mission system system uses five satellites with multiple sensors on each satellite to perform solar-terrestrial probe task. The satellites orbit the earth in formation and collect electromagnetic and particle data in the earth's magnetosphere. The MMS mission operates in three data modes: *slow*, *fast*, and *burst*. These data modes may also include different goals, orbits, and data priorities. Each satellite must be capable of determining the necessary task sequences to achieve prescribed goals based on the current environmental and system conditions, as well as revising task sequences in response to changing conditions.

To achieve autonomy, an automated planner is deployed within the MMS system to handle autonomous mode changes driven by the satellite position and the results of analyzing collected data. The task sequences are implemented by components for coordinating the trajectory and orientation of satellites, sensor selection and data collection for individual satellites, and data integration and compression to create telemetry streams that are beamed down to earth stations.

Figure V.2 shows three operational strings that a planner generates for a mission task of one of the satellites. Each operational string has different deployment priority (*i.e.*, high,

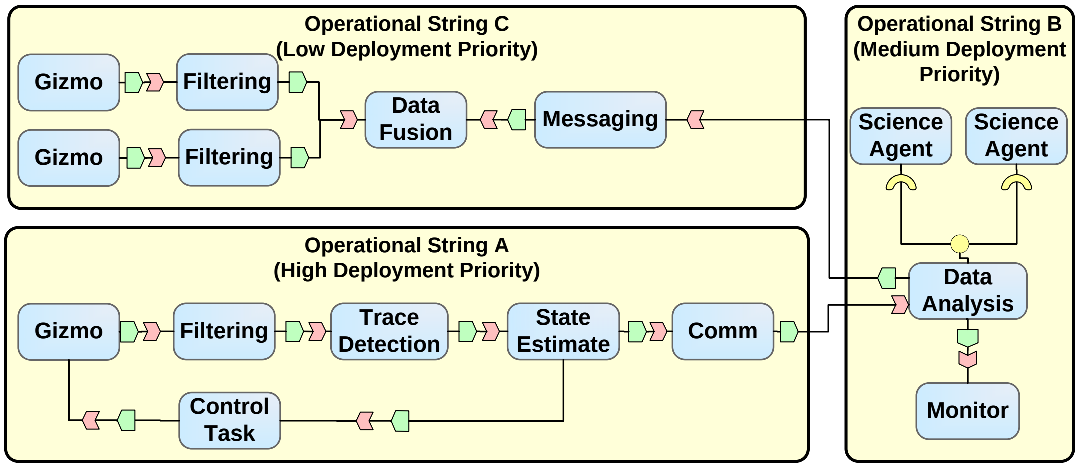


Figure V.2: Operational Strings Generated by Planner

medium, and low) that are determined by how each operational string is accessed by the overall MMS system. The three operational strings are briefly described as follows:

- **Operational string A** defines a mission-critical task that collects field data when a satellite moves to particular locations. To ensure this task is performed properly, the operational string must be deployed as fast as possible to avoid loss of data.

Operational string *A* can store the collected data in its own data store, but can also send the data to other operational strings through its event sources.

- **Operational string B** is designed for a domain-centric data analysis. Different scientific analysis tasks can be configured through the facets of components of this operational string. For example, Science Agent components can be configured to achieve scientific objectives, such as analyzing models of complex phenomena like extended weather forecasting.
- **Operational string C** is for less essential data analysis task and can collect auxiliary field data, such as Sun zenith, satellite view zenith [96], which can serve as additional input for analysis. This operational string only operates occasionally, *e.g.*, when the data analysis component in operational string *B* explicitly issues a request to request such data as additional input for scientific analysis models. The components in operational string *C* are driven by events exchanged through their event sources and sinks.

Operational strings are organized from domain perspective, *e.g.*, each operational string is designed to accomplish certain domain tasks, such as collecting certain field data, or perform certain analysis on different data models. In our MMS scenario, operational string *A* services (*e.g.*, collecting essential field data for scientific analysis) are most important for the MMS system, so it has the highest deployment priority among the three operational strings. Conversely, operational string *C* has the lowest deployment priority among the three since it is designed as a less essential service, *i.e.*, collecting auxiliary data only when necessary. Finally, operational string *B* is designed to have medium deployment priority because its scientific analysis role is less important than operational string *A*, but more important than operational string *C*. Operational string *B*, however, needs to send events to operational *C* to notify it to collect auxiliary field data and perform analysis when necessary. The deployment priorities of operational strings are not the same as execution priorities because the latter deals with real-time QoS at run-time.

As shown in Figure V.2, there are two dependencies between operational strings: from *A to B* and from *B to C*. These dependencies cross the boundary of an individual operational string. We therefore call them *external dependencies*, in contrast to those dependencies within an operational string.

V.1.2 Challenges of Ensuring D&C Predictability in the MMS Case Study

Below we describe four challenges that arise when operational strings are deployed dynamically in open DRE systems, such as the NASA MMS mission case study described above.

Challenge 1: Avoid deployment priority inversion between two operational strings. In conventional D&C technologies, such as the OMG D&C specification [79, 93], when a component of an operational string has a connection (either facet/receptacle or event sink/source) to another component in a separate operational string, an *external reference* must be specified to indicate the remote component and the provided port in the other operational string upon which it depends. To deploy this operational string successfully, the external reference endpoint of the other operational string must be activated before the deployment of source operational string can occur. When such a dependency is from a higher priority operational string to a lower priority operational string, however, the low priority operational string must be deployed *before* the high priority operational string can be deployed to avoid deployment failure caused by the dependency, which results in a priority inversion at deployment-time.

For example, in our MMS system case study described in Section V.1.1 the dependency from operational string *B* (medium priority) to operational *C* (low priority) can cause a deployment priority inversion between operational strings *B* and *C*. This dependency requires the deployment of operational string *C* before operational string *B* to resolve the dependency. Not all components in operational string *C* need be deployed to resolve the external dependency between *B* and *C*.

Subsection V.2.2.1 describe how we address this challenge by promoting components from the lower priority string to the higher priority string.

Challenge 2: Avoid deployment priority inversion propagation effect. A more general priority inversion situation involves multiple operational strings. In this case, to resolve a dependency from a higher priority string to a lower priority string, not only must the lower priority string be deployed before the high priority operational string, but also the operational strings the lower priority string depends on. When these operational strings have lower priority than the high priority string, however, deployment priority inversion will occur between operational strings.

For example, in our MMS system case study operational string *A* has a high priority and an external dependency on operational string *B*. More specifically, it is the `Data Analysis` component of operational string *B* that *A* depends on. The `Data Analysis` component further depends on the `Messaging` component in operational string *C*, however, which can cause another deployment priority inversion between *A* and *C*.

Subsection V.2.2.2 describes how we address this challenge by recursively tracing dependencies from the high priority string to all lower priority strings by finding the transitive closure.

Challenge 3: Avoid deployment failure when circular dependency exists among multiple operational strings. Conventional D&C techniques [79, 93] will fail if a circular dependency exists among multiple operational strings. The central, coordinated and phased deployment technique applied in conventional D&C technologies can effectively address the circular dependency problem within a single operational string. Conventional D&C technologies cannot handle cases, however, where a D&C request involves multiple operational strings and if a circular dependency exists among these operational strings.

Limitations with circular dependencies arise because the deployment of an operational string is treated as an indivisible process, *i.e.*, conventional D&C technologies treat operational strings as primitive units. If a dependency exists between two operational strings, therefore, one must be deployed before another to resolve such dependency requirements. Such treatment, however, makes it hard to handle circular dependencies among different operational strings.

For example, in our MMS system case study a circular dependency will exist between *A, B, and C* if the less essential operational string *C* depends on operational string *A* because it needs to access a type of service provided by *A*. Since conventional D&C approaches treat each operational string deployment as an indivisible process, such deployments cannot be handled properly.

Subsection V.2.2.3 describes how we address this challenge by promoting components in the circular dependency trace among operational strings.

Challenge 4: Improve the overall utility of the operational strings being deployed. The dynamic nature of open DRE systems require on-demand deployment of many operational strings that cooperate with each other to ensure the system is kept in sync with changing mission goals or environmental changes. Since multiple operational strings with different importance to the entire DRE system may be deployed at the same time, the goal of a D&C framework is to deploy these operational strings in an effective way to improve the overall QoS of DRE systems in the following two dimensions:

- Since operational strings with the highest priority are the most important to the entire DRE system, these operational strings should be deployed as early as possible to ensure the DRE system responsiveness due to the changing environment or mission modes. For example, in our MMS system case study, operational string *A* should always be deployed immediately since it has the highest priority.

- Since each operational string has a utility value associated with it, a D&C framework should try to finish deployment of each individual operational string as early as possible by taking its utility value into account. For example, in our MMS system case study there is a dependency from operational string A to operational string B. Although the deployment of operational string A should finish first due to its higher priority, the time to finish deploying operational string B is also a contributing factor to the overall system utility and should thus be considered.

Subsection V.2.2.4 describes how we address this challenge by selectively applying the PARIGE algorithm to the input of operational strings.

V.2 A Partial Priority Inheritance via Graph Recomposition Algorithm

This section describes how we resolved the challenges described in Section V.1.2 using an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm converts each operational string into a graph, where each vertex and edge of the graph represent a component and a connection/dependency between two components, respectively. If there is an external dependency between operational strings, then the graph converted from one operational string will have a special type of vertex that represents the external dependency. This special vertex type contains information about the actual refereed operational string and the component in the operational string that it depends on.

The PARIGE algorithm promotes components from one graph to another based on operational string characteristics, including their priorities and their dependency relationships with other operational strings in the same deployment request. After graphs for all the operational string are recomposed to account for the component promotion, a new set of operational strings will be populated from these recomposed graphs. These new strings avoid deployment priority inversion between operational strings and break the circular dependency among all the operational strings. These new set of operational strings can then be deployed by conventional D&C technologies, such as J2EE or OMG D&C models.

To demonstrate the effectiveness of PARIGE, we integrated the algorithm into the central coordinator `ExecutionManager` of OMG D&C model to perform experimental analysis. The `ExecutionManager` runs as a daemon and is used to manage the deployment process for a domain. In accordance with the D&C specification, a *domain* is a target environment composed of *nodes*, *interconnects*, *bridges*, and *resources*. An `Execution-Manager` plays the role of central coordinator that manages the nodes in the DRE system environment. On each node, a **NodeManager** runs as a daemon process and manages the deployment of all components that reside on that node, irrespective of which application they are associated with.

V.2.1 Overview of the PARIGE Algorithm

Although the PARIGE algorithm recomposes operational strings by promoting components from one operational to another, it has also the following properties that makes it well-suited for D&C of DRE systems:

1. The PARIGE algorithm does not affect the functional behavior of component-based DRE systems. Component-based DRE systems are deployed in the form of operational strings that consist of multiple monolithic components connected with each other via their ports. Two operational strings can have dependencies with each other. A dependency between two operational strings is essentially a connection from a monolithic component in one operational string to a port of a monolithic component in the other operational string.

The PARIGE algorithm evaluates the component dependency relationships and their priorities and recomposes these operational strings to avoid deployment priority inversion. From the perspective of *all* operational strings to be deployed, however, the individual monolithic components and their connections among each other are not modified by the algorithm. In particular, the effect of the PARIGE algorithm on operational string recomposition is only visible for the D&C framework, which does not affect the running system's functional behavior. This algorithm thus does not affect the functionally of operational strings because the topology of all the operational strings (including all the monolithic components and connections) that fulfills functional behavior of the system remains unchanged.

2. The PARIGE algorithm does not affect the QoS behavior of operational strings. When components are promoted from a lower priority operational string to a higher priority operational strings, the priority of the components is also bumped up to match the priority of the higher priority string, which is essential for a task to avoid priority inversion at deployment-time. Since the PARIGE algorithm only promotes components that one or more higher priority operational strings have dependencies on at deployment time, it does not change the actual real-time priorities or other real-time QoS configurations designed for run-time. Therefore, the PARIGE algorithm does not affect the QoS behavior of operational strings.

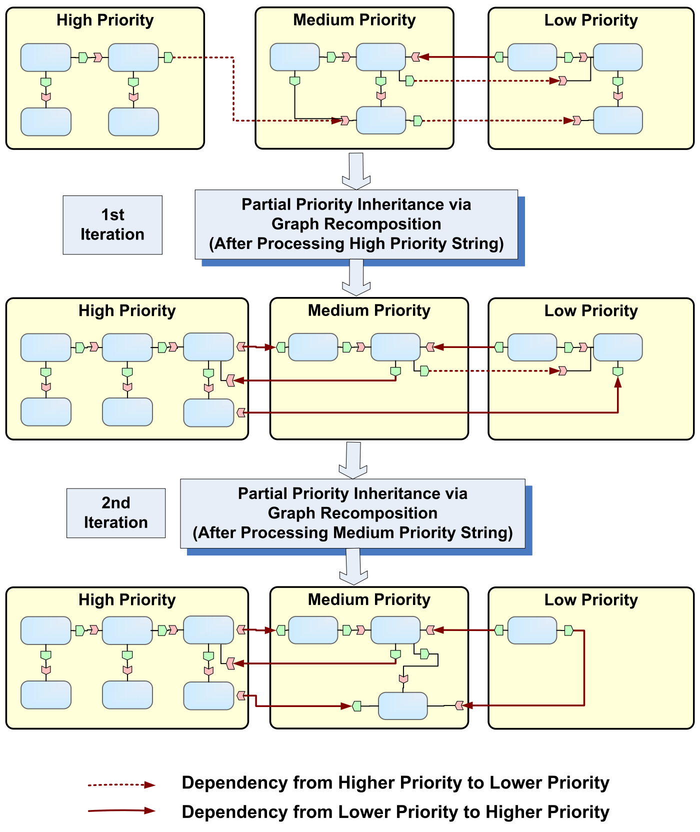

High Priority
Medium Priority
Low Priority
1st Iteration
Partial Priority Inheritance via Graph Recomposition (After Processing High Priority String)
High Priority
Medium Priority
Low Priority
2nd Iteration
Partial Priority Inheritance via Graph Recomposition (After Processing Medium Priority String)
High Priority
Medium Priority
Low Priority
Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.3: PARIGE Algorithm in Action

Figure V.3 presents an overview of the PARIGE algorithm by showing an example with three operational strings having priorities high, medium, and low.

The dotted and solid arrows represent dependencies between operational strings. In particular, the dotted arrows in the figure represent priority inverted dependencies, *i.e.*, dependencies from higher priority operational strings to lower priority operational strings. Likewise, the solid arrows represents external dependencies without causing priority inversion.

