MODEL-DRIVEN ENGINEERING OF COMPONENT-BASED DISTRIBUTED, REAL-TIME AND EMBEDDED SYSTEMS

By

Krishnakumar Balasubramanian

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

December, 2007

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

Dr. Janos Sztipanovits

Dr. Gabor Karsai

Dr. Jeff Gray

To Amma and Appa for their patient support over the years

To Vidhya and Sarathy for all the encouragement

ACKNOWLEDGMENTS

The past seven years have proven to be quite an eventful journey in a variety of ways. I have had the wonderful opportunity to meet and make friends with a number of people and visit a number of interesting places, all of which have left me enriched with the whole experience. First of all, I would like to thank my advisor, Dr. Douglas C. Schmidt, for providing me with an opportunity to work with him, initially with the Distributed Object Computing (DOC) group at St.Louis and later in Nashville. Doug has been the single biggest source of inspiration in my career as a graduate student, both in good and bad times. Over the years, I have learned a lot from his words of wisdom on a wide variety of topics, both academic and non-academic, and hope that it will continue in the future as well. Next, I would like to thank Dr. Aniruddha S. Gokhale for the various fruitful collaborations over the past four years. It has been a pleasure to work with Andy, who is also a member of my dissertation committee. Dr. David Levine, Dr. Christopher D. Gill and Dr. Ron K. Cytron served as my advisors and supported me during my stay in St. Louis; I thank them for making my journey more enjoyable.

I would like to thank Dr. Janos Sztipanovits, Dr. Gabor Karsai and Dr. Jeff Gray for collaborations over the past four years and for agreeing to serve on my dissertation committee. I am especially grateful for the time Jeff devoted to reviewing my proposal and would like to thank him for his help. 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: Sylvester Fernandez, Christopher Andrews and Theckla Louchios at Lockheed Martin Co., Eagan, for agreeing to be beta-testers of PICML; Prakash Manghwani, Matt Gillen, Praveen Sharma, Jianming Ye, Joe Loyall, Richard Schantz and George Heineman at BBN Technologies, Boston, for providing us with the component-based UAV application used as a case study in this dissertation; John Slaby, Steve Baker, Thomas Silveria and Paul Work at Raytheon Co., Portsmouth, for being power users of PICML and also providing us with

the shipboard computing case study; Partick Lardieri, Gautam Thaker, Daniel Waddington and Richard Buskens at Lockheed Martin Advanced Technology Laboratories, Cherry Hill, for supporting the work that resulted in the creation of SIML and PAM.

My stay in the DOC group at WashU was very enjoyable due to the following past and present members: Sharath Cholleti, Morgan Deters, Pradeep Gore, Frank Hunleth, Yamuna Krishnamurthy, Martin Linenweber, Ravipratap Madimsetti, Mike Plezbert. I would like to thank Angelo Corsaro, Ossama Othman, Carlos O'Ryan, Irfan Pyarali and Nanbor Wang for answers to many questions over the years. After my move to Nashville, I have greatly enjoyed the company of the following people: Aditya Agrawal, Hariharan Kannan, Sachin Mujumdar, Sujata Mujumdar, Anantha Narayanan, Prakash Raghottamachar, Pramila Rani.

I would like to thank Jeff Parsons for not holding back with his views on just about everything; discussions with Jeff have made many a dull day brighter. Balachandran Natarajan eased my transition from St. Louis to Nashville and remains a very good friend. Jaiganesh Balasubramanian, Arvind Krishna and Nishanth Shankaran, provided company for many enjoyable hours spent honing our collective psycho-analysis skills, endless debates and some fine dining. The past and present members of ISIS deserve special mention for all the enjoyable break room discussions.

The amigos from Sapphire have always been there for me and I am very thankful for their continued friendship. Last but not the least, I would like to thank my family: my parents for their love and patience through many difficult years; my sister, Vidhya, and my brother-in-law, Sarathy, for their encouragement and help when I needed both. But for my family, this dissertation would have remained a dream.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES viii

LIST OF FIGURES ix

Chapter

I. INTRODUCTION 1

I.1. Emerging Trends and Technologies 1
I.2. Overview of Component Middleware 3
I.3. Research Challenges 7
I.4. Research Approach 10
I.5. Research Contributions 11
I.6. Dissertation Organization 12

II. EVALUATION OF ALTERNATE APPROACHES 14

II.1. Composition of Component-based Systems 14
II.1.1. System Composition: Alternate Approaches 16
II.1.2. System Composition: Unresolved Challenges 20
II.2. Optimization of Component-based Systems 23
II.2.1. System Optimization: Alternate Approaches 24
II.2.2. System Optimization: Unresolved Challenges 26
II.3. Integration of Component-based Systems 29
II.3.1. System Integration: Alternate Approaches 29
II.3.2. System Integration: Unresolved Challenges 31
II.4. Summary 32

III. TECHNIQUES FOR COMPOSING COMPONENT-BASED SYSTEMS . 33

III.1. Overview of PICML 34
III.2. Composition using QoS-enabled Component Middleware - A Case Study 36
III.2.1. Challenges in Composing the UAV Application 38
III.2.2. Resolving UAV Composition Challenges with PICML . 41
III.3. Summary 56

IV. TECHNIQUES FOR OPTIMIZING COMPONENT-BASED SYSTEMS . 57

IV.1. Challenges in Large-scale Component-based DRE systems 61
IV.1.1. Key Sources of Memory Footprint Overhead 61
IV.1.2. Key Sources of Latency Overhead 65
IV.2. Deployment-time Optimization Techniques 67
IV.2.1. Deployment-time Optimization Algorithms 68
IV.2.2. Design and Functionality of the Physical Assembly Mapper 81
IV.3. Empirical Evaluation and Analysis 86
IV.3.1. Experimental Platforms 87
IV.3.2. Experimental Setup 90
IV.3.3. Empirical Footprint Results 91
IV.3.4. Empirical Latency Results 94
IV.4. Summary 100

V. TECHNIQUES FOR INTEGRATING COMPONENT-BASED SYSTEMS 102

V.1. Functional Integration - A Case Study 107
V.1.1. Shipboard Enterprise Distributed System Architecture . 107
V.1.2. Functional Integration Challenges 109
V.2. DSML Composition using GME 116
V.3. Integrating Systems with SIML 120
V.3.1. The Design and Functionality of SIML 120
V.3.2. Resolving Functional Integration Challenges using SIML 123
V.3.3. Evaluating SIML 129
V.4. Summary 132

VI. COMPARISON WITH RELATED RESEARCH 134

VI.1. Related Research: Composition Techniques 134
VI.2. Related Research: Optimization Techniques 136
VI.3. Related Research: Integration Techniques 139
VI.4. Summary 143

VII. CONCLUDING REMARKS 144

VII.1.Lessons Learned 144
VII.1.1.Composition Techniques 145
VII.1.2.Optimization Techniques 146
VII.1.3.Integration Techniques 148
VII.2.Summary of Research Contributions 150
VII.3.Future Research Directions 152

Appendix

A. List of Publications 156

A.1. Book Chapters 156
A.2. Refereed Journal Publications 156

A.3. Refereed Conference Publications 157
A.4. Refereed Workshop Publications 158

REFERENCES 159

LIST OF TABLES

Table Page

I.1. Summary Of Research Contributions 12

IV.1. QoS Policy Configuration 95

V.1. Evaluating Functional Integration using SIML 130

LIST OF FIGURES

Figure		Page
I.1.	Key Elements in the CORBA Component Model	4
I.2.	Research Approach	10
II.1.	Composition Dimensions	15
II.2.	Compositions of Systems from COTS Components	21
II.3.	Composition Overhead in Component Assemblies	27
III.1.	Emergency Response System components	39
III.2.	Component Deployment Planning	52
III.3.	Single Image Stream Assembly	54
III.4.	UAV Application Assembly Scenario	55
IV.1.	Physical Assembly	58
IV.2.	Physical Assembly Mapper	84
IV.3.	Sample Operational String	88
IV.4.	Basic Single Processor Scenario	89
IV.5.	Static and Dynamic Footprint	92
IV.6.	Total Footprint	93
IV.7.	Original Latency Measurements	96
IV.8.	Cross-pool/Lane Latency Measurements	97
IV.9.	Basic Single Processor Latency Measurements	99
V.1.	Enterprise Distributed System Architecture	108
V.2.	Functional Integration Challenges	110