The numbered vertices in Figure V.3 denote the vertices promoted from one graph into another. For example, in the first iteration of the algorithm, one vertex is promoted from the medium priority operational string to the high priority operational string and another vertex is promoted from the low priority operational string to the high priority string. In the second iteration, another vertex is promoted from the low priority operational string to the medium priority string.

The PARIGE algorithm recomposes the graphs by parsing the input set of graphs and removing dotted arrows by promoting some component(s) from a lower priority operational string to a higher priority string. This process may introduce some new dependencies between operational strings due to component promotion. The algorithm, however, only introduces solid arrows, *i.e.*, only dependencies from lower priority operational strings to higher priority strings exist after the recomposition.

When the algorithm finishes, all dotted arrows in the graphs will be removed and there will be no dependencies from higher priority operational strings to lower priority operational strings. As a result, both priority inversions at run-time and deployment-time can be avoided. Moreover, all circular dependencies among operational strings, if any, will also be removed in the recomposition process.

V.2.2 Analysis of Operational String Dependencies with Deployment Priority Inversions

The goal of the PARIGE algorithm is to remove *all* dependencies from higher priority operational strings to lower priority operational strings. To accomplish this, the algorithm starts with the operational string having the highest priority and processes all the external dependencies of this operational string. After all external dependencies from the highest priority operational string are removed, the algorithm then processes the operational string with the next highest priority. When multiple operational strings have the same priority, we apply the following tie-breaking policies sequentially: (1) evaluating the second metric of each operational string, if given, (2) evaluating the number of external dependencies to the same priority operational strings and treating the operational string with the least number of external dependencies as the higher priority than others, and (3) breaking the tie randomly if such a tie still exists.

When processing an external dependency from a higher priority operational string to a lower priority operational string, the algorithm must trace the dependency into other operational strings and promote components from them if the lower priority operational string has dependencies to them. For example, if a high priority operational string depends on a component X in a medium priority operational string, and if component X also has dependency on a component in a low priority operational string, then the component in the low priority string also must be promoted into the high priority string.

We define a *dependency trace* as a totally ordered sequence S. Each element in the sequence is a component of an operational string that has a priority value associated with it. The starting element of the sequence is the source component of the external dependency of interest. The PARIGE algorithm analyzes all the dependency traces in the operational strings and recomposes the operational strings based on dependency trace characteristics. To analyze the dependency trace we classify the operational dependency relationships into the three categories described below, which will be used to address the first three challenges in Section V.1.2.

V.2.2.1 Handling Challenge 1 → Promotion of Components Between Two Operational Strings (Non-circular)

In this case, a dependency occurs between two operational strings, where a high priority operational string has a dependency on a lower priority operational string, as shown in Figure V.4. We assume no circular dependency exists in this case (circular dependencies will be discussed in Section V.2.2.3).

As shown in Figure V.4, the unique characteristic of this category is that the dependency trace does not cross the boundary of the lower priority operational string. Since no other

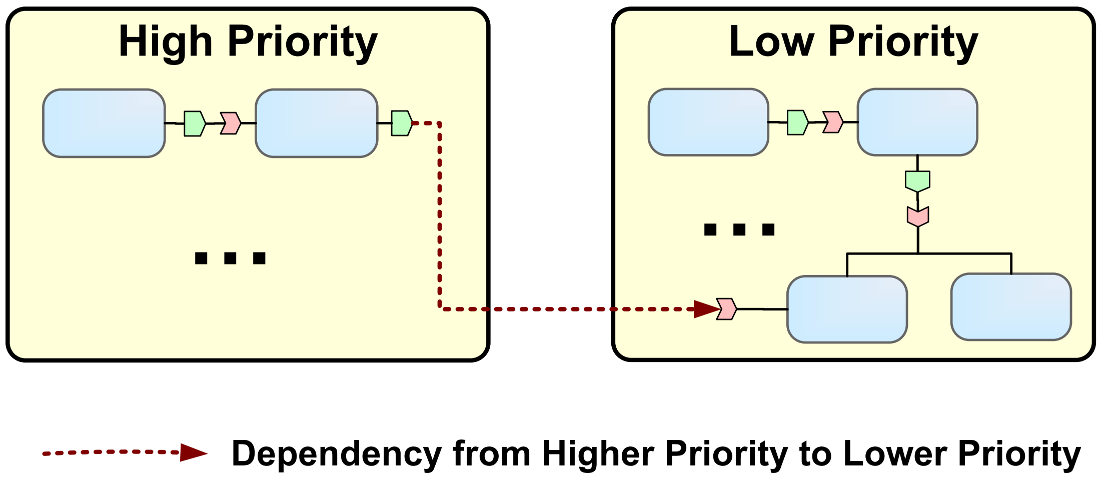


Figure V.4: A Dependency Trace Spanning Two Operational Strings Only

operational strings are involved besides the two operational strings of interest, removing such a priority inverted external dependency only requires promoting all components in the dependency trace from the lower priority operational string to the high priority one.

In the context of MMS case study, a priority inverted external dependency exists between operational string *B* and operational string *C*, as illustrated in the upper half of

Figure V.5. Our solution removes this priority inverted external dependency by promot-

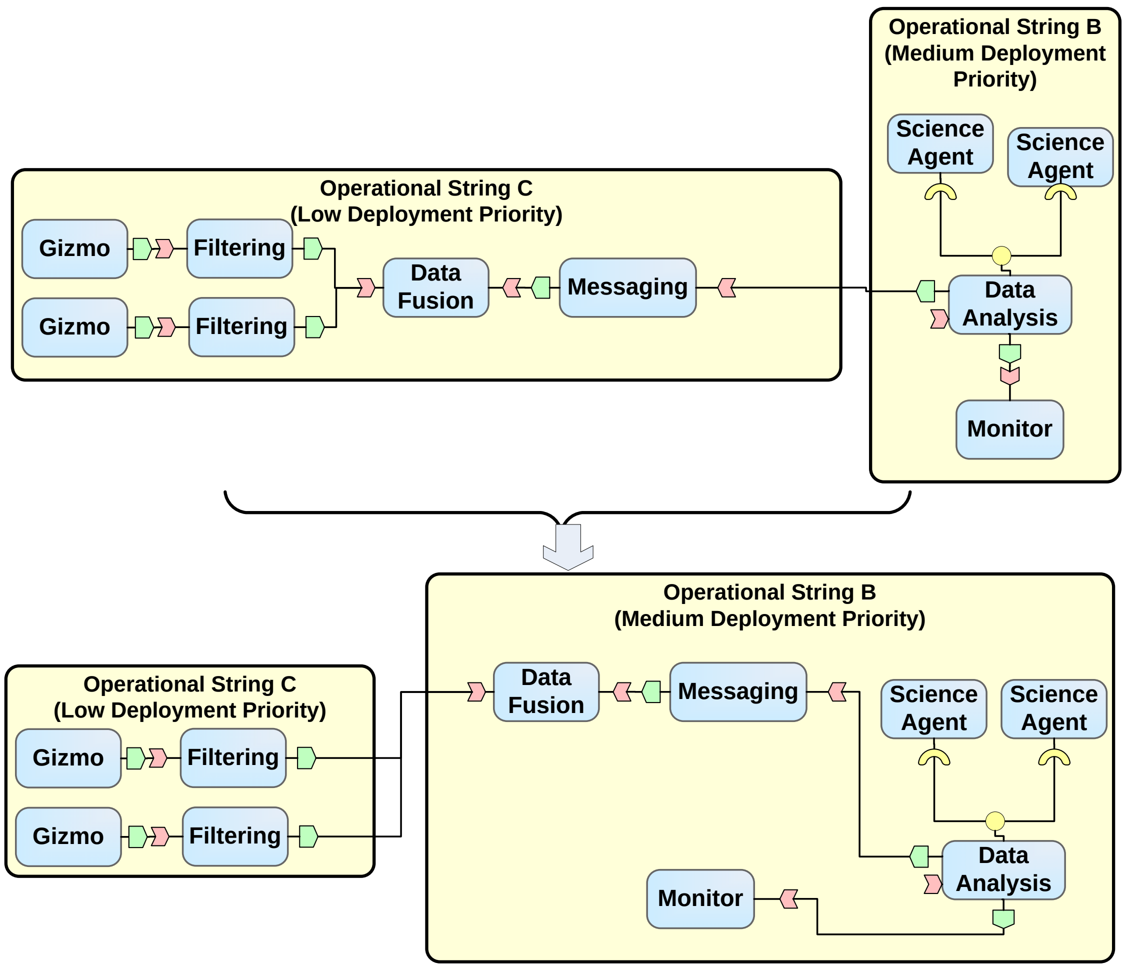


Figure V.5: A Dependency Trace across Two Operational Strings

ing components `Messaging` and `Data Fusion` from operational *C* to operational *B*, as illustrated in the bottom half of Figure V.5. After the promotion, operational string *B* does not have any dependencies to operational string *C*. The two newly created external dependencies from the two `Filtering` components in operational string *C* to operational *B* are essential to make sure the functional behavior of the MMS system is not changed. Since these two newly created external dependencies are not priority inverted, deployment priority inversion between the two operational strings can be avoided.

V.2.2.2 Handling Challenge 2 → Promotion of Components Across Multiple Operational Strings (Non-circular)

This more general case involves multiple operational strings, with a dependency trace that spans across the operational strings. As before, we assume no circular dependency exists since that will be discussed as a separate case in Section V.2.2.3.

A dependency trace that spans across multiple operational strings can be further categorized into the following two situations.

1. **Ordered dependency trace.** Figure V.6 shows an ordered dependency trace. In

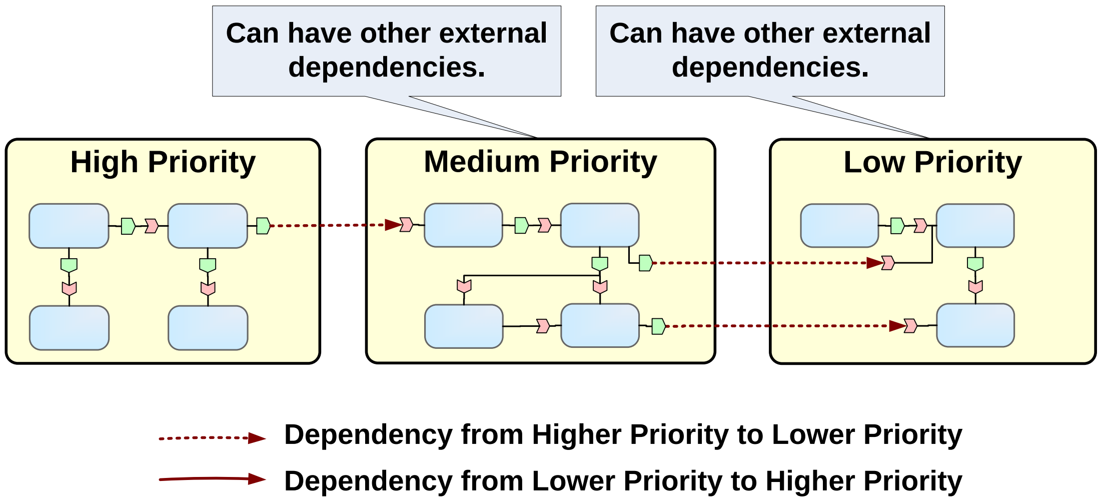

Can have other external dependencies.
Can have other external dependencies.
High Priority
Medium Priority
Low Priority
Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.6: An Ordered Dependency Trace

an ordered dependency trace the priorities of each element in the sequence have a non-increasing order, *i.e.*, all external dependencies in the sequence are priority-inverted. As a result, all the priority-inverted external dependencies must be removed through the component promotion mechanism described in Section V.2.1. The category described in Section V.2.2.1 where only two operational strings are involved is a special case of an ordered dependency trace. To remove all priority-inverted external dependencies, the PARIGE algorithm simply promotes all components in the dependency trace into the operational string where the first component of the dependency trace is located.

2. Unordered dependency trace. Figure V.7 shows an unordered dependency trace, where the priorities of the elements in the dependency trace do not have a particular order, *i.e.*, some external dependencies are priority-inverted (shown as dotted lines), whereas others are not (shown as solid lines). The PARIGE algorithm always starts with the highest

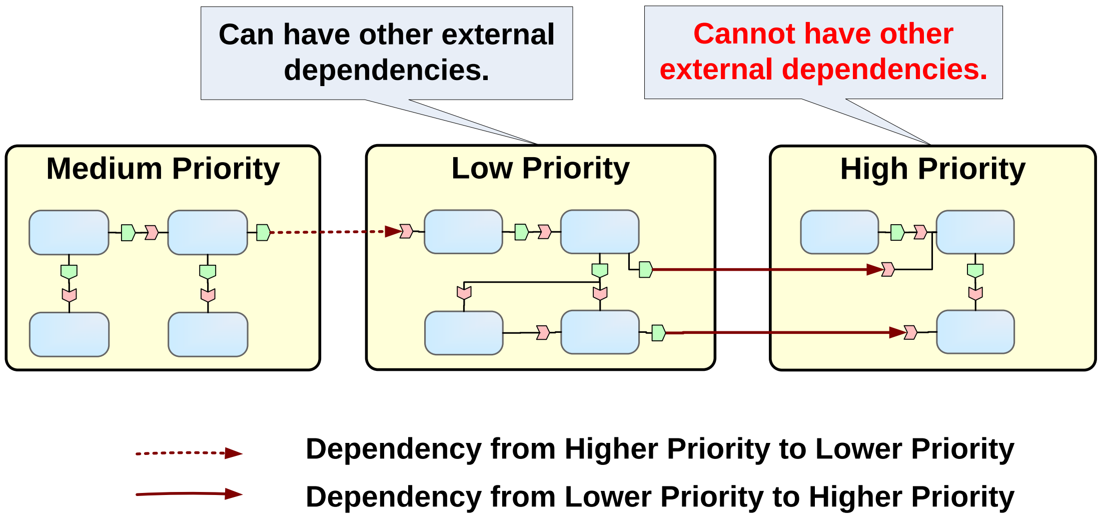
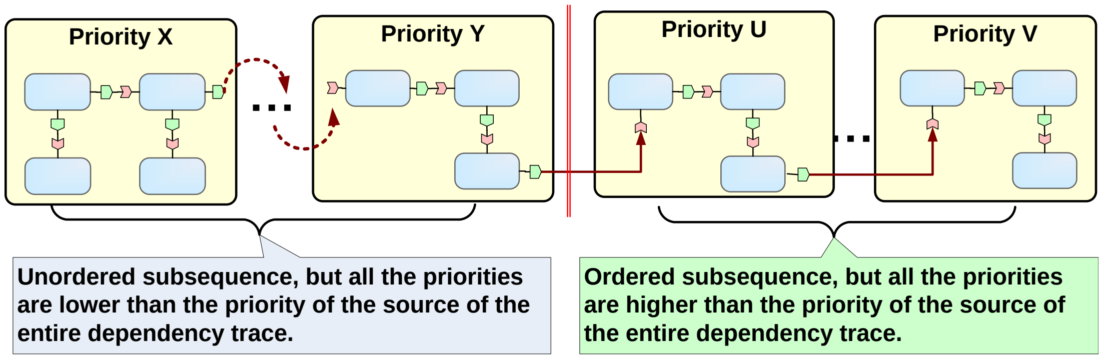

Can have other external dependencies.
Cannot have other external dependencies.
Medium Priority
Low Priority
High Priority
Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.7: An Unordered Dependency Trace

priority operational string and removes all external dependencies on it before moving to the next operational string. The algorithm therefore ensures that in an unordered dependency trace, the elements whose priorities are higher than that of the starting element will have no external dependencies, which ensures the convergence of the algorithm.