V.3. Domain-Specific Modeling Language Composition in GME 118

V.4. Design of System Integration Modeling Language (SIML) Using Model Composition 121

V.5. Generating a Web Service Gateway Using SIML 123

CHAPTER I

INTRODUCTION

I.1 Emerging Trends and Technologies

During the past two decades, advances in languages and platforms have raised the level of software abstractions available to developers. For example, developers today typically use expressive object-oriented languages such as C++ [130], Java [2], or C# [47], rather than FORTRAN or C. Object-oriented (OO) programming languages simplified software development by providing higher level abstractions and patterns. For example, OO languages provide support for associating data and related operations as well as decoupling interfaces from the implementations. Thus well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity and elegance. Resting on the foundations of the OO languages, reusable class libraries and application framework platforms [115] were developed. This also led to the development of robust distributed object computing middleware (DOC) which applied design patterns [17] (like Broker) to abstract away low-level operating system and protocol-specific details of network programming. This resulted in the development of distributed systems since the DOC middleware hid a lot of the complexity associated with building such systems using previous generation middleware technologies. DOC middleware standards like CORBA [102] and Java RMI [133] coupled with mature implementations like TAO [119] led to development of more robust software and more powerful distributed systems. Although DOC middleware provided a number of advantages over previous generation middleware, a number of significant limitations remain. Some of the limitations with DOC middleware include:

- **Inability to provide multiple alternate views per client.** An object in DOC middleware like CORBA typically implements a single class interface, which may be related by inheritance with other classes. In contrast, a component can implement

many interfaces, which need not be related by inheritance. A single component can therefore appear to provide varying levels of functionality to its clients.

- **Inability of clients to navigate between interfaces of a server in a standardized fashion.** Components provide transparent "navigation" operations, *i.e.*, moving between the different functional views of a component's supported interfaces. Conversely, navigation in objects is limited to moving up or down an inheritance tree of objects via downcasting or "narrow" operations. It is also not possible to provide different views of the same object since all clients are granted the same level of access to the object's interfaces and state.

- **Extensibility of the middleware limited to language (Java, C++) and/or platform (COM, CORBA).** Objects are units of instantiation, and encapsulate types, contracts, and behavior [135] that model the physical entities of the problem domain in which they are used. They are typically implemented in a particular language and have some requirements on the layout that each inter-operating object must satisfy. In contrast, a component need not be represented as a class, be implemented in a particular language, or share binary compatibility with other components (though it may do so in practice). Components can therefore be viewed as providers of functionality that can be replaced with equivalents components written in another language. This extensibility is facilitated via the Extension Interface design pattern [118], which defines a standard protocol for creating, composing, and evolving groups of interacting components.

- **Accidental complexities in configuration of middleware, specification and enforcement of policy.** Traditional DOC middleware provided very primitive mechanisms *i.e.*, low-level mechanisms for configuration of the middleware as well as specification of various policies. Since configuration and specification of policy was done using *imperative* techniques, it was typically done in the same language as that

of the implementation. This led to the configuration of the middleware becoming complex, tedious and error-prone.

- ***Ad hoc* deployment mechanisms.** Deployment of systems using traditional DOC middleware is also done in an *ad hoc* fashion using custom scripts. The scripts were usually targeted at deploying a single system, and hence had to be rewritten for every new system, or in some cases for even different versions of the same system. The development and maintenance of this *ad hoc* infrastructure for deployment was an unnecessary burden on distributed, real-time and embedded (DRE) system developers.

Thus, it is clear that system developers have to face significant challenges when building complex enterprise DRE systems using DOC middleware. One promising solution to alleviate the complexities of traditional DOC middleware is component middleware technologies.

I.2 Overview of Component Middleware

Component middleware technologies like EJB [134], Microsoft .NET [81], and the CORBA Component Model (CCM) [88] raised 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.1, key elements and benefits of component middleware technologies like CCM 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.

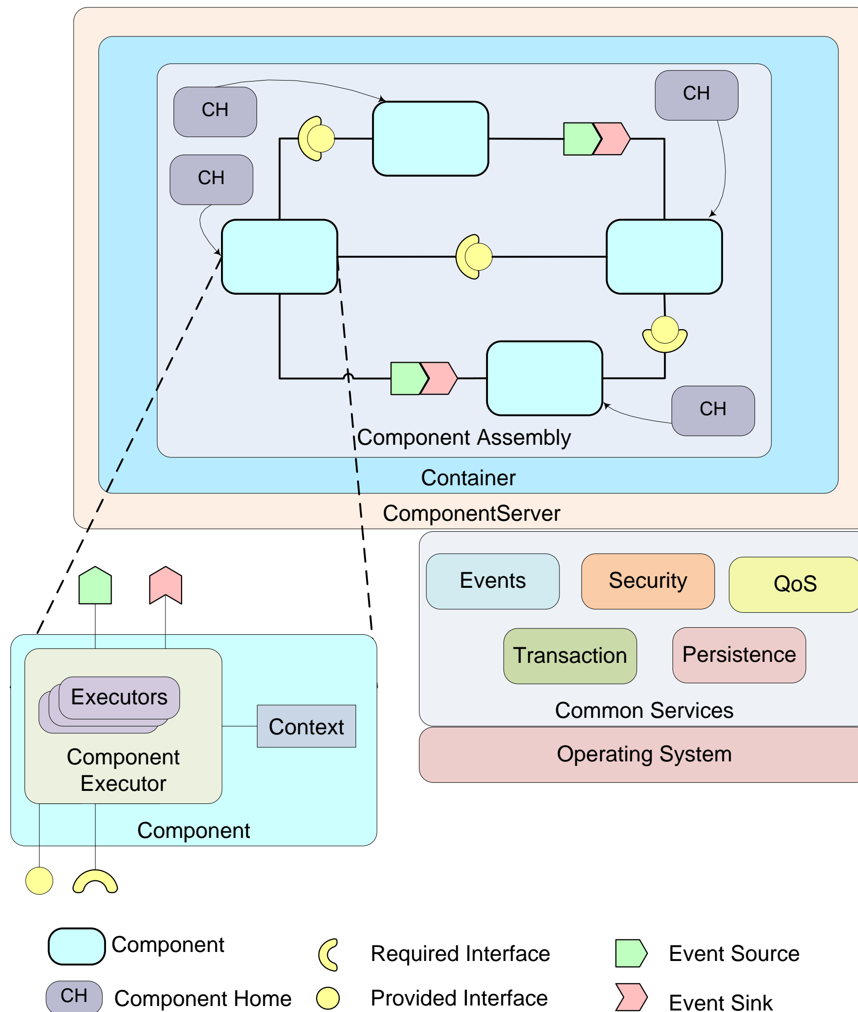


Figure I.1: Key Elements in the CORBA Component Model

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.

- **Component home**, which is a factory that creates and manages components. A component home provides flexibility in managing the lifecycle of components, including various strategies for component creation.

- **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 Quality-of-Service (QoS) policy configuration and enforcement, and are also the lowest unit at which policy is enforced; a container regulates shared access to the middleware infrastructure by the components.

- **Component context**, which links each component with its execution context and enables navigation between its different ports, as well as access to its connected neighbors. Component context eliminates coupling between a component implementation and its context, and hence, allows the reuse of a component in multiple execution contexts.

- **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.

- **Common Services**, which provide common middleware services, such as transaction, events, security and persistence. Common services implement the platform-specific aspects of transaction, events, security and persistence and allow components to utilize these services through the container.

Components interact with clients (including other components) via component ports. Component ports implement the Extension Interface pattern [118], 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 thus enable a component to use the functionality provided by facets of other components.

- **Event sources and sinks**, which define a standard interface for the Publisher/Subscriber pattern [17]. 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 [44] needed to support the Publisher/Subscriber 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.

Today's reusable class libraries and application framework platforms minimize the need to reinvent common and domain-specific middleware services, such as transactions, discovery, fault tolerance, event notification, security, and distributed resource management. For example, enterprise systems in many domains are increasingly developed using applications composed of distributed components running on feature-rich middleware frameworks. In component middleware, components are designed to provide reusable capabilities to a range of application domains, which are then composed into domain-specific assemblies for

application (re)use. The transition to component middleware is gaining momentum in the realm of enterprise DRE systems because it helps address problems of inflexibility and reinvention of core capabilities associated with prior generations of monolithic, functionally-designed, and stove-piped legacy applications. Legacy applications were developed with the precise capabilities required for a specific set of requirements and operating conditions, whereas components are designed to have a range of capabilities that enable their reuse in other contexts. As shown in Figure I.1, some key characteristics of component middleware that help the development of complex enterprise distributed systems include:

- Support for transparent remote method invocations,
- Exposing multiple views of a single component,
- Language-independent component extensibility,
- High-level execution environments that provide layer(s) of reusable infrastructure middleware services (such as naming and discovery, event and notification, security and fault tolerance),
- Tools that enable application components to use the reusable middleware services in different compositions.

I.3 Research Challenges

Although component middleware provide a number of advantages over previous technologies, several vexing problems remain. Some of the key challenges in developing, deploying and configuring component-based large-scale enterprise DRE systems using component middleware include:

1. **Lack of high-level system composition tools.** Although component middleware provides many tools for developing individual components using general purpose programming languages, there are few tools that exist for composing systems from

individual components. Thus, developers are still forced to deal with composition using previous generation tools like IDEs. Such tools lack the ability to check architectural constraints of the system and hence these problems don't show up until the system is deployed, or worse, after the system has been deployed, *i.e.*, at run-time. Since it costs much more to fix problems later in the software development cycle, lack of system composition tools is a big challenge to ensure successful adoption of component middleware technologies. A key research challenge is therefore the lack of system composition tools that focus on strategic architectural issues, such as system-wide correctness and performance, and provide an integrated view of the system.

2. **Complexity of declarative platform API and notations.** Over the years, complexity of the platform API have evolved faster than the ability of general-purpose languages to mask this complexity. For example, popular middleware platforms, such as EJB and .NET, contain thousands of classes and methods with many intricate dependencies and subtle side effects that require considerable effort to program and tune properly. Though these platforms expose *declarative* techniques for performing various system development and deployment tasks, the technologies chosen to express these *declarative* techniques are often not user-friendly. For example, platforms like .NET, EJB and CCM use XML [13] technologies as the notation for all *metadata* related to specification of policy, configuration of middleware as well as deployment of applications. Unlike traditional programming languages, XML syntax is very verbose. Due to the repetitive nature of the XML syntax, manual editing of XML files of even small sizes is very error-prone. A key research challenge is therefore the complexity of declarative platform API and notations of the metadata prevalent in the component middleware technologies.

3. **Overhead due to high-level abstraction in large-scale systems.** In any complex

system built using components, it is rare to find a single component that realizes a complex functionality on its own, *i.e.,* as a standalone component. We refer to indivisible, standalone components as *monolithic* components. Each monolithic component normally performs a single specific functionality to allow reuse of its implementation across the whole system. Thus, a number of inter-connected components are often composed together to create an assembly which realizes the complex functionality. Each such composition of components into an assembly results in a small and often unnoticeable overhead compared to implementing the functionality as a single component. By applying the same principle to composing the entire system, it is easy to get into a situation where a number of such small overheads add up to become a significant portion of the total execution time, thereby causing reduced QoS to clients. In worst cases, the application overhead can become so intolerable that the system is no longer usable. Thus, a key research challenge is the composition overhead of the high-level component middleware technologies when applied to large-scale systems.

4. **Complexity in integrating heterogeneous commercial-off-the-shelf (COTS) middleware technologies.** With the emergence of commercial-off-the-shelf (COTS) component middleware technologies, software system integrators are increasingly faced with the task of integrating heterogeneous enterprise distributed systems built using different COTS technologies. Although there are well-documented patterns and techniques for system integration using various middleware technologies [49, 139], system integration is still largely a tedious and error-prone manual process. To improve this process, component developers and system integrators must understand key properties of the systems they are integrating, as well as the integration technologies they are applying. A key research challenge is to provide automated and

reusable tools that enable system integrators to overcome the difficulties due to heterogeneity at the protocol level, the data format level, the implementation language level, and/or the deployment environment level.

I.4 Research Approach

To address the problems with the complexity of platforms and the inability of third-generation languages to alleviate this complexity and express domain concepts, we propose an approach that applies Model-Driven Engineering (MDE) technologies to the design, development and deployment of component-based enterprise DRE systems. As shown in Figure I.2, our approach involves a combination of:

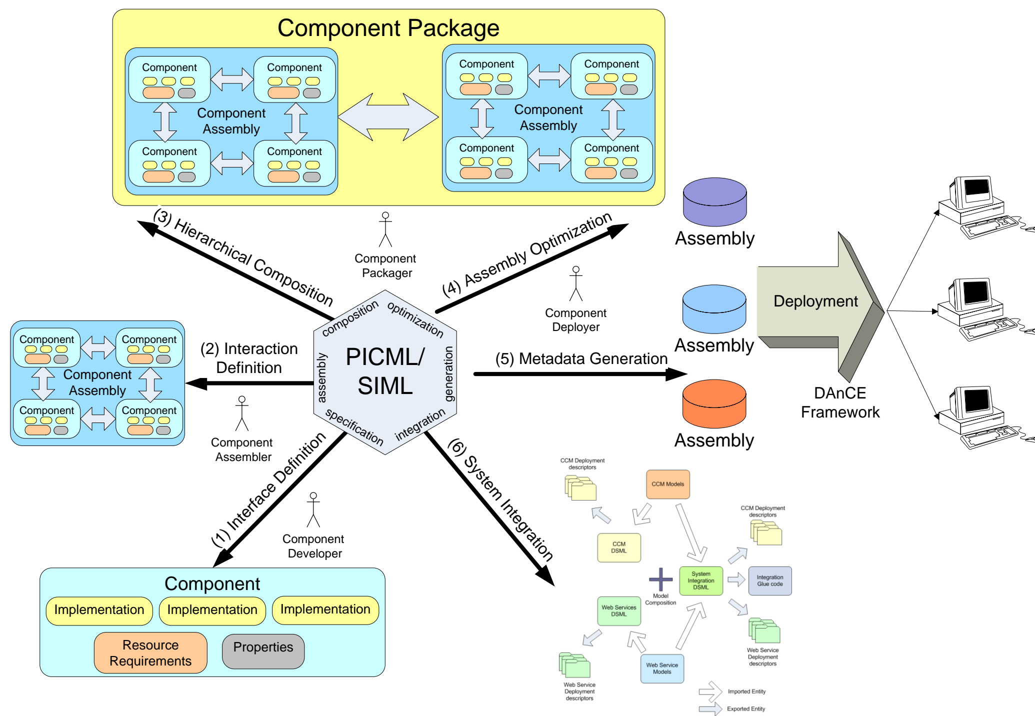


Figure I.2: Research Approach

- ***System Composition Techniques***, which includes a domain-specific modeling language (PICML) and associated tools to allow component interface definition, component interaction definition, multi-level composition of systems from individual components and automatic generation of metadata from the models. This research provides system level composition tools that allows composing systems from individual components. Chapter III describes the system composition tools in detail.