For example, given the three operational strings from Section V.1, if the high priority operational string has an external dependency on a component in low priority operational string and this component must be promoted into the medium priority operational string. When this promotion happens, the high priority string will depend on the medium priority string, which introduces additional priority-inverted external dependencies.

To remove all priority-inverted external dependencies of an unordered dependency trace, we break the entire dependency trace into two concatenated segments. As shown in Figure V.8, the first segment is a priority unordered subsequence, where all the priorities of operational strings are lower than the priority of the source of the dependency trace. The


Priority X
Priority Y
Priority U
Priority V
Unordered subsequence, but all the priorities are lower than the priority of the source of the entire dependency trace.
Ordered subsequence, but all the priorities are higher than the priority of the source of the entire dependency trace.

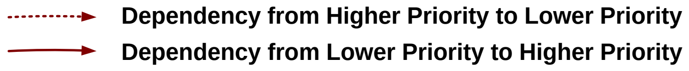

Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.8: Two Partitions of An Unordered Dependency Trace

second segment is a priority ordered subsequence, where all priorities are higher than the priority of the source of the dependency trace. For the first segment, we can promote all the components in the subsequence into the operational string where the first component of the dependency trace is located, which will ultimately result in an ordered dependency trace.

In the context of MMS case study, a priority inverted external dependency exists between operational string *B* and operational string *C*, and another priority inverted external dependency exists between operational string *A* and operational string *B*, as illustrated in the upper half of Figure V.9. Our solution traverses the dependency trace across all the three operational strings, and removes both priority inverted external dependencies. By doing this, components `Messaging` and `Data Fusion` are promoted from operational *C* to operational *A*, and components `Data Analysis` and `Monitor` are promoted from operational *B* to operational *A*, as illustrated in the bottom half of Figure V.9. In our MMS case study, only one priority inverted external dependency exists from operational string *A*, so after traversing it, operational string *A* does not have any more priority inverted external dependency remaining. The D&C service can then deploy operational string *A* before other

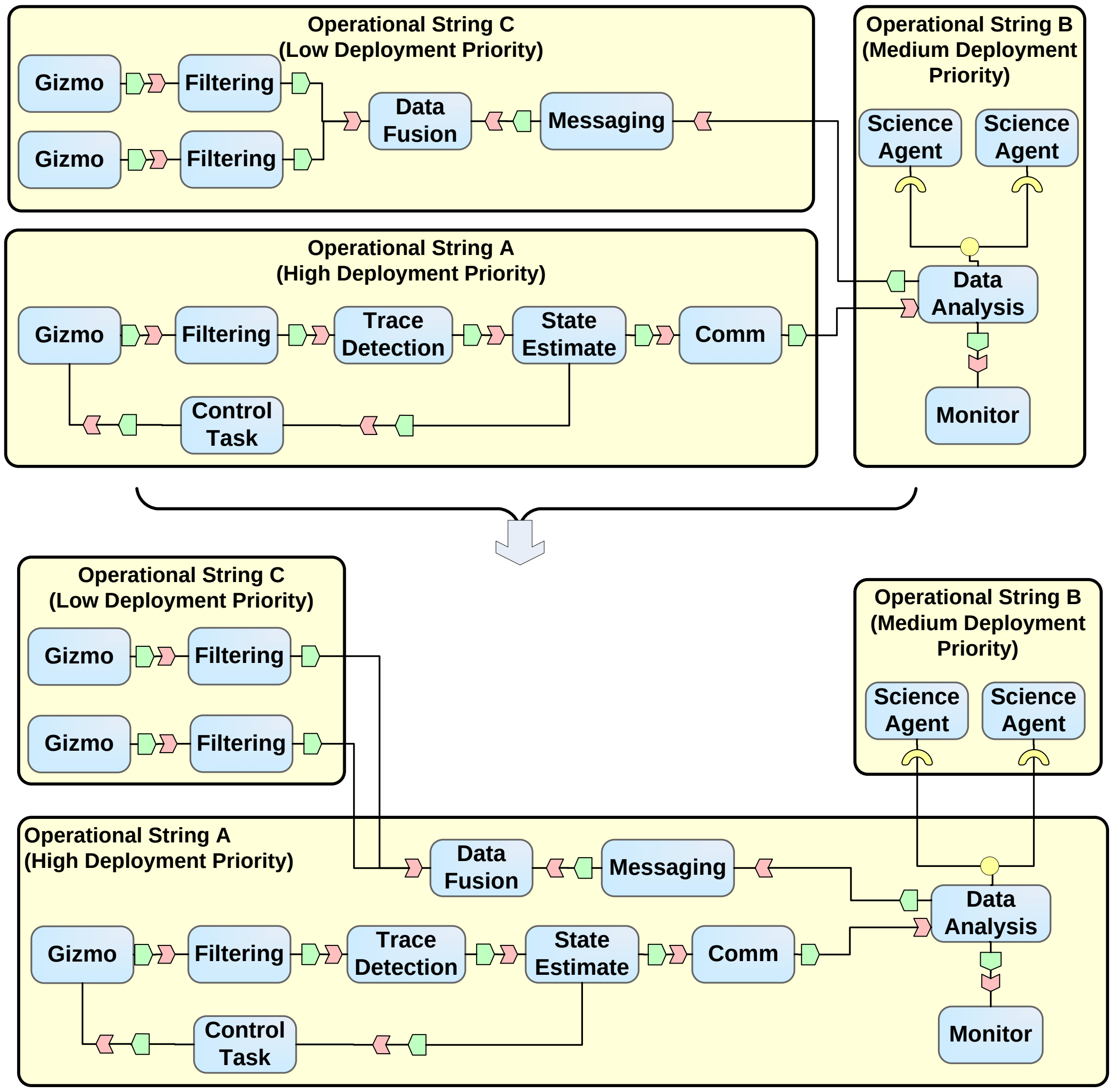


Figure V.9: A Dependency Trace across Three Operational Strings

operational strings due to its highest priority, therefore avoiding priority inversion between operational string *A* and other operational strings.

V.2.2.3 Handling Challenge 3 → Promotion of Components in the Circular Dependency Trace Among Operational Strings

Circular dependencies may exist among two or more operational strings, as shown in Figure V.10. When we discussed how the PARIGE algorithm processed an unordered de-

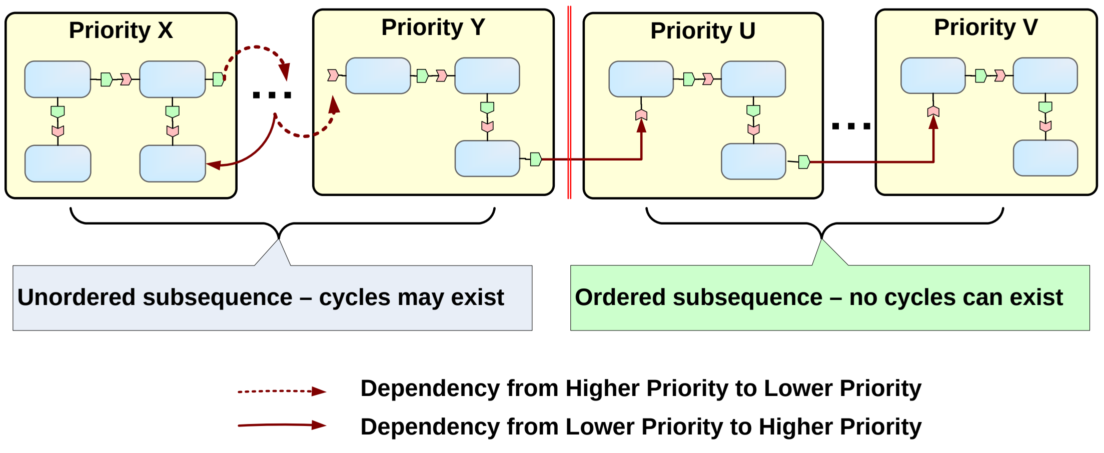

Priority X
Priority Y
Priority U
Priority V
Unordered subsequence – cycles may exist
Ordered subsequence – no cycles can exist
Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.10: Circular Dependencies in a Dependency Trace

pendency trace in Section V.2.2.2, we showed that a dependency trace can be divided into two subsequences. The priority of each element in the first subsequence is less than or equal to the priority of the source element of the dependency trace. Likewise, the priority of each element in the second subsequence is greater than that of the source elements of the dependency trace.

Since no components in the second subsequence can have priority inverted external dependencies, cycles among operational strings can only occur in the first subsequence, as shown in Figure V.10. To remove cycles across multiple operational strings, only components in the first subsequence must be promoted. The PARIGE algorithm therefore only needs to promote components existing the first subsequence to bring cycles into the same operational string.

In the context of MMS case study, a circular dependency exists between operational string *A* and operational string *B*, as illustrated in the upper half of Figure V.11. Our so-

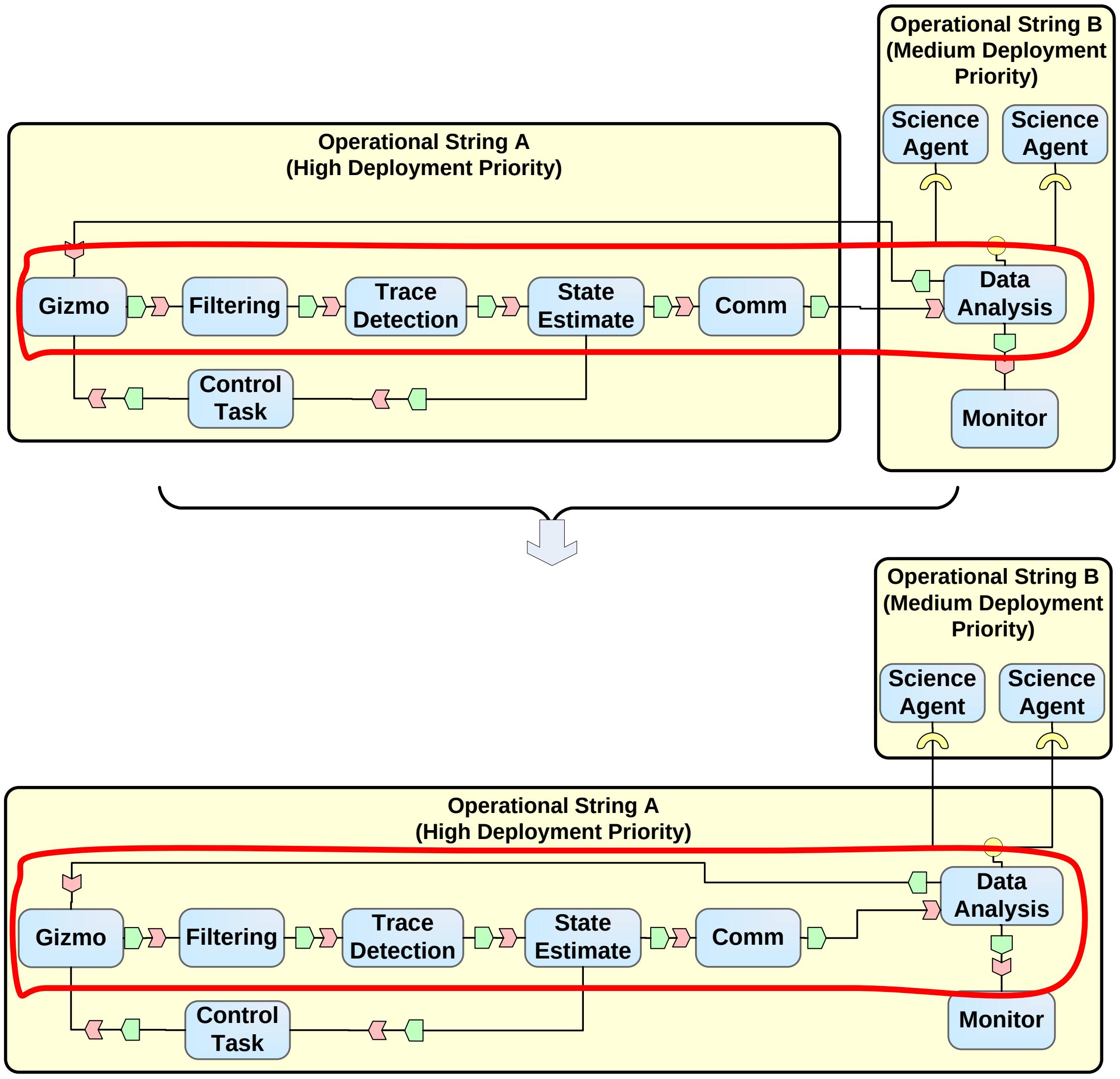


Figure V.11: A Circular Dependency Trace Case

lution traverses the dependency trace across from the operational string *A*, and promoting components `Data Analysis` and `Monitor` from operational *B* to operational *A*, as illustrated in the bottom half of Figure V.11. After such promotion, the circular decadency trace still exists but it is contained *within* the operational string A rater than across the boundary operational strings, thereby avoiding deployment failure.

V.2.2.4 Handling Challenge 4 → Selectively Applying the Algorithm to Operational Strings

The deployment latency of an operational string is affected by the size of the operational string. The dependency trace technique described above will promote components from lower priority operational strings to higher priority operational strings. Although this technique can ensure operational strings with the highest priority be deployed as early as possible, it may reduce the overall utility by delaying the deployment of lower priority operational strings in the worst case, as described as Challenge 4 in Section V.1.2. This delay occurs when *all* components in the lower priority operational strings must be promoted into the higher priority operational string, which makes both operational strings merge together and hence finish their deployment at the same time. Merging operational strings can adversely affect the responsiveness of lower priority operational strings since these strings could have been deployed earlier to serve their peers or clients, but still satisfying the dependency requirements without affecting the responsiveness of higher priority operational strings.

To overcome this difficulty, we apply an optimization technique that is based on the results produced by all the dependency traces of an operational string, as described in Subsections V.2.2.1, V.2.2.2, and V.2.2.3. If the dependency traces of an operational string promote *all* the components of a lower priority operational string into the higher priority operational string, the PARIGE algorithm simply chooses the cached graph representation of these two operational strings before the dependency traces techniques are applied. This optimization improves the overall utility of the operational strings by finishing the deployment of lower priority operational string first. The deployment of lower priority string can thus be finished earlier rather than at the moment when all components in both operational strings are deployed.

V.2.3 Design of the PARIGE Algorithm

The PARIGE algorithm uses multi-graph breadth first search (BFS) to trace dependencies and graph reconstruction to promote components and connections between components. Each graph corresponds to an operational string and can have two types of vertices:

- A **regular vertex**, which refers to a component within the operational string.
- A **reference vertex**, which refers to a component in another operational string on which it has a dependency.

For example, two operational strings have dependencies shown in Figure V.12. Based on

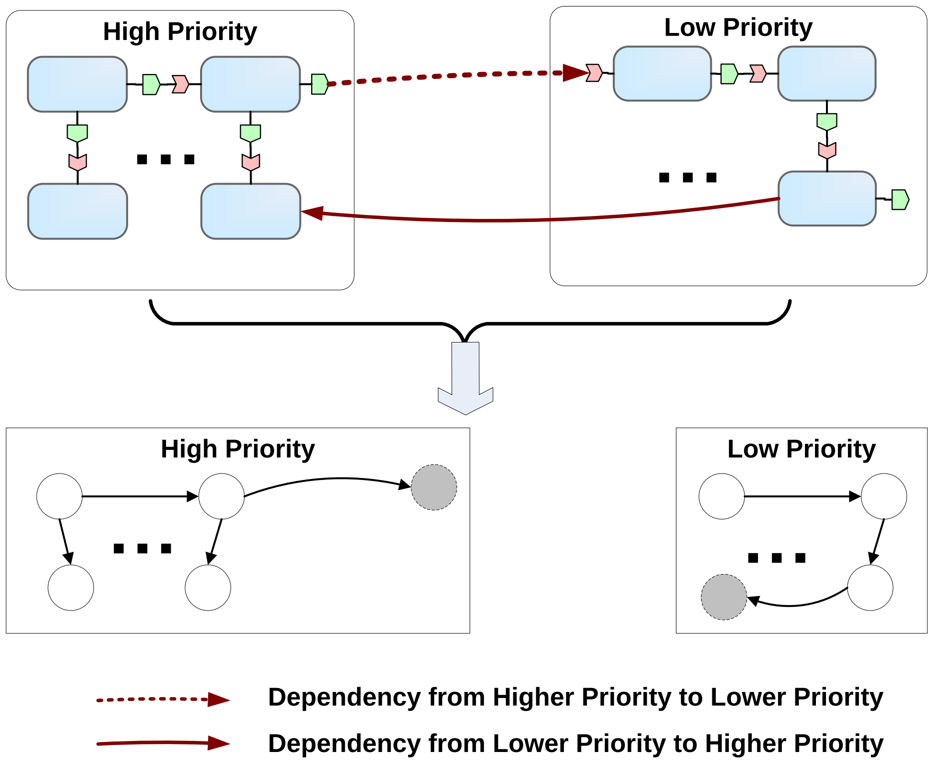

High Priority
Low Priority
High Priority
Low Priority
Dependency from Higher Priority to Lower Priority
Dependency from Lower Priority to Higher Priority

Figure V.12: Convert External Dependencies into Reference Type Vertices

the dependency relationship between these two operational strings, the PARIGE algorithm converts them into two graphs with one reference type vertex in each graph, with reference type shown as shaded vertices in the figure. Given a reference type vertex, in O(1) time we can find the referred operational string and referred component within the operational string.

In our algorithm, we define the following two types of operators against vertices in the graph:

- $Promote(V)$: Promote a reference type vertex to a regular type vertex.
- $Demote(V_1, G_1, V_2, G_2)$: Demote a regular type vertex V_1 in graph G_1 in to a reference type vertex, and the referred graph and vertex will be G_2 and V_2, respectively.

Algorithm 1 presents the PARIGE algorithm, which uses a recursive procedure defined in Algorithm 2 to process each dependency trace.

Algorithm 1 PARIGE Algorithm

```
Input: Set of Operational Strings (Represented as Graphs)
Output: Set of Operational Strings (Represented as Graphs)
Sort strings with decreasing priority;
foreach String $G_i$ in the sorted array do
    foreach External Dependency $j$ of String $G_i$ do
        Get the source vertex $V_{j1}$ of $j$;
        Get the destination vertex $V_{j2}$ of $j$;
        if priority($V_{j1}$) > priority($V_{j2}$) then
            Get the referenced string $G_e$ and vertex $V_e$;
            process_dependency_edge ($G_i$, $V_{j2}$, $G_e$, $V_e$);
        end
    end
end
```

Algorithm 2 A Recursive Procedure to Process a Dependency Trace

```
Input: $G_1$: source graph with higher priority
       $G_2$: destination graph with lower priority
       $V_1$: source vertex (reference type)
       $V_2$: destination vertex (regular type)
Output: Recomposed graphs $G_1$ and $G_2$
Promote ($V_1$);
Demote ($V_2$, $G_2$, $V_1$, $G_1$);
Do a BFS ($G_2$, $V_2$) and foreach visited edge $E_i$ do
    Get the source vertex $V_{i1}$;
    Get the destination vertex $V_{i2}$;
    if $V_{i2}$ is regular type then
        Remove_Vertex ($G_2$, $V_2$);
        Add_Vertex ($G_1$, $V_2$);
        Remove_Edge ($G_2$, $E$);
        Add_Edge ($G_2$, $E$);
    end
    else
        Get the referred graph $G_ie$ of $V_{i2}$;
        Get the referred vertex $V_{ie}$ of $V_{i2}$;
        process_dependency_edge ($G_1$, $V_{i1}$, $G_ie$, $V_{ie}$);
    end
end
```

V.2.4 Analysis of the Algorithm

To show that it is possible to apply the PARIGE algorithm at run-time to deploy operational strings dynamically, we now analyze the time complexity of the PARIGE algorithm and evaluate different effects of applying this algorithm to different configurations of operational string.

V.2.4.1 Time Complexity Analysis

The input of the PARIGE algorithm is defined as follows:

N: Total number of operational strings

C: Total number of dependencies between operational strings

$|V|$: Average number of components within an operational string

$|E|$: Average number of connections within an operational string

As shown in Algorithm 1, the sort of all external dependencies has a complexity of $O(C * log(M))$. The recursive procedure shown in Algorithm 2 shows that processing each dependency trace has a time complexity of $O(N * (|V| + |E|))$. The overall time complexity of the PARIGE algorithm exhibits a linear complexity of $O(C * N * (|V| + |E|)$. In practice, C tends to be much smaller compared with $|V|$ and $|E|$.

In summary, the linear property of our algorithm makes it possible to apply it dynamically at run-time. Section V.3 evaluate the performance overhead empirically in the context of the NASA MMS mission system case study.

V.2.4.2 PARIGE Algorithm Effects on Operational String Deployment

Two effects that the PARIGE algorithm could have on the predictability of operational string deployment are described below.

Operational string growth effect. This effect measures the cost of promoting a number of components from lower priority operational strings to higher priority operational strings. Since the deployment of each component takes time and consumes resources, the fewer components that are promoted, the more benefits the algorithm can provide since priority-inverted dependencies can be satisfied without deploying many components in lower priority operational strings.

In the worst case, *all* components from lower priority operational strings could be promoted to higher priority operational strings, which essentially merges different operational strings together. In production DRE systems, such worst cases happen rarely, *i.e.*, all the components in all operational strings have just only one dependence trace. Even in such situations, the PARIGE algorithm still performs the same as a conventional approach that does not take priority into account and only accounts for dependencies among operational strings.

Component host distribution effect. This effect means that due to the promotion of components, components that can be deployed by contacting the `NodeManager` once now contacts the same `NodeManager` multiple times during deployment. For example, if an operational string A has 1 component to deploy on $Node_A$ and operational string B has 2 components to deploy on $Node_B$ these two operational strings can be deployed by contacting the `NodeManager` on each node only once. If the algorithm moves one component from operational string B to operational string A, then the `NodeManager` on $Node_B$ must be contacted twice, once when deploying the promoted component in operational string A and once when deploying the remaining component in operational string B.

Such an effect can increase the overall deployment time due to the increasing number of round trip delays. One way to alleviate this problem is to increase the parallelism among different nodes by using asynchronous techniques between the `ExecutionManager` and `NodeManagers`, such as the Asynchronous Method Invocation (AMI) messaging policy provided by CORBA [106]. For example, AMI can coordinate all the `NodeManagers` in the domain parallelism deployment can be achieved among all the nodes, therefore alleviating the component host distribution effects.

V.3 Empirical Results

To evaluate the benefits of the PARIGE algorithm, we applied it to a representative DRE system prototype of the NASA MMS mission system described in Section V.1. This section first describes the characteristics of the hardware and software testbed and explains our experiment design. We then empirically evaluate the effectiveness of the PARIGE algorithm in three dimensions: (1) Sections V.3.2 and V.3.3 empirically measure the effectiveness of PARIGE to address challenges 1, 2, and 3 in Section V.1, Section V.3.4 empirically measures the effectiveness of PARIGE to address challenge 4 in Section V.1, which uses higher level metrics to measure the DRE system D&C QoS, and (3) Section V.3.5 empirically measures the performance overhead of the PARIGE algorithm.

V.3.1 Hardware and Software Testbed

We used the ISISlab open testbed (`www.dre.vanderbilt.edu/ISISlab`) for all our experiments. Our experiments used up to 6 nodes running Linux FC4 with Ingo Molnar's real-time kernel patches. When operational strings are deployed we use one node to run the central coordinator `ExecutionManager` and the rest of the nodes as the deployment targets.

The NASA MMS mission system prototyped used for our experiments was developed using the CIAO [124] and DAnCE [20] component middleware. This application consists of 45 components grouped together into 3 operational strings Figure V.13 shows an example operational string consisting of a science agent component that decomposes mission

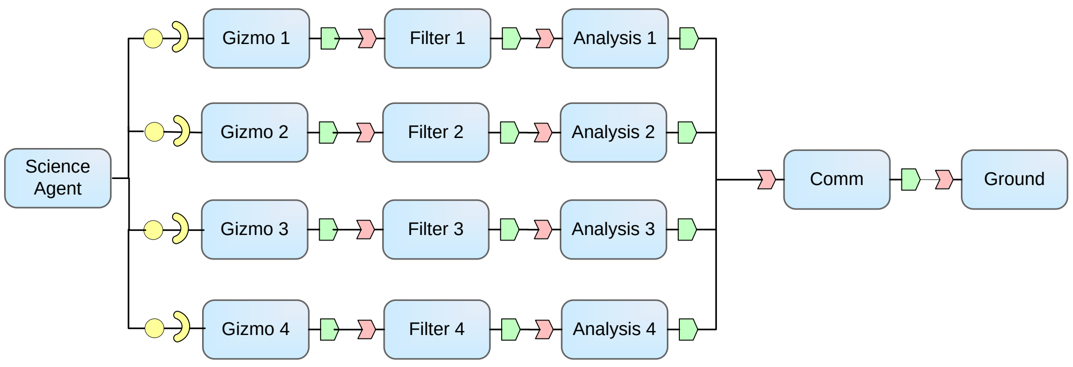


Figure V.13: A Sample Operational String of the Experiments

goals into navigation, control, data gathering, and data processing applications. Multiple gizmo components are connected to the science agent and are also connected to different payload sensors. Each gizmo component collects data from the sensors, which have varying data rate, data size, and compression requirements.

The collected data is passed through filter components, which remove noise from the data. The filter components pass the data onto analysis components, which compute a quality value indicating the likelihood of a transient plasma event. Finally, the analyzed data from each analysis component is passed to a comm (communication) component, which transmits the data to the ground component at an appropriate time.

An operational string can span across multiple physical nodes. The assignment of components to nodes is determined by a planner using high-level resource planning algorithms, and was available as input to the PARIGE algorithm. We intentionally choose different assignments for our experiments to compare how they could affect the predictability and performance different operational strings deployments.

V.3.2 Effects of Operational String Recomposition

Hypothesis. The hypothesis of this experiment is that the PARIGE algorithm should not change the functional correctness of the input operational strings but should produce correct dependencies between operational strings. In particular, the PARIGE algorithm must ensure (1) all the dependencies between components of original operational strings should remain the same after the recomposition and (2) the dependencies between operational strings must be changed such that no dependencies exist from higher priority operational strings to lower priority operational strings.

Experimental design. The experiments consist of 3 operational strings with each operational string having 15 components. The high priority operational string has one dependency to the medium priority operational string and the medium operational string has one dependency to the low priority operational string. We measure the total number of components, number of nodes, and number of dependencies (both internal and external) of each operational string before and after applying the PARIGE algorithm.

Empirical results and analysis. Table V.1 summarizes the number of components, number of nodes, and number of dependencies for each operational string before and after we run the PARIGE algorithm.

The results in the figure indicate that the number of components of high priority operational string increases from 15 to 19, while that of both medium priority and low priority

Table V.1: PARIGE Effect on Operational Strings

	Hi	Med	Lo	Total
Components Before	15	15	15	45
Components After	19	13	13	45
Dependencies Before (H->L/L->H)	1/0	1/0	0/0	2
Dependencies After (H->L/L->H)	0/0	0/3	0/3	6
Dependencies Before (Internal + External)				53 (51+2)
Dependencies After (Internal + External)				53 (47+6)

operational strings decreases from 15 to 13, so the total number of components do not change. In addition, before the experiment, the 3 operational strings have 51 internal dependencies and 2 external dependencies, with 53 dependencies in total. After the experiment, the number of internal dependencies decreases by 4 to 47 and the total number of dependencies increase by 4 to 6, with the total number still remains the same, which is in accord with the first part of our hypothesis. Finally, after applying the PARIGE algorithm, all dependencies from higher priority to lower priority operational strings are removed, which validates the second part of our hypothesis.

V.3.3 Deployment Latency vs. Deployment Priority

Hypothesis. The hypothesis of this experiment is that the PARIGE algorithm can avoid priority inversion when deploying multiple operational strings where higher priority operational strings have dependencies on lower priority operational strings.