- ***System Optimization Techniques***, which includes an optimization framework (PAM) that uses the application context available in the models to optimize the execution and footprint of applications built using components. The proposed research optimizes away the composition overhead associated with component middleware technologies, by performing optimizations that were previously infeasible if operating at the middleware level. Chapter IV describes the optimization technologies in detail.

- ***System Integration Techniques***, which includes a domain-specific modeling language based integration framework (SIML) that enables the functional integration of heterogeneous middleware technologies like CCM and Web Services. This research creates a domain-specific modeling language for integration by composing multiple domain-specific modeling languages. Chapter V describes the integration technologies in detail.

I.5 Research Contributions

Our research on MDE-based composition, optimization and integration techniques has resulted in improved tool-support for component middleware. The key research contributions of our work on PICML, PAM and SIML are shown in Table I.1.

Table I.1: Summary Of Research Contributions

Category	Benefits
System Composition Techniques (PICML)	1. Demonstrates novel techniques for hierarchical system-level composition of COTS component middleware-based DRE systems using a MDE approach, 2. Provides a platform for expressing domain-specific constraints that are evaluated at design-time, effectively reducing the number of errors that are discovered after system deployment, 3. Improves developer productivity by automating key aspects of deployment of component-based systems including QoS configuration and automatic generation of syntactically valid metadata, 4. Serves as the foundation for a variety of MDE-based optimization and integration techniques.
System Optimization Techniques (PAM)	1. Proposes a new class of optimization techniques, “deployment-time” optimizations, 2. Describes three “deployment-time” optimization techniques for reducing the overhead of large-scale COTS component middleware-based systems, 3. Demonstrates the effectiveness of these techniques by implementing them in a MDE prototype to effect significant reduction in footprint and latency of systems in two different DRE domains, 4. Proposed optimization techniques are automatic (*i.e.*, do not require user intervention), non-intrusive (*i.e.*, do not require changes to existing systems or implementations) and preserve standards compliance.
System Integration Techniques (SIML)	1. Proposes a novel approach to enable functional integration of heterogeneous COTS component middleware technologies using (meta)model composition, 2. Demonstrates the effectiveness of the approach using a prototype composite system integration DSML, which is composed from DSMLs representing two different COTS middleware technologies, 3. Improves productivity of the system integrator by automatically generating “all” of the integration glue code directly from the models, 4. Provides a foundation for integrating multiple component middleware technologies.

I.6 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter II describes the research related to our work on composition, optimization and integration of component-based DRE systems and points out the gaps in existing research; Chapter III describes the PICML toolchain, our DSML-based approach to composition of component-based systems and shows how the high-level abstraction provided by PICML resolves the challenges related to composition; Chapter IV first describes the sources of overhead in typical component-based DRE systems and then describes our model-driven “deployment-time” optimization techniques implemented using the Physical Assembly Mapper (PAM) and evaluates the benefits of these techniques empirically; Chapter V describes the System Integration Modeling Language (SIML) and shows how (meta)model composition can be applied to integrate heterogeneous middleware technologies and automate key aspects of

functional integration by generating integration glue code automatically from the models; Chapter VI compares our work on PICML, PAM and SIML with other approaches and evaluates the benefits; and Chapter VII provides a summary of the research contributions, presents concluding remarks and outlines future research work.

CHAPTER II

EVALUATION OF ALTERNATE APPROACHES

This section summarizes alternate approaches to design, develop, compose, optimize and integrate component-based DRE systems. Our goal in this chapter is to survey the existing approaches. Detailed comparison of our work with the work described in this section is in Chapter VI.

II.1 Composition of Component-based Systems

System composition refers to composing a system by inter-connecting different individual components. Figure II.1 shows the different dimensions across which component composition is defined. These include:

- **Structural Dimension.** Structural dimensions are related to the structural properties of composition of a system. Systems can be sub-divided into two categories structurally:

  1. *Flat*, where connections between the components in the system are at the same level; all the components are defined at the same level, *i.e.,* they are peers,
  2. *Hierarchical*, where components are grouped together into assemblies which may further be composed of sub-assemblies, and connections between components exist at both levels.

- **Temporal Dimension.** Temporal dimension is related to the time at which the composition happens. Systems can be sub-divided into two categories in the temporal dimension:

  1. *Static*, where the components are combined together at build-time statically,

2. *Dynamic*, where the connections between components are orchestrated at deployment time using declarative metadata by a deployment engine.

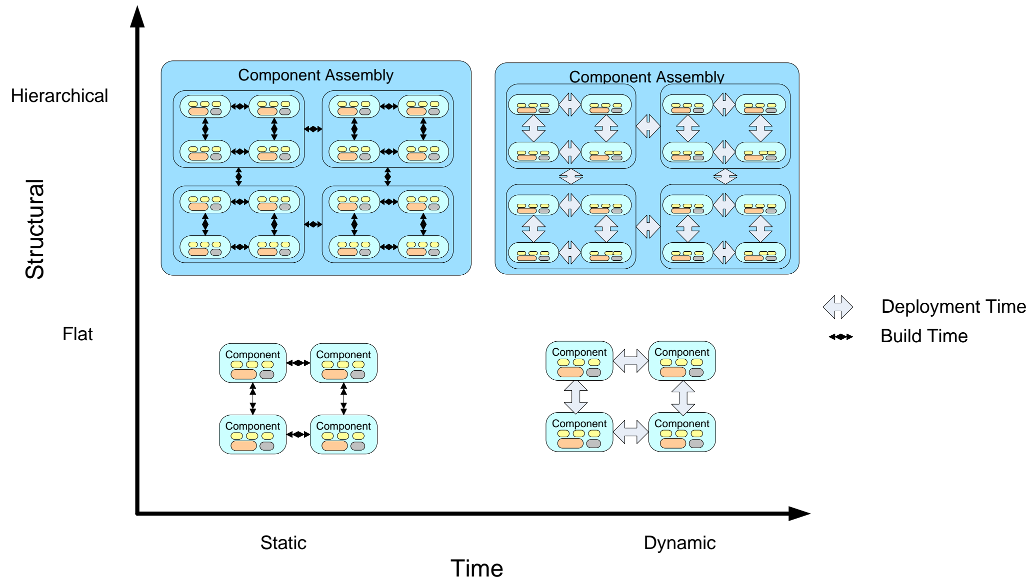


Figure II.1: Composition Dimensions

Component middleware promotes the development of libraries of pre-built and tested individual components, which offer different levels of capabilities and performance to clients. Although this paradigm increases the opportunities for systematic reuse, it can also complicate software lifecycle processes. In particular, component middleware shifts responsibility from software development engineers to other types of software engineers (such as software configuration and deployment engineers) and systems engineers. Software development engineers traditionally created entire applications in-house using top-down design methods that could be evaluated throughout the lifecycle. In contrast, software configuration/deployment engineers and system engineers must increasingly assemble enterprise, distributed systems by customizing and composing reusable components from existing frameworks, rather than building them from scratch. Thus, it is clear that system composition is becoming a critical part of enterprise DRE system development.

II.1.1 System Composition: Alternate Approaches

Composition of component-based systems has been studied extensively in the research community. Research on composition of component-based systems can be broadly categorized into four categories: (1) Component Development Environments, which deal with graphical environments that allow definition and composition of components, (2) Component Programming Techniques, which deal with improvements to programming languages and new programming methodologies and techniques to support component composition, (3) Functional Verification of Components, which deals with verification of components and compositions for various properties like QoS, deadlocks, real-time behavior, and (4) Declarative Notations to Express Design Intent, which deal with use of declarative notations for codifying various development, deployment, and configuration activities. These four areas are discussed below:

1. **Component Development Environments.** Various component development environments that provide a visual diagrammatic specification of complex models have been proposed including *Embedded Systems Modeling Language* (ESML) [58] and Ptolemy II [16]. WREN [73] is a component-based environment that emphasizes building systems composed of components retrieved from common software distribution sites as opposed to being completely developed in-house. The work also identifies some key requirements of component-based development environments including support for modular design, self-description, presence of global namespaces, support for application composition in addition to component development, support for component configuration, support for multiple views and reuse through reference to alleviate the maintenance problems.

2. **Component Programming Techniques.** A comprehensive collection of work related to Component-Based Software Engineering (CBSE) including (a) definition of components, component-models and services, (b) business case for components,

product-line architectures, software architectures, standard-based component models and (c) legal implications of component-based software is presented in [46]. Research on composition techniques at the programming language level include the work on Scala [99], extensions to languages to support collaboration-based designs using mixin-layers in a static fashion [127] as well as in a dynamic fashion [105]. The topic of generating product-line architectures has been addressed in [8] with an extension of this work to non-code artifacts in [9]. A seminal work on defining generative programming methodologies, tools and applications is [24]. Other work on composition techniques include the work on variability management in the context of product-line architectures in [79], which compares Feature-Oriented Programming(FOP) [108] with Aspect-Oriented Programming(AOP) [61]. Recent efforts [107] have also been focused on optimal strategies for composition of Web Services, where a number of publicly available Web Services are composed together to satisfy a high-level requirement. A summary of the existing techniques and requirements for composition of Web Services is [83].

3. **Functional Verification of Components.** Other approaches to assist composition of component-based systems through functional verification including model-checking are Cadena [45], *Virginia Embedded Systems Toolkit* (VEST) [129] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [64].

4. **Declarative Notations to Express Design Intent.** Declarative notations to express design intent is a hot topic in the research community. Manifestations of the declarative approach to configuration of the system starts from the operating system level and goes all the way up to component-based system packaging.

   Research on administration of personal computer systems [29] has focused on replacing the imperative updates to configuring and updating the operating system with declarative techniques, which rely on a system model as a function that can be applied

to a collection of system parameters to produce a statically typed, fully configured system instance. This research has been prototyped on Singularity OS [52].

On the other end of the spectrum is Pan [21], a high-level configuration language for system administration of a large number of machines, ranging from large clusters to desktops in large organizations. The approach taken to configuration is to store configuration information in a database in two alternate forms: a high-level declarative description (Pan) and a low-level XML-based notation. Automated tools are provided which convert Pan to XML.

Declarative notations have also been applied to the task of instrumenting a live system as implemented in DTrace [18]. DTrace is an online instrumentation facility which uses a declarative high-level language to describe predicates and actions at a given point of instrumentation. The DTrace mechanism has been integrated into the Solaris Operating system.

Emerging standards like Web Services are based on Web Services Description Language (WSDL) [19] to describe Web services starting with the messages that are exchanged between the service provider and requester. The messages themselves are described abstractly and then bound to a concrete network protocol and message format. A message consists of a collection of typed data items. An exchange of messages between the service provider and requester are described as an operation. WSDL uses XML Schema [11, 136] as the language for describing the service descriptions.

xADL [25] is an infrastructure for development of software architecture description languages (ADLs), which relies on using XML for description of the language itself. It provides a base set of reusable and customizable architectural modeling constructs and an XML-based modular extension mechanism. The primary goal of xADL is to

unify the plethora of ADL notations in prevalence, and to reduce the effort expended in building tools to support ADLs.

Recent research on network protocol design [114] has resulted in a generic application protocol kernel for connection-oriented, asynchronous interactions called BEEP. Messages are usually textual (structured using XML). BEEP is itself not a protocol for sending and receiving data directly. Rather, it allows definition of application protocol in a declarative fashion on top of it, reusing several mechanisms such as: asynchronous communications, transport layer security, peer authentication, channel multiplexing on the same connection, message framing, and channel bandwidth management.

Another declarative RPC protocol that is becoming popular is the Simple Object Access Protocol (SOAP) [141]. SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics. Rather, it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. This allows SOAP to be used in a large variety of systems ranging from messaging systems to RPC.

The .NET framework employs a number of declarative mechanisms to build, configure and deploy [112] systems. The use of declarative notations (built on top of XML) is pervasive in the .NET architecture. XML is used to configure the run-time behavior of not just shared libraries (assembly in .NET parlance) but also entire applications. XML is also used to describe the configuration of security policies at various levels of abstraction including application, machine or even enterprise.

Other standards-based component middleware including CCM [88] and EJB [134]

also define declarative mechanisms using XML for composition and assembly of components and component packages, as well as for orchestrating deployment of component systems [103].

Much emphasis of the related research has been on component programming models and languages to allow construction of components, *i.e.,* how to write better components, and functional verification of individual components. Another issue with related research is that a lot of *tool-specific component technologies* have been proposed, whereas there is a need for *component technology agnostic tools*. However, with the standardization of component programming models, and the availability of commercial-off-the-shelf (COTS) components, focus needs to shift away from "programming-in-the-small" to "composing-systems-in-the-large" and away from proprietary component models to standards-based component models. Another area which has not been given enough attention is the deployment of component-based systems and support for managing deployment artifacts. Section II.1.2 describes the key unresolved challenges in composition of component-based systems, which forms the basis for our research.

II.1.2 System Composition: Unresolved Challenges

As shown in Figure II.2, the challenges in building distributed systems are thus shifting from focusing on the construction of individual components to composition of systems from a large number of individual subcomponents, and ensuring correct configuration of the subcomponents. Composition of systems from individual components needs to ensure that the connections between components are compatible, as well as ensure that the deployment descriptors for the composed systems are valid.

Unfortunately, problems associated with composing systems from components often become manifest only during the integration phase. Problems discovered during integration are much more costly to fix than those discovered earlier in the lifecycle. A key research challenge is thus exposing these types of issues (which often have dependencies

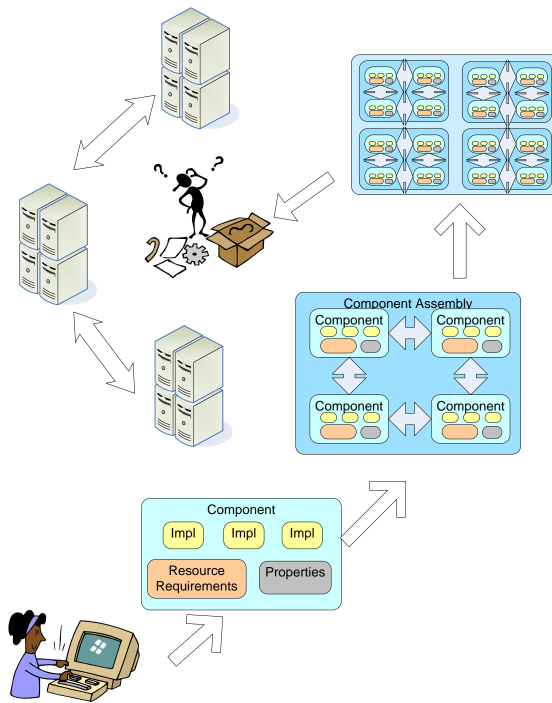


Figure II.2: Compositions of Systems from COTS Components

on components that are not available until late in development) earlier in the lifecycle, e.g., prior to the system integration phase. The following is a list of the unresolved challenges with composition of systems from standards based components:

1. **Lack of tool support for defining consistent component interactions.** Existing interface definition tools are primitive in the sense that the interfaces for different components are specified separately, and getting the interface definitions right involves tedious edit, compile, fix cycle. Also, while the individual interfaces themselves may be strongly typed, the lack of component interconnection information in interface definitions languages like CORBA Interface Definition Language (IDL) [88] and WSDL [19] makes the task of composing systems more difficult. This is because inconsistencies in the component interactions are not detected until either deployment-time, or in some cases until run-time.