Experimental design. We conducted two experiments on different configurations of operational string dependencies. Our first experiment consisted of 3 operational strings, each of which having 15 components evenly distributed into 5 nodes. Therefore, each node has 9 components. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. The dependency between two operational strings is shown in Figure V.14.

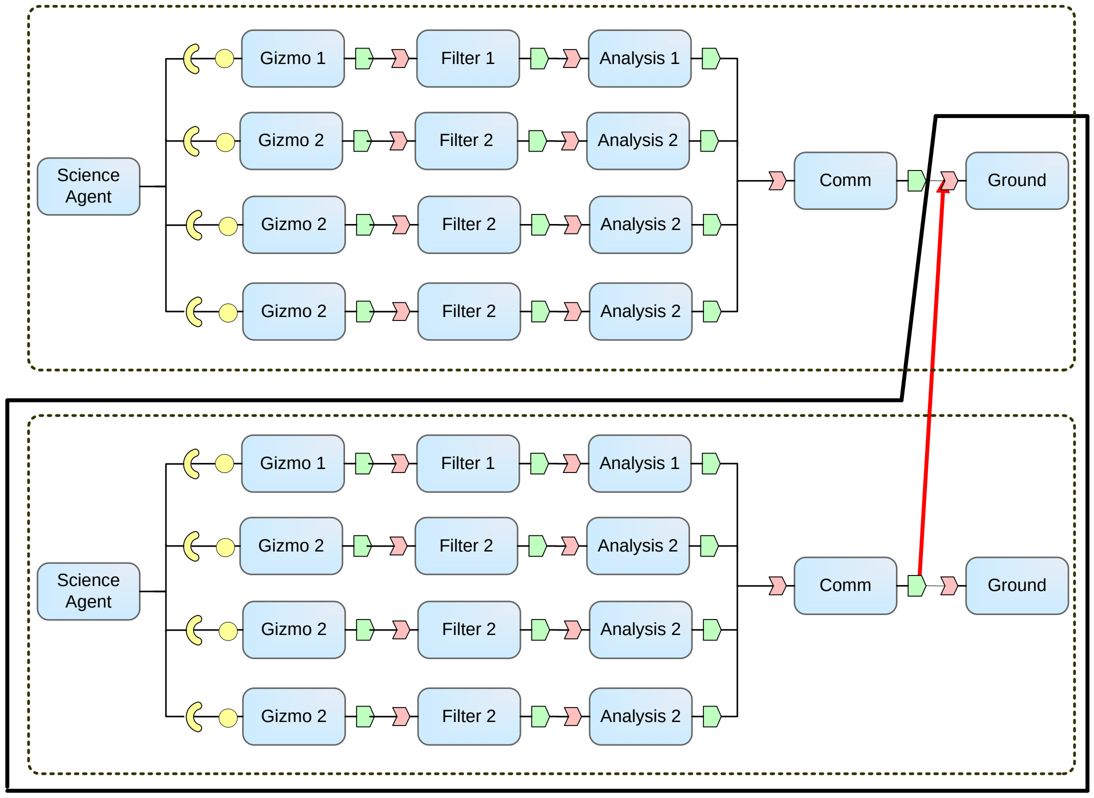


Figure V.14: Operational String Configuration with Low Growth Rate

Next we conducted another experiment with the external dependency between two different components in the operational strings, as shown in Figure V.15. We measured how the end-to-end deployment latency of each operational string can be affected in this configuration. In this experiment, there are only two operational strings, each having 15 components. The high priority operational string has one dependency on the low priority operational string, as shown in Figure V.15.

Both experiments first measured the end-to-end latency for deploying each operational string without applying the PARIGE algorithm. We then measured the end-to-end latency

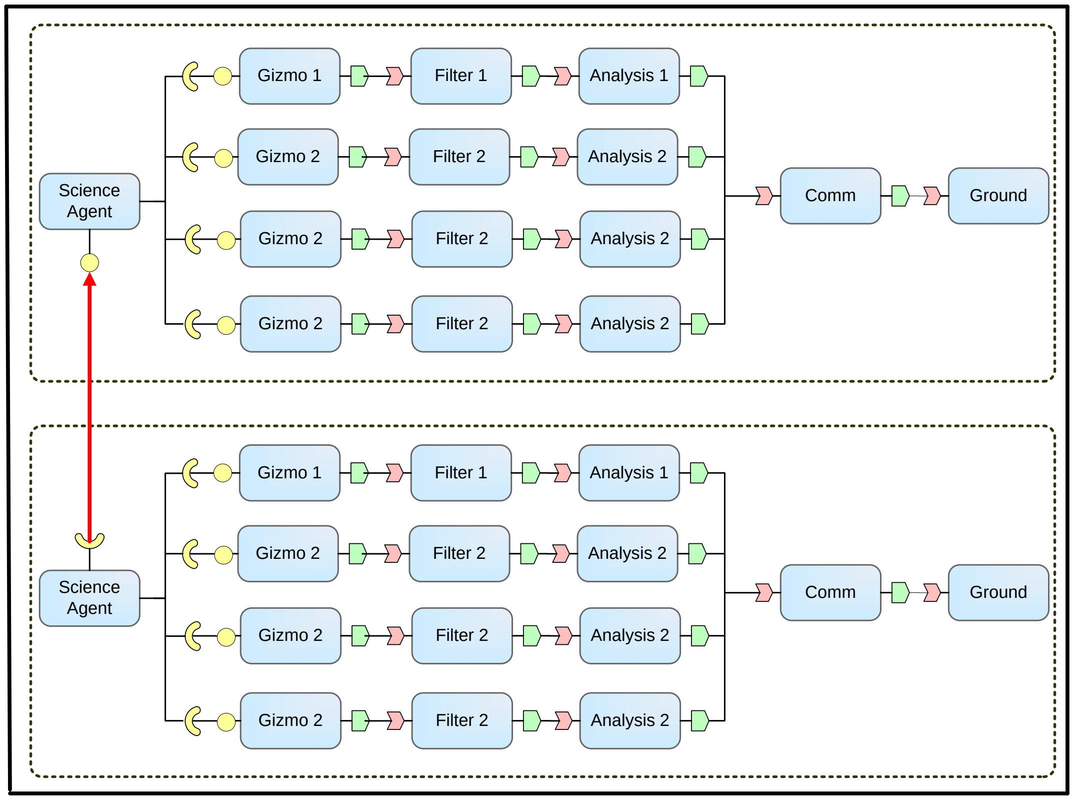


Figure V.15: Operational String Configuration with High Growth Rate

for deploying each operational strings with the PARIGE algorithm to see how latency relates to their priorities.

Empirical results and analysis.

Figures V.16 and V.17 shows the end-to-end latency of D&C request for each operational string in the two experiments described above. As shown in the Figure V.16, without applying the PARIGE algorithm, the high priority operational string yields the highest latency while the low priority operational string yields the lowest latency, while the latency of medium priority operational string lies in between.

In our experiments, there is one dependency from high priority operational string to medium priority operational string and another dependency from medium priority operational string to low priority operational string. Without applying the PARIGE algorithm,

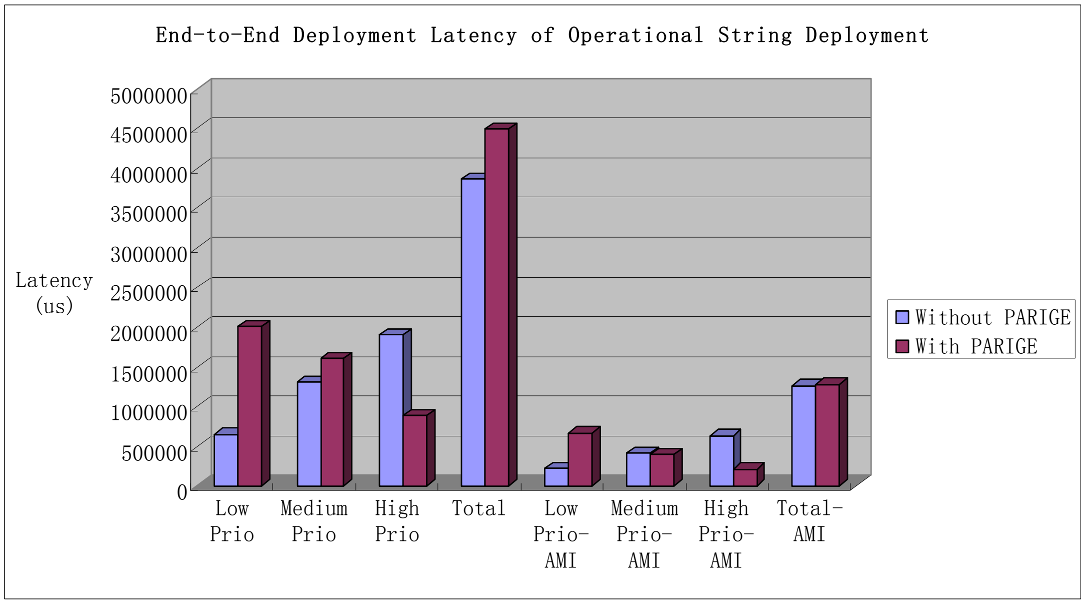
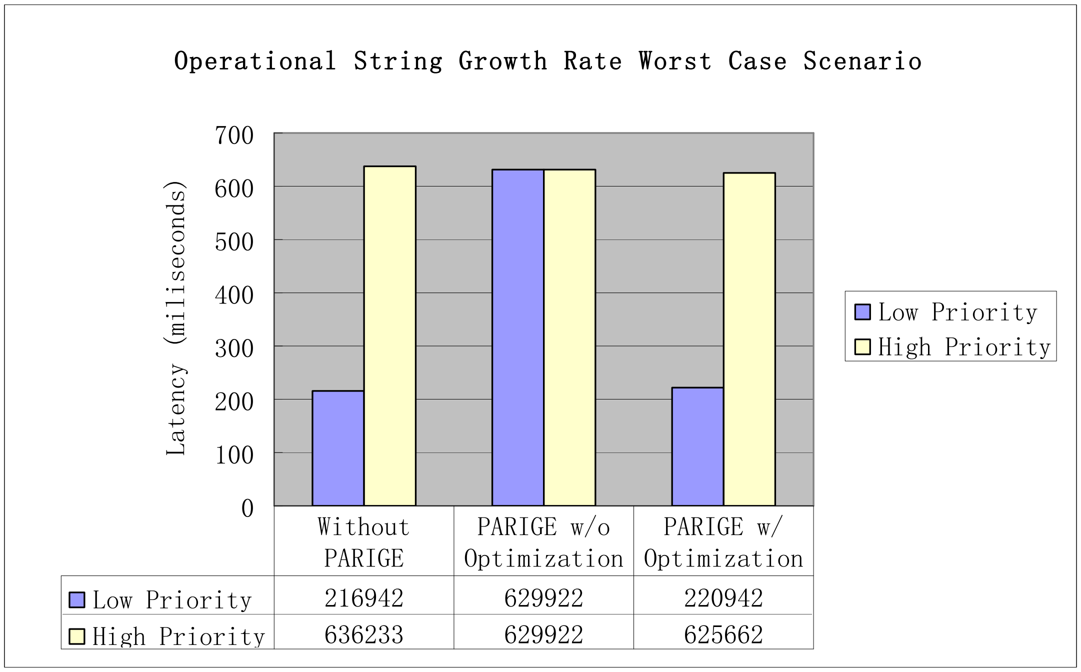


Figure V.16: D&C Latency Changes by the PARIGE Algorithm

therefore, the low priority operational string must be deployed first among the three, followed by medium priority and high priority operational strings, respectively. The PARIGE algorithm removes the priority inverted dependencies which avoids deployment priority inversion, as illustrated in the figure.

Figures V.16 also shows how the *component host distribution effect* introduced by the PARIGE algorithm is masked by applying AMI messaging policy, as described in Section V.2.4.2. In our experiment, applying AMI improves the performance of the deployment in two aspects. First, the deployment latency of *each* operational string is reduced because of the `ExecutionManager` can coordinate the `NodeManagers` to do deployment in parallel. Second, it masks the component host distribution effect, which results in a reduced total latency of all operational strings, as shown in the figure.

Figure V.17 shows that after applying the PARIGE algorithm the high priority operational string has the lowest latency since it has no external dependency on any other operational strings. The size change of each operational string is also minimal since the number of promoted components is small due to the dependency trace characteristics.



	Without PARIGE	PARIGE w/o Optimization	PARIGE w/ Optimization
▨ Low Priority	216942	629922	220942
□ High Priority	636233	629922	625662

Figure V.17: D&C Latency Changes by the Algorithm

On the other hand, the dependency between the two operational strings in our second experiment caused all components from the low priority operational string to promote to the high priority string, essentially merging the two operational strings together. As a result, the latency of deploying the high priority operational string is nearly the same as deploying it without applying the PARIGE algorithm. However, in a DRE system with multiple operational strings to deploy, it is rare that all components have only one dependence trace, as described in Section V.2.4.2.

V.3.4 Effectiveness of the PARIGE Algorithm Based on High Level Metrics

Hypothesis. The hypothesis of this experiment is to reduce the quiescence time of operational strings during the period when these operational strings are deployed. This experiment measures the effectiveness of the PARIGE algorithm based on human-perceivable metrics. As described in Section V.1, the dynamic nature of open DRE systems require on-demand injection of certain operational strings to ensure systems are kept in sync with changing mission goals. When operational strings with different importance to the entire DRE system must be deployed at the same time, the D&C framework must deploy these operational strings in effectively to ensure the overall system QoS. In this experiment, therefore, each operational string is assigned with a mission effectiveness value (MEV) to quantitatively represent its utility value for the entire system, using two utility metrics described below.

Metric *M1*: average MEV loss. The following utility function *M1* measures the average MEV loss caused by the operational string deployment cost.

$$M1 = \sum_{i=1}^{m} \frac{DeploymentTime(S_i)}{TotalDeploymentTime} \times MEV(S_i), \quad i = 1..m$$

Since each operational string has an associated deployment cost (measured by its deployment latency), this cost will necessarily cause a period of quiescence time, which results in a MEV loss.

M1 is computed by first dividing the deployment time of each operational string by the total end-to-end deployment time for all operational strings and then multiply by the MEV of that operational string to produce the weighted MEV loss for that operational string. Next, we sum up all weighted MEV loss of each operational string. In the context of our MMS case study as descried in Section V.1, *M1* measures the weighted quiescence time of the the three operational strings and their peer components during the period when the three operational strings are deployed.