2. **Lack of an integrated system view.** Traditional environments for component development provide a split view of the system where there is a design view, *e.g.*, Unified Modeling Language (UML) [96] models of the system, and there is a "code-centric" view, *e.g.*, Microsoft Visual Studio, Eclipse [98]. Thus, there is a lack of an integrated system view to help the system developers reason at the system-level.

3. **Lack of tool support for multi-level composition.** System developers also need tools that allow viewing the system at multiple levels of granularity (complexity). Also, when one portion of a system changes, the change propagation is done in an *ad hoc* fashion which is tedious and error-prone when done manually.

4. **Lack of context-aware policy specification mechanisms.** Components of a system might need to be configured with different parameters based on the usage context of a component. However, existing integrated development environments (IDEs) lack support for context-aware policy specifications which results in maintenance issues when the number of components and the number of contexts in which a component is used in a system grows.

5. **Lack of scalable composition techniques.** Most graphical environments (*e.g.*, Microsoft Visual Studio, Eclipse) and composition techniques are effective when the number of components in a system number in the tens or hundreds. However, when the number of components in a system is in the thousands, it is extremely unproductive to perform composition activities manually. Even if a tool environment provides such a capability, it may not be customizable. Existing mechanisms to customize an environment involve writing plugins, or addons [72], which assumes familiarity with the tool environment itself and is an extra burden on system developers.

6. **Complexity of Declarative Notations.** A key research challenge is the accidental

complexities of the declarative notations required to configure the component middleware. Although the move towards declarative notations is an advance over previous generation imperative techniques, the declarative techniques have chosen to use tool-friendly technologies like XML as the medium for expression of design intent. XML is non-intuitive and error-prone to write manually (with or without tool support). Any changes to a system requires modification to an XML document which is a cumbersome process. Thus by choosing XML as the underlying notation for declarative techniques, the problems associated with imperative system configuration have just been shifted into a different space, *i.e.,* configuration using XML.

7. **Lack of support for system evolution.** Another research challenge is maintaining and evolving the declarative metadata associated with a system. Any complex system will undergo a number of changes (minor and major) as part of it's evolution, and hence it is critical that the declarative metadata also evolve with the system. *Ad hoc* and *naive* approaches to management of metadata will result in problems during deployment time, or even at run-time, both of which are costly to fix.

II.2 Optimization of Component-based Systems

Over the past five decades, software researchers and developers have been creating abstractions that (1) help them to program in terms of their design intent rather than in terms of the underlying computing environments (e.g., CPU, memory, and network devices) and (2) shield them from the complexities of these environments. From the early days of computing, these abstractions included both language and platform technologies. For example, early programming languages, such as assembly and FORTRAN, shielded developers from complexities of programming with machine code. Likewise, early operating system platforms, such as OS/360 and UNIX, shielded developers from complexities of programming directly to hardware. More recently, higher-level languages (such as C++, Java, and

C#) and platforms (such as component middleware) have further shielded application developers from the complexities of the hardware. Although existing languages and platforms raised the level of abstraction, they can also incur additional overhead. For example, common sources of overhead in component middleware include marshaling/de-marshaling costs, data copying and memory management, static footprint overhead due to presence of code paths to deal with every possible use case, dynamic footprint overhead due to redundant run-time infrastructure helper objects, the endpoint/request de-multiplexing, and context switching/synchronization overhead. Although some implementations of component middleware try to minimize this overhead, there is a limit to the optimizations done by the middleware developers. In particular, middleware developers can only apply optimizations that are applicable across all applications in a particular domain, which effectively limits the number of valid optimizations performed by default.

II.2.1 System Optimization: Alternate Approaches

Optimizing middleware to increase the performance of applications has long been a goal of system researchers [20, 33, 36, 39, 76]. In this section we will explore a representative sample of the research that has been applied to optimizing middleware for component-based systems. Optimization techniques to improve application performance can be categorized along two dimensions: (1) the layer at which the optimization is applied, *e.g.*, whether the optimization is restricted to the middleware layer alone or spans multiple layers, including applications above the middleware, (2) the time at which such optimization techniques are applied, *i.e.* design/development-time, run-time or deployment-time. Research along the different dimensions of optimization can be summarized as follows:

1. **Design/development-time approaches**. Design-time approaches to component middleware optimization include static configuration of CIAO [131], context-specific

middleware specializations for product-line architectures [65], application of Aspect-Oriented Programming (AOP) techniques to automatically derive subsets of middleware based on use case requirements [50], and modification of applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [101]. In addition, middleware has been synthesized in a "just-in-time" fashion by integrating source code analysis, and inferring features and synthesizing implementations [151].

2. **Run-time approaches.** Research on approaches to optimizing middleware at run-time has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context [31], dynamic adaptation of desired behavior as described in QuO [153], domain-specific middleware scheduling optimizations for DRE systems [42], using feedback control theory to affect server resource allocation in Internet servers [152] as well as to perform real-time scheduling in Real-time CORBA middleware [70].

   Run-time approaches to application-specific optimizations have also focused on data replication for edge services, *i.e.*, replicating servers at geographically distributed sites [41], optimizing web services utilizing reflective techniques encapsulated in the request metadata [85], and improving algorithms for event ordering within component middleware by making use of application context information available in models [126].

   Research on alternate component middleware like EJB have focused on automating the performance management [30] of applications by employing a performance monitoring framework which works in collaboration with a performance anomaly detection framework. By relying on redundant implementation of components, *i.e.*,

a component with the same functionality but optimized for different run-time environments, implementations can be swapped for more optimal ones depending on the anomalies detected.

3. **Deployment-time approaches.** Deployment-time optimization research includes BluePencil [69], which is a framework for deployment-time optimization of web services. BluePencil focuses on optimizing the client-server binding selection using a set of rules stored in a policy repository and rewriting the application code to use the optimized binding.

One common theme with the research on middleware optimization has been the use of run-time reflection [10] to adapt the behavior of the middleware such that application performance is optimized. Although this may be suitable for some system, not all enterprise DRE systems can afford the luxury of run-time reflection in the critical path. Another theme with the research on optimizations is the requirement for multiple implementations to be provided to the middleware to choose from. This strategy is not entirely application transparent, and imposes an extra burden on the system developers. Finally, one of the important missing pieces in the optimization research is the lack of a high-level notation to guide the optimization frameworks, *i.e.*, there is no intermediate abstract syntax tree (AST) of the application that is available to the middleware to use as a basis for performing optimizations.

II.2.2 System Optimization: Unresolved Challenges

One of the biggest factors in affecting system performance is not a single significant decrease (which are usually easy to identify quickly) but a slew of small decreases. It is hard to notice this overhead creep into the system without a sophisticated Distributed Continuous Quality Assurance [77] infrastructure, and considerable diligence on part of the developers, which does not scale up well to large-scale systems. Component middleware standards

do not advocate any standard optimizations since it is not possible to perform them in the middleware without the knowledge of the application context, *i.e.*, such optimizations are not domain invariants. Tools that automatically optimize component assemblies (compositions) are not prevalent. It is hard to both identify and optimize component implementations manually, since the usage of components tends to span multiple hierarchies in any complex system. Further, an optimization that is applicable in one context may not be applicable in another context. Thus, it is not possible to perform these optimizations in isolation, but rather one should perform them based on every unique use case. Finally, performing these optimizations manually by hand becomes infeasible with system evolution.