Metric *M2*: Highest priority operational string MEV loss. The following utility function *M2* is similar to *M1*.

$$M2 = \sum \frac{DeploymentTime(S_{max})}{TotalDeploymentTime}$$

Rather than measuring the weighted MEV loss of all operational strings, however, *M2* measures the MEV loss the highest priority operational string(s) caused by the deployment.

M2 is computed by dividing the total deployment time of the highest priority operational string by total end-to-end deployment time for all operational strings. In the context of our MMS case study as descried in Section V.1, *M1* measures the quiescence time of the highest priority operational string *A* and its peer components during the period when the three operational strings are deployed.

Experimental design. These experiments consisted of 3 operational strings having priorities high, medium, and low, respectively, with each string having 15 components. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. The mission effectiveness values of the three operational string are summarized in Table V.2. In general, higher priority operational strings provide higher mission effectiveness values to DRE system than lower priority operational strings.

Table V.2: Mission Effectiveness Values of Operational Strings

String Priority	High	Medium	Low
MEV	3	2	1

The first experiment has a high promotion growth effect, and the second experiment has a low promotion growth effect, as described in Section V.2.4. The exact configuration of external dependencies in the experiment are the same as those in Section V.3.3.

Empirical results and analysis. Table V.3 shows that when the operational growth effect is low, *M1* was reduced by 40% and *M2* was reduced by 69%, which indicates that the PARIGE algorithm can significantly reduce the mission effectiveness loss for all operational strings. In the high growth effect situation, however, the worst case scenario happens when operational strings are merged together. The result in such worst case scenario shows that *M2* is the same as our baseline case, *i.e.*, without applying the PARIGE algorithm. This operational string merge effect indicates that the utility to the entire DRE

Table V.3: Impact of Operational String Growth Effect

	M1	*M2*
Baseline (without PARIGE algorithm)	4.71	1
Low growth effect	2.83	0.31
High growth effect	5.79	1
High growth effect w/ optimization	4.71	1

system by the highest priority operational string(s) remains the same, but *M1* (which measures the weighted MEV of all operational strings) is worse than before. To overcome this shortcoming, we applied an optimization technique that selectively chooses the deployment descriptor before applying the PARIGE algorithm, as described in Section V.2.2.4, thereby avoiding the degradation of *M1*, as shown in Table V.3.

V.3.5 Performance Overhead of the PARIGE Algorithm

Hypothesis. The hypothesis of this experiment is that the performance overhead of the PARIGE algorithm is small enough so it can be applied to deploy operational strings at run-time. In contrast to off-line analysis techniques, the PARIGE algorithm must be deployed by `ExecutionManager` to handle requests at run-time, therefore, the PARIGE algorithm should not incur excessive performance overhead to the end-to-end latency of deployment of operational strings.

Experimental design. The experiments consist of 3 operational strings each having 15 components and 2 external dependencies in total. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. We first measured the end-to-end latency for deploying all the operational strings without applying the PARIGE algorithm. We then measured the end-to-end latency for deploying increasing number of operational strings with the PARIGE algorithm to measure how much latency overhead was contributed by running the algorithm.

Empirical results and analysis. We first measure the PARIGE algorithm performance itself to determine how its performance is affected by the size of the problem, *i.e.*, number of components (determined by number of operational strings) and number of priority-inverted external dependencies. We then measure its performance overhead against an actual example with 3 operational strings and 2 external dependencies, as described above.

Figure V.18 shows the performance result of the PARIGE algorithm itself with increasing number of components and number of external dependencies. The results show that the

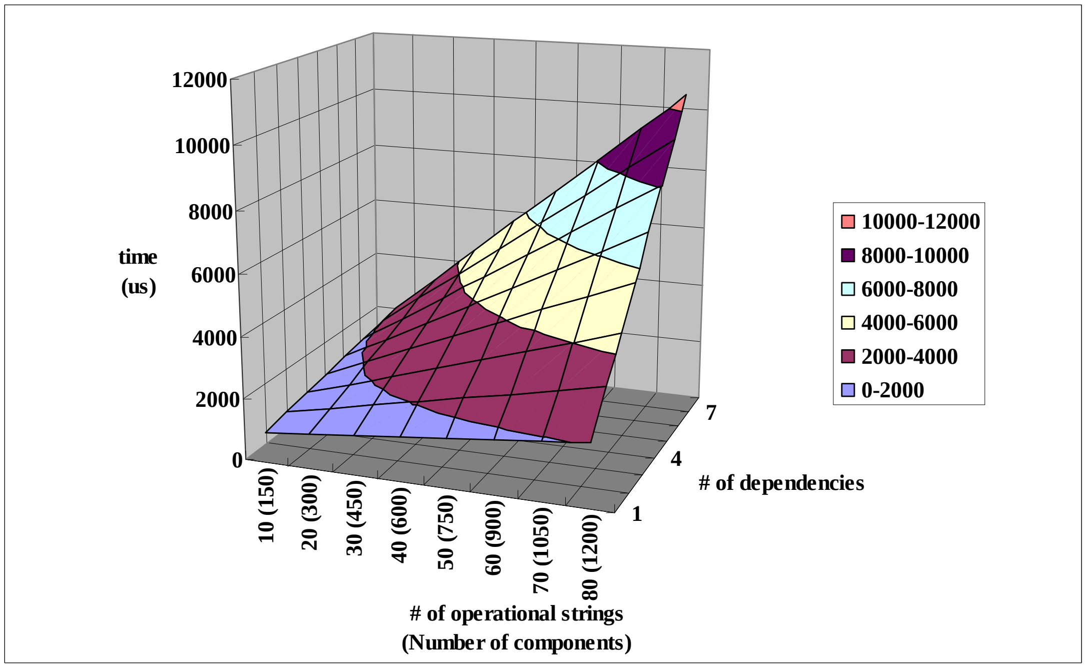


Figure V.18: Running Time of the PARIGE Algorithm

performance of the PARIGE algorithm is roughly linear to both the number of components and number of external dependencies, which is consistent with the analysis performed in Section V.2.4.1. The linear run-time performance characteristics of the PARIGE algorithm makes it suitable for dynamically deploying operational strings online at run-time because the deployment latency of all operational strings exhibits a *linear* time complexity to the number of components in the operational strings.

As long as the performance overhead of the PARIGE algorithm is acceptable to deploy one component, therefore, it should be acceptable to deploy any number of components. To validate this claim, we conducted an experiment that deployed up to 64 operational strings with 960 total components. The results in Figure V.19 shows that the deployment latency of all operational strings with and without the PARIGE algorithm. The experiment measures different number of operational strings and different number of components, ranging from 1 operational string with 15 components to 64 operational strings with 960 components. These results show that the actual performance overhead of the PARIGE algorithm

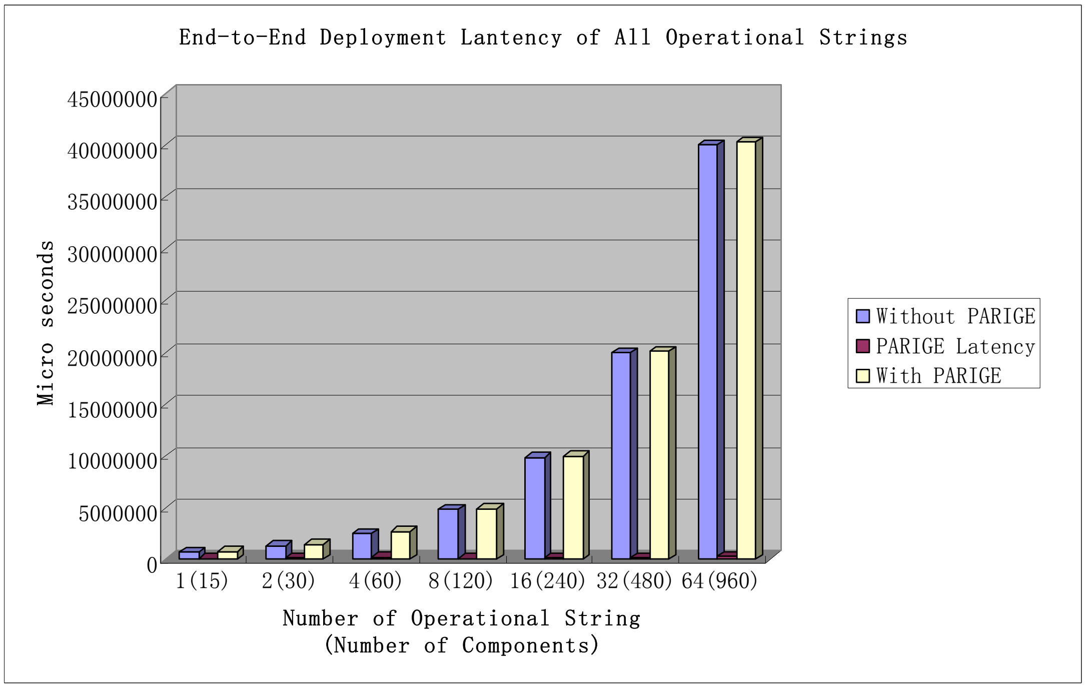


Figure V.19: Performance Overhead of the PARIGE Algorithm

for our experiment is consistently less than $\sim$1%, which further validates our earlier analysis.

V.4 Summary

The predictability and scalability of D&C frameworks is essential to support the QoS requirements of open DRE systems. This chapter describes a multi-graph algorithm that helps ensure the predictability of deploying multiple operational strings. We first analyze how deployment priority inversion can occur when operational strings have various dependency relationships. We then empirically show how the *partial priority inheritance via graph recomposition* (PARIGE) algorithm can effectively avoid deployment priority inversions and thus improve the predictability of component deployment in DRE systems.

The PARIGE algorithm has many similarities to the canonical priority inheritance protocol (PIP) [110] used for synchronization in real-time systems. The PIP ensures that when a thread blocks one or more high priority threads, it executes its critical section at the highest priority level of all the threads it blocks, *i.e.*, it inherits the highest threads priority. After executing its critical section, the thread returns to its original priority level.

In the PARIGE algorithm, lower priority operational strings are promoted to execute at the priority of higher priority operational strings to avoid deployment priority inversions. The "critical section" in the PIP is thus similar to the "deployment and configuration" activities in the PARIGE algorithm. Our work on the PARIGE algorithm, however, differs from the PIP in the following ways:

- The PARIGE algorithm avoid deadlocks because (1) it removes all priority inverted dependencies between operational strings and then deploy operational strings from the highest priority to the lowest priority sequentially, and (2) it recompose operational strings so circular dependency trace does not cross the boundary of operational strings. In contrast, the PIP may incur deadlocks because of nested resource locks.

- Only part of the operational string is affected, *i.e.*, the PARIGE algorithm just increases the deployment priorities of components with dependencies from higher priority operational strings. In contrast, the PIP does not have such fine-grained level of control because it is a general-purpose scheduling mechanism for resource sharing.

- The PARIGE algorithm is more sophisticated than the priority inheritance protocol because it traverses multiple graphs to identify which components require promotion. In contrast, the PIP is much simpler since it is locality-constrained, *i.e.*, it applies only to one resource and does not concern about how other resources are scheduled.

CHAPTER VI

CONCLUDING REMARKS

QoS-enabled component middleware is increasingly being adopted for large-scale software systems, particularly for DRE systems. Although QoS-enabled component middleware provides a higher layer of abstraction to build large-scale systems by composing reusable components, existing deployment and configuration techniques suffer from two major problems: (1) lack of capability to integrate, configure, and deploy systemic QoS concerns, and (2) lack of predictability while performing on-demand deployment at run-time for open DRE systems to accommodate changing environment or mission goals.

To address these problems, this dissertation focuses on improving both computing performance and human productivity associated with the D&C of component-based DRE systems. To improve human performance, this dissertation applies various meta-programming techniques, model-driven engineering techniques and architectural frameworks to alleviate key inherent and accidental complexities in the D&C process. To improve predictability, this dissertation has systematically identified sources of deployment priority inversion, and provides a multi-graph based algorithm called PARIGE to address these challenges.

The following is a summary of lessons learned thus far from our work developing and applying various deployment and configuration techniques for component-based DRE systems.

VI.1 Lessons Learned

VI.1.1 Automated D&C for Component-based DRE Systems

The following is a summary of lessons learned from our work developing and applying DAnCE to compose component-based DRE systems:

1. **Separating real-time QoS concerns from component business logic is essential for large-scale component-based DRE systems.** Conventional middleware generally lacks mechanisms to handle deployment concerns for DRE systems to meet their real-time requirements. The results of applying DAnCE to build a variety of systems in diverse DRE system domains including avionics mission computing, warehouse inventory tracking and shipboard computing by different users, underscore the importance of separating real-time QoS concerns from component business logic. In particular, DAnCE's automated real-time QoS provisioning capabilities greatly improve productivity of large-scale DRE systems development, integration, and deployment.

2. **Large-scale component-based DRE systems can be developed using standards-based component middleware and D&C platforms.** Existing standards-based component middleware and D&C frameworks do not address real-time QoS provisioning issues. Our experience of developing DAnCE and applying DAnCE to a variety of DRE systems reveals that component-based DRE systems can be indeed developed using standards-based component middleware and D&C frameworks. In fact, all the application components developed using CIAO and DAnCE are standard CCM components, and DAnCE can automatically provision real-time QoS capabilities to them without violating standards-based D&C profile schema.

3. **Raising the level of abstraction allows additional analysis to be performed on DRE systems.** In large-scale component-based DRE systems, many components need to interact with each other to accomplish certain tasks. Therefore, it is essential to have a central view of the DRE system such that the entire DRE system can be analyzed in a coordinated fashion. Unlike conventional D&C frameworks, DAnCE allows real-time QoS concerns to be configured later, (*i.e.*, just before system deployment phase), which allows the entire DRE system to be analyzed based on particular deployment scenarios.

VI.1.2 Publish/Subscribe Services Provisioning for Component-based DRE Systems

The following is a summary of lessons learned from our work developing and applying CIAO's publish/subscribe framework, EQAL MDE tool, and DAnCE D&C framework to provision publish/subscribe services for component-based DRE systems:

1. **Publish/Subscribe service configuration concerns should be separated from component implementation.** Our experience of developing a publish/subscribe framework within component middleware allows us to decouple publish/subscribe service configuration and deployment decisions from application logic, which enhances component reusability by allowing different publish/subscribe services and their QoS specifications (and their associated implementations) to change based on particular deployment scenarios. In particular, the container-managed architecture we developed provides the most flexible real-time publish/subscribe service configuration capabilities while preserving the benefits of the container programming model; hence, it is applicable for the widest range of DRE systems and especially useful for developing large-scale complex DRE systems.

2. **Early detection of errors improves productivity significantly.** The EQAL MDE tool allows DRE system deployers to rapidly create and synthesize publish/subscribe QoS configurations and federation configurations via *models* that are much easier to understand and analyze than hand-crafted code. EQAL helps alleviate the complexity of validating the QoS policies of publish/subscribe services for DRE component applications, which is particularly important for large-scale DRE systems that evolve over long periods of time. EQAL reduces the amount of code written by application developers for event-based DRE systems by employing a configurable publish/-subscribe service framework, which eliminates the need to write code that handles event channel lifecycles, QoS configurations, and supplier/consumer connections.

3. **Further optimizations are required for integrating COTS publish/subscribe services in component middleware.** Our lessons learned in provisioning publish/-subscribe services indicate that component middleware standards often introduce some event dispatching overhead compared to the object-oriented middleware. For example, the CCM standard inevitably introduces some performance overheads for event dispatching caused mainly by valuetype based CCM event type marshaling/de-marshaling costs and some levels of indirection between different entities including component executors, servants, contexts and containers. It is worth noting that discussion of these overheads is orthogonal to the focus of this dissertation. One possible solution approach to optimize away such overheads is through a model-based component optimization technique called *fusion* [2].

VI.1.3 Ensuring Deployment Predictability of DRE Systems

The following is a summary of lessons learned from our work developing and applying the PARIGE algorithm to ensure the predictability of D&C of component-based DRE systems:

1. **The overlap of deployment-time with run-time makes D&C frameworks essential to ensure system QoS.** The benefits provided by component middleware significantly change the lifecycle of DRE system development. Due to the complexities of open DRE systems, D&C frameworks assume more responsibilities to ensure system QoS because deployment of system services/components occurs throughout the lifecycle of the systems. By using information available at deployment time, D&C frameworks can effectively identify the complex dependency relationships among operational strings and perform various on-line optimizations, such as the operational string recomposition technique of the PARIGE algorithm presented in this dissertation.

2. **Automated recomposition of operational strings can help ensure deployment predictability of DRE systems.** Although operational strings can simplify the design of DRE systems, it is hard to manually ensure deployment predictability of all operational strings due to the complex dependencies among many operational strings. The PARIGE algorithm presented in this dissertation enhances the deployment predictability of different operational strings by recomposing operational strings automatically based on the input to the D&C framework and transparently to system deployers.

3. **Recomposition of operational strings by the PARIGE algorithm is orthogonal to later management of operational strings.** Component-based DRE systems are often designed, deployed and managed in the form of operational strings that are first class entities whose lifecycles are managed by D&C frameworks. Although the PARIGE algorithm recomposes operational strings and modifies their topologies to avoid deployment priority inversion, it does not change the behavior of the operational strings once the algorithm-modified operational strings are deployed. In particular, the D&C framework's `ExecutionManager` only recomposes operational strings for its initial deployment, but once deployed the metadata descriptors for the operational strings are still the original ones without any modification. Subsequent management of operational strings will thus not be affected by the PARIGE algorithm.

4. **The effectiveness of the PARIGE algorithm depends on operational string characteristics and their dependencies.** The PARIGE algorithm can improve deployment predictability of operational strings with negligible performance overhead, as demonstrated in Section V.3.5. The effectiveness of the PARIGE algorithm varies for different configurations of operational strings and the external dependencies among them. As shown by the empirical results in Section V.3.3, the PARIGE algorithm is

most effective when operational string growth is minimal and least effective when growth is large. In the worst case scenario—when there exists only a single dependency trace across all the operational strings—all operational strings are merged into a single operational string. In this case, the PARIGE algorithm provides no predictability improvement. Since the performance overhead of the PARIGE algorithm is negligible, however, it will perform approximately the same as without the algorithm being applied. Our future work will investigate how to quantify the gain of the PARIGE algorithm by measuring the operational string growth effect and the deployment cost of different components based on the input of the operational strings.

5. **Advanced operating system and middleware features are important complements to the PARIGE algorithm.** The PARIGE algorithm can incur certain effects when recomposing operational strings, such as the *component host distribution effect* discussed in Section V.2.4. Our experience shows that modifying the multi-graph based PARIGE algorithm itself alone is insufficient to address this undesired effect because the algorithm introduces constraints on host collocation/distribution, which affects its performance. One way to alleviate this problem is to apply asynchronous method invocations, as presented in Section V.3.3.

VI.2 Future Research Directions

This section presents some future research directions based on our experience in developing various deployment and configuration techniques for component-based DRE systems.

- **Formalize Deployment and Configuration Planning via D&C Patterns.**

  Different components consume different amounts of resources, such as memory, CPU, file descriptors, and I/O handlers. Therefore, the target running environment must be configured properly to provide an efficient environment for solution components. Different servers have different scalability profiles that are determined by the type of components they host. For example, the size of a database may increase at a different rate than the number of users of the system. Also, different servers have different security requirements that are determined by the type of components they host. For example, components that present information to the user often have different security requirements than components that implement business logic. Furthermore, the techniques for meeting availability and reliability requirements vary by type of component. Finally, every host boundary that a component invocation crosses adversely affects performance. Component invocations that cross the network are much slower than component invocations in the same application domain or process.

As a result, how to structure the servers and distribute component functionality across them to efficiently meet the operational requirements of the solution becomes a challenging task. As of today system designers still mostly rely on *ad hoc* techniques to do such D&C planning. A promising solution approach is to capture the best practice of such D&C planning in the form of formalized *D&C patterns*. The D&C of different types of systems can be guided by different D&C patterns, which can not only provide efficient D&C solution approaches but also identify potential design faults in earlier phase, *i.e.*, deployment-time phase rather than run-time phase.

- **Apply PARIGE to Improve DRE System D&C Planning.** Our future work will determine whether/how the results from the PARIGE algorithm runs can provide feedback to system designers. For example, the D&C framework can analyze the input and output to the PARIGE algorithm for each deployment request. Using this historical input/output information, a D&C framework can potentially identify those operational strings responsible for most deployment priority inversion. We conjecture that this approach will help improve DRE system D&C by reorganizing operational strings more effectively.

- **Improve System Performance via D&C-Driven Middleware Specialization.** General-purpose implementations of standard middleware are designed to be reusable since they need to satisfy a broad range of functional and QoS application requirements. Unfortunately, diversified features and capabilities provided by general purpose middleware often result in heavy-weight middleware infrastructure that adversely affects computing performance of DRE systems [54, 55]. The D&C phase provides a promising opportunity to make it possible to perform various analysis to reason about the right set of customizations and configurations to middleware to improve the overall DRE system performance. Therefore, it is beneficial to apply specialization techniques (such as partial evaluation [15] and generative programming [18]) to optimize DRE systems by using metadata contained in a DRE system deployment profile.

- **D&C for Internet Scale Networks and Grid Computing.** With the advent of grid computing and Internet computing, emerging technologies centered around these areas promise to change the way we tackle complex problems. These technologies will enable large-scale aggregation and sharing of computational, data and other resources across institutional boundaries. One future research direction in D&C is to investigate how to deploy and configure such complex applications efficiently and effectively while taking into account challenges such as Internet content delivery, mobility of computing resources, security and fault tolerance. For grid computing systems, how to achieve effective content delivery becomes a major challenge in D&C, and one promising solution approach to this is to leverage various point-to-point (P2P) techniques [14].

- **Dynamic Publish/Subscribe Service Provisioning within Component Middleware.** It is still a challenging problem to deploy and configure publish/subscribe services like Data Distribution Service (DDS) within component middleware because of its "dynamic" QoS behavior. Two main issues exist in integrating DDS into component middleware such as CCM.

  The first issue is how to reconcile different interfaces and APIs between the DDS model and CCM event communication model. For example, to increase scalability, DDS topics may contain multiple independent data channels identified by `keys`, which allows nodes to subscribe to many, possibly thousands, of similar data streams with a single subscription. However, the CCM event communication model only supports simple `eventtype` definition. On the other hand, the CCM event communication model still relies on declarative specification to specify the communication channel between a publisher and a subscriber, while DDS hides all the details about how such communication channels should be established. Because these two models have quite different semantics in defining publishers, consumers and mediation layers, how to map the concepts of one model to another becomes a challenging task because the mapping many affect many aspects of the CCM, including component programming models, container models, and D&C models.

  The second issue is how to integrate DDS to component middleware without affecting a real-time QoS guarantee provided by DDS even at run-time. To reconcile the differences between the two communication models, some adapter layer must be introduced, which should not affect QoS which DDS provides. Therefore, a future research direction is to investigate how to provide D&C services at run-time to allow CCM components to be dynamically deployed and undeployed which can automatically result in automatic discovery and control flow establishment in DDS.

The DAnCE D&C framework and CIAO component middleware are open-source and available for download at `www.dre.vanderbilt.edu/CIAO`. The EQAL MDE tool is available for download at `www.dre.vanderbilt.edu/cosmic`.

APPENDIX A

LIST OF PUBLICATIONS

Research on this dissertation has led to the following journal, conference and workshop publications.

A.1 Book Chapters

1. Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale, Jeff Gray, Yuehua Lin, and Gunther Lenz. Evolution in Model-Driven Software Product-line Architectures. *Designing Software-Intensive Systems: Methods and Principles* Edited by Dr. Pierre F. Tiako, IDEA Group, 2007

2. Gan Deng, Douglas C. Schmidt, Christopher Gill, and Nanbor Wang. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. *Handbook of Real-Time and Embedded Systems* (I. Lee, J. Leung, and S. Son, eds.), CRC Press, 2007.

A.2 Refereed Journal Publications

1. Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind Krishna, and George T. Edwards, Gan Deng, Emre Turkay, Jeff Parsons, and Douglas C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *Elsevier Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, Edited by Mehmet Aksit, 2007

2. Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale, and Nanbor Wang. The Design and Performance of Component Middleware for QoS-enabled Deployment and Configuration of DRE Systems. *Elsevier Journal of Systems and Software, Special Issue Component-Based Software Engineering of Trustworthy Embedded Systems,* 2006.

A.3 Refereed Conference Publications

1. Gan Deng, Ming Xiong, Aniruddha Gokhale, and George Edwards. Evaluating Real-time Publish/Subscribe Service Integration Approaches in QoS-enabled Component Middleware. *Proceedings of the 10th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '07)*, Santorini Island, Greece, May 7-9, 2007.

2. Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale, and Andrey Nechypurenko. Modularizing Variability and Scalability Concerns in Distributed Real-time and Embedded Systems with Modeling Tools and Component Middleware. *Proceedings of the 9th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '06)*, Gyeongju, Korea, April 24-26, 2006.

3. Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. *Proceedings of the 3rd Working Conference on Component Deployment, (CD 2005),* Grenoble, France, November 28-29, 2005.

4. Jaiganesh Balasubramanian, Balachandran Natarajan, Gan Deng, Douglas C. Schmidt,

Aniruddha Gokhale, and Jeff Parsons. Evaluating Techniques for Dynamic Component Updating. *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '05)*, Agia Napa, Cyprus, Oct 31 - Nov 4, 2005.

5. George Edwards, Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services. *Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering (GPCE 04)*, Vancouver, Canada, October 2004.

6. Andrey Nechypurenko, Tao Lu, Gan Deng, Emre Turkay, Douglas C. Schmidt, and Aniruddha Gokhale. Concern-based Composition and Reuse of Distributed Systems, *Proceedings of The 8th International Conference on Software Reuse, ACM/IEEE (ICSR8)*. Madrid, Spain, July 2004.

7. Gan Deng, Aniruddha Gokhale and Balachandran Natarajan. Model-Driven Integration of Federated Event Services in Real-Time Component Middleware. *Proceedings of the 42nd ACM Southeast Conference*, Huntsville, AL, April 2-3, 2004.

A.4 Refereed Workshop Publications

1. Gan Deng. Resolving Component Deployment and Configuration Challenges for DRE Systems via Frameworks and Generative Techniques. *Doctoral Symposium of ACM 28th International Conference of Software Engineering (ICSE 2006)*, Shanghai, China, May 20-28, 2006

2. Gan Deng, Douglas Schmidt, and Aniruddha Gokhale. Supporting Configuration and Deployment of Component-based DRE Systems Using Frameworks, Models,

and Aspects. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005)*, San Diego, CA, October 16-20, 2005.

3. Gan Deng, Gunther Lenz, Douglas C. Schmidt. Addressing Domain Evolution Challenges for Model-driven Software Product-line Architectures (PLAs). *Proceedings of the ACE/IEEE MoDELS 2005 Workshop on MDD for Software Product-lines: Fact or Fiction?*, Jamaica. October 2, 2005.

4. Andrey Nechypurenko, Tao Lu, Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale. Applying MDA and Component Middleware to Large-scale Distributed Systems: a Case Study. *Proceedings of the First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, Enschede, the Netherlands, March 2004.

5. Gan Deng, Tao Lu, Emre Turkay, Aniruddha Gokhale, Douglas C. Schmidt, and Andrey Nechypurenko. Model Driven Development of Inventory Tracking System. *Proceedings of the ACM OOPSLA 2003 Workshop on Domain-Specific Modeling Languages*, Anaheim, CA, October 26, 2003.

A.5 Submitted for Publication

1. Gan Deng, Douglas C. Schmidt, and Aniruddha Gokhale. PARIGE: Ensuring Deployment Predictability of Distributed Real-time and Embedded Systems. *Proceedings of the 11th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '08)*, Orlando, Florida, May 5-7, 2008.

REFERENCES

[1] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. *Software Quality Control*, 15(3):265–281, 2007.

[2] Krishnakumar Balasubramanian. *Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, September 2007.

[3] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[4] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems. *International Journal of Embedded Systems: Special Issue on the Design and Verification of Real-Time Embedded Software*, 2:142–155, 2006.

[5] Meriem Belguidoum and Fabien Dagnat. Dependency Management in Software Component Deployment. *Electronic Notes in Theoretical Computer Science*, 182: 17–32, 2007. ISSN 1571-0661.

[6] N. Bencomo, G. Blair, G. Coulson, P. Grace, and A. Rashid. Reflection and Aspects Meet Again: Runtime Reflective Mechanisms for Dynamic Aspects. In *AOMD '05: Proceedings of the 1st Workshop on Aspect Oriented Middleware Development*, New York, NY, USA, 2005. ACM Press.

[7] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[8] Lynne Blair, Gordon S. Blair, Anders Anderson, and Trevor Jones. Formal Support For Dynamic QoS Management in the Development of Open Component-based Distributed Systems. *IEEE Software*, 148(3), November 2001.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[10] Xia Cai, M.R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes. *Asia-Pacific Software Engineering Conference*, 00:372, 2000.