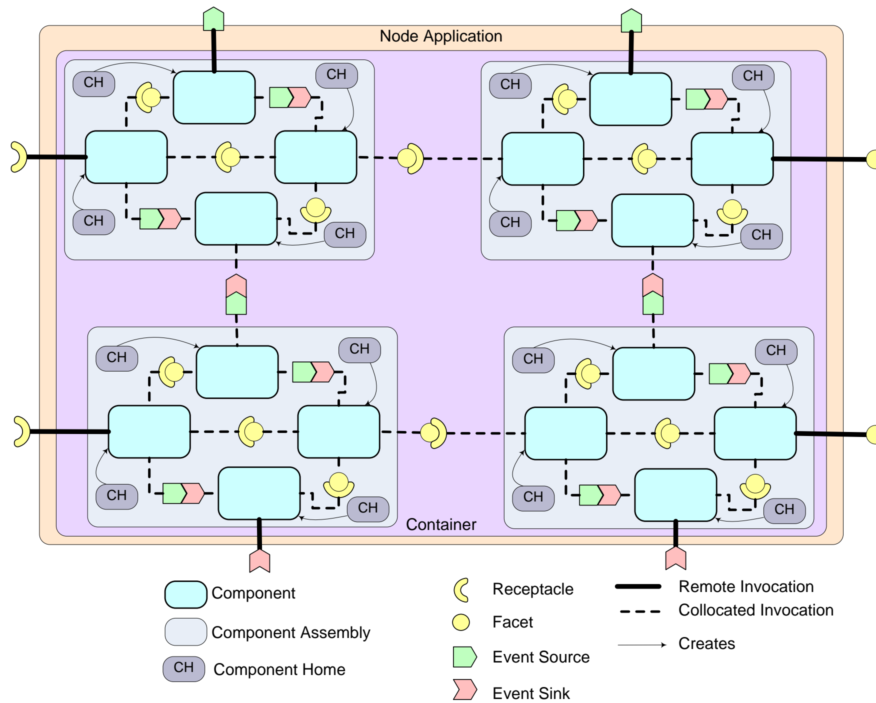


Figure II.3: Composition Overhead in Component Assemblies

The following is a list of unresolved challenges with respect to application-specific optimizations when using standards-based component middleware:

1. **Lack of application context.** A significant problem with component middleware is

the number of missed optimization opportunities in the middleware due to lack of application context information. For example, when component middleware generates glue code to facilitate remote method invocations, it generates code with the assumptions that every component is remote, as shown in Figure II.3. Often, however, all components that make up a subsystem are deployed onto the same node or even in the same process. Since the application composition information is not available during glue code generation, the middleware generated glue code is often inefficient. A key research challenge is therefore to eliminate the overhead of applying high-level abstractions like component middleware *automatically* to ensure that the system still meets the desired performance requirements.

2. **Overhead of platform mappings.** Platform mappings for component middleware (*i.e.*, code required to hook up a component into the run-time), are typically defined with the assumption that every component is (or can be) remote. However, in certain cases, blind adherence to the platform mapping can result in significant overhead for applications built using component middleware. For example, in Figure II.3 it is clear that the components that are internal to the assembly do not have any connection with other components outside. However, a default implementation of the internal components will generate a single factory object per component instance which is responsible for creation of components. This is wasteful in terms of both static (for each component type) and dynamic (for each component instance) footprint. A key research challenge is to recognize such anomalies and optimize away the overhead.

3. **Issues due to QoS (mis-)configuration.** Although component middleware technologies provide declarative methods to configure QoS, large-scale component systems can observe sub-optimal performance due to QoS mis-configuration. This happens not only due to incorrect QoS configuration at the individual component level, but can also happen at the level of component assemblies due to duplicate QoS policies.

At the level of component assemblies, associating QoS policies with components in an individual fashion can lead to increasing the number of effective QoS policies required to deploy a system. Since the number of unique QoS policies determines the run-time resources like the number of containers created, this can result in increasing the resource demands on the system. A key research challenge is to minimize the number of run-time resources associated with QoS configuration.

II.3 Integration of Component-based Systems

With the maturation of commercial-off-the-shelf (COTS) component middleware technologies, such as Enterprise Java Beans (EJB) [134], CORBA Component Model (CCM) [88], and Microsoft .NET Framework [81], software developers are increasingly faced with the task of integrating heterogeneous enterprise distributed systems built using different COTS technologies, rather than just integrating proprietary software developed in-house. Although there are well-documented patterns [49] and techniques [14] for integrating systems via various component middleware technologies, system integration is still largely a tedious and error-prone manual process. To improve this process, component developers and system integrators must understand key properties of the integration technologies they are applying and the systems they are integrating.

II.3.1 System Integration: Alternate Approaches

This section surveys the technologies that provide the context of our work on system integration in the domain of enterprise distributed systems. We classify techniques and tools in the integration space according to the role played by the technique/tool in system integration as follows:

1. **Integration evaluation tools** like IBM's WebSphere [54] enable system integrators to specify the systems/technologies being integrated and evaluate the integration strategy and tools used to achieve integration; system execution modeling [128]

tools, such as CoWorker Utilization Test Suite (CUTS) [48], help developers conduct "what if" experiments to discover, measure, and rectify performance problems early in the lifecycle (*e.g.*, in the architecture and design phases), as opposed to the integration phase.

2. **Integration design tools.** Object Management Group (OMG)'s UML profile for Enterprise Application Integration (EAI) [91] defines a Meta Object Facility (MOF) [93] for collaboration and activity modeling.

3. **Integration patterns** [139] provides guidance to system integrators in the form of best patterns and practices, with examples using a particular vendor's products. Common integration patterns, with an emphasis on system integration via asynchronous messaging using different commercial products are catalogued in [49].

4. **Resource adapters** are used during integration to transform data and services exposed by service producers to a form amenable to service consumers. Examples include *data transformation* (mapping from one schema to another), *protocol transformation* (mapping from one network protocol to another), or *interface adaptation* (which includes both data and protocol transformation). Existing standards (such as the Java Messaging Specification [132] and J2EE Connector Architecture Specification [82]) and tools (such as IBM's MQSeries [53]) provide the architectural framework for performing resource adaptations.

5. **Integration frameworks.** The semantic web and the Web Ontology Language (OWL) [22] have focused on the composition of services from unambiguous, formal descriptions of capabilities as exposed by services on the Web. Research on service composition has focused largely on automation and dynamism [107], integration on large-scale "system-of-systems," such as the Grid [37]. Other work has focused on optimizing service compositions such that they are "QoS-aware" [149]; in such "QoS-aware"

compositions, a service is composed from multiple other services taking into account the QoS requirements of clients.

6. **Integration quality analysis.** Research on QoS issues associated with integration has yielded languages and infrastructure for evaluating *service-level agreements*, which are contracts between service providers and consumers that define the obligations of the parties involved and specify what measures to take if service assurances are not satisfied. Examples include (1) the Web Service-Level Agreement language (WSLA) [71] framework, (2) the WS-Agreement framework [100], and (3) Rule-Based Service Level Agreement [106].

A common problem with the integration tools is that the system integrators are still required to do low-level integration activities like writing glue code, and configuring resource adapters. There is a lack of automation in integration related activities.

II.3.2 System Integration: Unresolved Challenges

Despite the benefits of component middleware, key challenges in functional integration of systems remain unresolved when integrating large-scale systems developed using heterogeneous COTS middleware. These challenges include:

1. **Integration Design**, which involves choosing the right abstraction for integration. Although integration evaluation tools help identify potential integration problems and evaluate the overall integration strategy, they do not replace the actual task of integration itself since these tools use simulation-/emulation-based abstractions of the actual systems. Integration patterns themselves do not directly provide tools for integration, but instead provide pattern-based guidance to apply existing tools to achieve more effective integration. A key research challenge is to build tools that help system integrators in integration design and allows them to make different trade-offs.

2. **Interface Mapping**, which reconciles the source and target datatypes. Existing integration design tools provide limited support for interface mapping by generating stubs and skeletons, for facilitating interface mapping, and performing protocol mapping. A key research challenge is to provide automatic mapping of the interfaces between different heterogeneous technologies.