[11] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, July 2000.

[12] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[13] Xuejun Chen and Martin Simons. A Component Framework for Dynamic Reconfiguration of Distributed Systems. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 82–96, London, UK, 2002. Springer-Verlag. ISBN 3-540-43847-5.

[14] David Clark. Face-to-Face with Peer-to-Peer Networking. *IEEE Computer*, 34(1): 18–21, 2001.

[15] Charles Consel and Olivier Danvy. Tutorial Notes on Partial Evaluation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 493–501, 1993.

[16] Geoff Coulson, Paul Grace, Gordon Blair, Wei Cai, Chris Cooper, David Duce, Laurent Mathy, Wai Kit Yeung, Barry Porter, Musbah Sagar, and Wei Li. A Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing: Research Articles. *Concurrent Computing*, 18(8):865–874, 2006.

[17] Joseph K. Cross and Douglas C. Schmidt. Meta-Programming Techniques for Distributed Real-time and Embedded Systems. In *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002.

[18] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[19] Gan Deng, Tao Lu, Emre Turkay, Aniruddha Gokhale, Douglas C. Schmidt, and Andrey Nechypurenko. Model Driven Development of Inventory Tracking System. In *Proceedings of the OOPSLA 2003 Workshop on Domain-Specific Modeling Languages*, October 2003.

[20] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proc. of the 3rd Working Conf. on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005. Springer-Verlag.

[21] Mikael Desertot, Humberto Cervantes, and Didier Donsez. FROGi: Fractal Components Deployment over OSGi. In *Software Composition*, pages 275–290, 2006.

[22] Morgan Deters and Ron K. Cytron. Introduction of Program Instrumentation using Aspects. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, October 2001.

[23] Edsger Wybe Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[24] George Edwards, Gan Deng, Douglas C. Schmidt, Anirudda Gokhale, and Balachandran Natarajan. Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE)*, Vancouver, CA, October 2004.

[25] George Edwards, Douglas C. Schmidt, Aniruddha Gokhale, and Bala Natarajan. Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems. In *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, April 2004.

[26] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998.

[27] Andreas Gal, Olaf Spinczyk, and Wolfgang Schröder Preikschat. On Aspect-Orientation in Distributed Real-time Dependable Systems. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 261, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[29] Christopher Gill, Jeanna Gossett, Joseph Loyall, Douglas Schmidt, David Corman, Richard Schantz, and Michael Atighetchi. Integrated Adaptive QoS Management in Middleware: A Case Study. In *Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, Canada, May 2004. IEEE.

[30] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind S. Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas 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, 2007.

[31] Gordon S. Blair and Geoff Coulson and Anders Andersen and Lynne Blair and Michael Clarke and Fabio Costa and Hector Duran-Limon and Tom Fitzpatrick and Lee Johnston and Rui Moreira and Nikos Parlavantzas and and Katia Saikoski. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2 (6), June 2001.

[32] Pradeep Gore, Douglas C. Schmidt, Christopher Gill, and Irfan Pyarali. The Design and Performance of a Real-time Notification Service. In *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, CA, May 2004. IEEE.

[33] Martin Griss. Software Reuse: Architecture, Process, and Organization for Business Success. In *ICCSSE '97: Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, page 86, Washington, DC, USA, 1997. IEEE Computer Society.

[34] Martin Griss, Gilda Pour, and John Favaro. Making the Transition to Component-Based Enterprise Software Development Overcoming the Obstacles - Patterns for Success. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 527, Washington, DC, USA, 1999. IEEE Computer Society.

[35] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997.

[36] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh 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.

[37] George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.

[38] Didier Hoareau and Yves Mahéo. Constraint-Based Deployment of Distributed Components in a Dynamic Network. In *Architecture of Computing Systems*, pages 450–464, 2006.

[39] Jim Horning. Software Fundamentals: Collected Papers by David L. Parnas. *ACM SIGSOFT Software Engineering Notes*, 26(4):91–91, 2001.

[40] Anca-Andreea Ivan, Josh Harman, Michael Allen, and Vijay Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to

Heterogeneous Environments. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 103, Washington, DC, 2002.

[41] JBoss. JBoss AOP. http://www.jboss.org/products/aop.

[42] Mick Jordan, Grzegorz Czajkowski, Kirill Kouklinski, and Glenn Skinner. Extending a J2EE Server with Dynamic and Flexible Resource Management. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, 2004.

[43] Shoaib Kamil, Ali Pinar, Daniel Gunter, Michael Lijewski, Leonid Oliker, and John Shalf. Reconfigurable Hybrid Interconnection for Static and Dynamic Scientific Applications. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 183–194, New York, NY, 2007.

[44] S. Kannan, C. Restrepo, I. Yavrucuk, L. Wills, D. Schrage, and J.V.R. Prasad. Control Algorithm and Flight Simulation Integration Using the Open Control Platform for Unmanned Aerial Vehicles. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2000.

[45] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[46] Tatiana Kichkaylo and Vijay Karamcheti. Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 150–159, Washington, DC, USA, 2004.

[47] Tatiana Kichkaylo, Anca Ivan, and Vijay Karamcheti. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003.

[48] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[49] F. Kon, M. Roman, P. Liu, J. Mao, T Yamane, L. Magalhaes, and R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[50] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[51] K. Kon and R. Campbell. Dependence Management in Component-based Distributed Systems. *IEEE Concurrency*, 8(1):26–36, 2000.

[52] Hermann Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[53] Wojtek Kozaczynski and Grady Booch. Guest Editors' Introduction: Component-Based Software Engineering. *IEEE Software*, 15(5):34–36, 1998.

[54] Arvind Krishna, Aniruddha Gokhale, Douglas C. Schmidt, John Hatcliff, and Venkatesh Ranganath. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*, Leuven, Belgium, April 2006.

[55] Arvind S. Krishna, Aniruddha Gokhale, Douglas C. Schmidt, Venkatesh Prasad Ranganath, John Hatcliff, and Douglas C. Schmidt. Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems. In *Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines*, Half Moon Bay, Jamaica, October 2005.

[56] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 7, Washington, DC, USA, 2004.

[57] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, Greenwich, CT, 2003.

[58] Bert Lagaisse and Wouter Joosen. Component-Based Open Middleware Supporting Aspect-Oriented Software Composition. In *Component Based Software Engineering*, pages 139–154, 2005.

[59] Patrick Lardieri, Jaiganesh Balasubramanian, Douglas C. Schmidt, Gautam Thaker, Aniruddha Gokhale, and Tom 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*, 80(7):984–996, July 2007.

[60] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[61] David L. Levine, Christopher D. Gill, and Douglas C. Schmidt. Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction. *C++ Report*, 12(1), January 2000.

[62] Yu David Liu and Scott F. Smith. A Formal Framework for Component Deployment. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 325–344, 2006.

[63] Joseph P. Loyall, Richard E. Schantz, David Corman, James L. Paunicka, and Sylvester Fernandez. A Distributed Real-Time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 88–97, San Francisco, CA, 2005.

[64] Chenyang Lu, Xiaorui Wang, and Christopher Gill. Feedback Control Real-time Scheduling in ORB Middleware. In *Proceedings of the 9th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48, Washington, DC, May 2003.

[65] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996. ISSN 0163-5948.

[66] Prakash Manghwani, Joseph Loyall, Praveen Sharma, Matthew Gillen, and Jianming Ye. End-to-End Quality of Service Management for Distributed Real-Time Embedded Applications. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, volume 03, Los Alamitos, CA, USA, 2005.

[67] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt, and Bala Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004.

[68] Richard Monson-Haefel. *Enterprise JavaBeans, 3rd Edition*. O'Reilly and Associates, Inc., Sebastopol, CA, 2001.

[69] Richard Monson-Haefel and David A. Chappell. *Java Message Service*. O'Reilly, 1st edition, December 2000.

[70] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Using Interceptors to Enhance CORBA. *IEEE Computer*, 32(7):64–68, July 1999.

[71] Andrey Nechypurenko, Douglas C. Schmidt, Tao Lu, Gan Deng, and Aniruddha 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, March 2004.

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

[73] *Data Distribution Service for Real-time Systems Specification*. Object Management Group, 1.0 edition, March 2003.

[74] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.6.1 edition, May 2002.

[75] *Event Service Specification Version 1.1*. Object Management Group, OMG Document formal/01-03-01 edition, March 2001.

[76] Object Management Group. *Management of Event Domains Specification*. Object Management Group, OMG Document formal/01-06-03 edition, June 2001.

[77] Object Management Group. *Notification Service Specification*. Object Management Group, OMG Document formal/2002-08-04 edition, August 2002.

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

[79] *Deployment and Configuration Adopted Submission*. Object Management Group, OMG Document mars/03-05-08 edition, July 2003.

[80] Object Management Group. *Lightweight CCM FTF Convenience Document*. Object Management Group, ptc/04-06-10 edition, June 2004.

[81] *Unified Modeling Language: OCL version 2.0 Final Adopted Specification*. Object Management Group, OMG Document ptc/03-10-14 edition, October 2003.

[82] *Specification of the Portable Object Adapter (POA)*. Object Management Group, OMG Document orbos/97-05-15 edition, May 1997.

[83] Object Management Group. *Real-time CORBA Specification*. Object Management Group, OMG Document formal/05-01-04 edition, August 2002.

[84] OCI Integrated Information Systems. TAO DDS Open Source Project. http://downloads.ociweb.com/DDS/, 2006.

[85] DARPA Information Exploitation Office. Program Composition for Embedded Systems (PCES). `www.darpa.mil/ixo/`, 2000.

[86] *Deployment and Configuration of Component-based Distributed Applications, v4.0*. OMG, Document formal/2006-04-02 edition, April 2006.

[87] Carlos O'Ryan and Douglas C. Schmidt. Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation. In 5th *International Workshop on Object-oriented Real-time Dependable Systems*, Monterey, CA, November 1999.

[88] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2): 115–132, March 2002.

[89] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, London, UK, September 2001.

[90] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 100–109, Boston, Massachusetts, 2003.

[91] Prism Technologies. OpenSplice Data Distribution Service. http://www.prismtechnologies.com/, 2006.

[92] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7):1070–1085, July 2003.

[93] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. In *Proceedings of Second International Working Conference on Component Deployment*, pages 50–64, Edinburgh, UK, May 2004.

[94] Ragunathan Rajkumar, Mike Gagliardi, and Lui Sha. The Real-time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-time Systems: Design and Implementation. In *First IEEE Real-time Technology and Applications Symposium*, May 1995.

[95] Andreas Rasche and Andreas Polze. Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, page 164, Washington, DC, 2003.

[96] Peter Rigole, Tim Clerckx, Yolande Berbers, and Karin Coninx. Possibilities to Improve Ground-based Cloud Cover Observations Using Satellite Application Facility (SAFNWC) Products. *Geografija Scientific Journal*, 43(1):21–29, 2007.

[97] Peter Rigole, Tim Clerckx, Yolande Berbers, and Karin Coninx. Task-driven Automated Component Deployment for Ambient Intelligence Environments. *Pervasive Mobible Computing*, 3(3):276–299, 2007.

[98] Tom Ritter, Marc Born, Thomas Unterschütz, and Torben Weis. A QoS Metamodel

and its Realization in a CORBA Component Infrastructure. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, Honolulu, HI, January 2003.

[99] Wendy Roll. Towards Model-Based and CCM-Based Applications for Real-time Systems. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, May 2003.

[100] Christian PÍęrez SÍębastien Lacour and Thierry Priol. Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit. In *Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004)*, Edinburgh, UK, May 2004.

[101] Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha Pal. Packaging Quality of Service Control Behaviors for Reuse. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, pages 375–385, Crystal City, VA, April/May 2002.

[102] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2001.

[103] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[104] Douglas C. Schmidt. Evaluating Architectures for Multi-threaded CORBA Object Request Brokers. *Communications of the ACM Special Issue on CORBA*, 41(10): 54–60, October 1998.

[105] Douglas C. Schmidt and Carlos O'Ryan. Patterns and Performance of Real-time Publisher/Subscriber Architectures. *Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes*, 2002.

[106] Douglas C. Schmidt and Steve Vinoski. Programming Asynchronous Method Invocations with CORBA Messaging. *C++ Report*, 11(2), February 1999.

[107] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-time Object Request Brokers. *Computer Communications*, 21(4): 294–324, April 1998.

[108] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[109] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp,

and Lou DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk - The Journal of Defense Software Engineering*, November 2001.

[110] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[111] Praveen Sharma, Joseph Loyall, George Heineman, Richard Schantz, Richard Shapiro, and Gary Duzan. Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, October 2004.

[112] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions*, volume 576, Aug 2000.

[113] David C. Sharp. Avionics Product Line Software Architecture Flow Policies. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[114] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *IEEE Real-time and Embedded Technology and Applications Symposium*, Washington, DC, May 2003. IEEE Computer Society.

[115] Gurdip Singh, Prashant S. Kumar, and Qiang Zeng. Configurable Event Communication in Cadena. In *IEEE Real-time and Embedded Technology and Applications Symposium*, pages 130–139, May 2004.

[116] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[117] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An Event Based Object Model for Distributed Programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72–86, London, 1995. Springer-Verlag.

[118] Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas C. Schmidt, Aniruddha Gokhale, and Nanbor Wang. The Design and Performance of Component Middleware for QoS-enabled Deployment and Conguration of DRE Systems. *Elsevier Journal of Systems and Software, Special Issue Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):668–677, March 2007.

[119] Dipa Suri, Adam Howell, Douglas C. Schmidt, Gautam Biswas, John Kinnebrew, Will Otte, and Nishanth Shankaran. A Multi-agent Architecture for Smart Sensing in the NASA Sensor Web. In *Proceedings of the 2007 IEEE Aerospace Conference*, Big Sky, Montana, March 2007.

[120] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the International Conference on Software Engineering*, pages 107–119, May 1999.

[121] Thuan Thai and Hoang Lam. *.NET Framework Essentials*. O'Reilly, Sebastopol, CA, 2001.

[122] France Universite des Sciences et Technologies de Lille. The OpenCCM Platform. `corbaweb.lifl.fr/OpenCCM/`, 2003.

[123] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive Programming in JAsCo. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 75–86, Chicago, Illinois, 2005.

[124] Nanbor Wang, Christopher Gill, Douglas C. Schmidt, and Venkita Subramonian. Configuring Real-time Aspects in Component Middleware. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, October 2004.

[125] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS)*, page 301, Washington, DC, USA, 2002. IEEE Computer Society.

[126] ZZip. Compression Tool Under the GPL. `debin.org/zzip/`, 2001.