3. **Technology Mapping**, which reconciles various low-level issues when integrating heterogeneous COTS technologies. Existing standards for performing resource adaptations, however, approach the integration from a middleware and programming perspective, *i.e.*, system integrators must still handcraft the glue code that invokes the resource adapter frameworks to connect system components together. A key research challenge is to enable automatic generation of the integration glue code thereby relieving the system integrators to write more code to integrate existing systems.

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 composition capabilities of PICML resolves these limitations. Chapter IV describes the issues with optimization in component-based systems in detail and explains how they are resolved. The lack of simplification and automation in resolving these challenges significantly hinders effective system integration. Chapter V describes these challenges in greater detail and explains the approaches to resolve these challenges.

CHAPTER III

TECHNIQUES FOR COMPOSING COMPONENT-BASED SYSTEMS

The trend towards developing and reasoning about DRE systems via components provides many advantages compared with earlier forms of infrastructure software. For example, components provide higher-level abstractions than operating systems, third-generation programming languages, and earlier generations of middleware, such as distributed object computing (DOC) middleware. In particular, component middleware, such as CCM, J2EE, and .NET, supports multiple views per component, transparent navigation, greater extensibility, and a higher-level execution environment based on containers, which alleviate many limitations of prior middleware technologies. The additional capabilities of component-based platforms, however, also introduce new complexities associated with composing and deploying DRE systems using components, including (1) the need to design consistent component interface definitions, (2) the need to specify valid interactions and connections between components, (3) the need to generate valid component deployment descriptors, (4) the need to ensure that requirements of components are met by target nodes where components are deployed, and (5) the need to guarantee that changes to a system do not leave it in an inconsistent state. The lack of simplification and automation in resolving the challenges outlined above can significantly hinder the effective transition to – and adoption of – component middleware technology to develop DRE systems.

To address the needs of DRE system developers outlined above, we have developed the *Platform-Independent Component Modeling Language* (PICML). PICML is an open-source domain-specific modeling language (DSML) available for download at www.dre.vanderbilt.edu/cosmic that enables developers of component-based DRE systems to define application interfaces, QoS parameters, and system software building rules, as well as generate valid XML descriptor files that enable automated system deployment.

PICML also provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling of component assemblies.

PICML is designed to help bridge the gap between design-time verification and model-checking tools (such as Cadena [45], VEST [129], and AIRES [64]) and the actual deployed component implementations. PICML also provides higher-level abstractions for describing DRE systems, using component models that provides a base for (1) integrating analysis tools that reason about DRE systems and (2) platform-independent generation capabilities, *i.e.*, generation that can be targeted at multiple component middleware technologies, such as CCM, J2EE, and ICE [150].

III.1 Overview of PICML

Model-Driven Engineering (MDE) [43] is a paradigm that focuses on using models in most system development activities, *i.e.*, models provide input and output at all stages of system development until the final system itself is generated. A key capability supported by the MDE paradigm is the definition and implementation of *domain-specific modeling languages* (DSMLs), which can be viewed as a five-tuple [59] consisting of: (1) concrete syntax (C), which defines the notation used to express domain entities, (2) abstract syntax (A), which defines the concepts, relationships and integrity constraints available in the language, (3) semantic domain (S), which defines the formalism used to map the semantics of the models to a particular domain, (4) syntactic mapping (M_C: A→C), which assigns syntactic constructs (*e.g.*, graphical and/or textual) to elements of the abstract syntax, and (5) semantic mapping (M_S: A→S), which relates the syntactic concepts to those of the semantic domain.

Crucial to the success of DSMLs is *metamodeling* and *auto-generation*. A *metamodel* defines the elements of a DSML, which is tailored to a particular domain, such as the domain of avionics mission computing or emergency response systems. Auto-generation

involves automatically synthesizing artifacts from models, thereby relieving DSML users from the specifics of the artifacts themselves, including their format, syntax, or semantics. Examples of such artifacts include (but are not limited to), code in some programming language and/or descriptors, in formats such as XML, that can serve as input to other tools.

To support development of DRE systems using MDE, we have defined the *Platform-Independent Component Modeling Language* (PICML) using the *Generic Modeling Environment* (GME) [67]. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is meta-programmable, the same environment used to define PICML is also used to build models, which are instances of the PICML metamodel. Sidebar 1 describes the features of GME that allow development of DSMLs.

At the core of PICML is a DSML (defined as a *metamodel* using GME) for describing components, types of allowed interconnections between components, and types of component metadata for deployment. The PICML metamodel defines ~115 different types of basic elements, with 57 different types of associations between these elements, grouped under 14 different folders. The PICML metamodel also uses the OMG's Object Constraint Language (OCL) [146] to define ~222 constraints that are enforced by GME's constraint manager during the design process.

Using GME tools, the PICML metamodel can be compiled into a *modeling paradigm*, which defines a domain-specific modeling environment. From the PICML metamodel, ~20,000 lines of C++ code (which represents the modeling language elements as equivalent C++ types) are generated. This generated code allows manipulation of modeling elements, *i.e.*, instances of the language types using C++, and forms the basis for writing *model interpreters*, which traverse the model hierarchy to perform various kinds of generative actions, such as generating XML-based deployment plan descriptors. PICML currently has ~8 interpreters using ~222 generated C++ classes and ~8,000 lines of hand-written C++ code that traverse models to generate the XML deployment descriptors (described in

Sidebar 1: Generic Modeling Environment

The Generic Modeling Environment (GME) is an open-source, visual, configurable design environment for creating DSMLs and program synthesis environments, available for download from escher.isis.vanderbilt.edu/downloads?tool=GME. A unique feature of GME is that it is *meta-programmable*, which means that it can not only build DSMLs, but also build models that conform to a DSML. In fact, the environment used to build DSMLs in GME is itself built using another DSML (also known as the *meta-metamodel*) called "MetaGME," which provides the following elements to define a DSML:

- **Project**, which is the top-level container in a DSML,
- **Folders**, which are used to group collections of similar elements together,
- **Atoms**, which are the indivisible elements of a DSML, and used to represent the leaf-level elements in a DSML,
- **Models**, which are the compound objects in a DSML, and are used to contain different types of elements like References, Sets, Atoms, and Connections (the elements that are contained by a Model are known as *parts*),
- **Aspects**, which are used to provide a different viewpoint of the same Model (every part of a Model is associated with an Aspect),
- **Connections**, which are used to represent relationships between the elements of the domain,
- **References**, which are used to refer to other elements in different portions of a DSML hierarchy (unlike Connections, which can be used to connect elements within a Model),
- **Sets**, which are containers whose elements are defined within the same aspect and have the same container as the owner.

Sidebar 2) needed to support the OMG D&C specification [90]. Each interpreter is written as a DLL that is loaded at run-time into GME and executed to generate the XML descriptors based on models developed by the component developers using PICML.

III.2 Composition using QoS-enabled Component Middleware - A Case Study

To motivate and explain the features in PICML, we use a running example of a representative DRE system designed for emergency response situations (such as disaster recovery

efforts stemming from floods, earthquakes, hurricanes) and consists of a number of interacting subsystems with a variety of DRE QoS requirements. Our focus in this chapter is on the unmanned aerial vehicle (UAV) portion of this system, which is used to monitor terrain for flood damage, spot survivors that need to be rescued, and assess the extent of damage. The UAV transmits this imagery to various other emergency response units, including the national guard, law enforcement agencies, health care systems, firefighting units, and utility companies.

Developing and deploying emergency response systems is hard. For example, there are multiple modes of operation for the UAVs, including aerial imaging, survivor tracking, and damage assessment. Each of these modes is associated with a different set of QoS requirements. For example, a key QoS criteria involves the latency requirements in sending images from the flying UAVs to ground stations under varying bandwidth availability. Similar QoS requirements manifest themselves in the traffic management, rescue missions, and fire fighting operations.

In conjunction with colleagues at BBN Technologies and Washington University, we have developed a prototype of the UAV portion of the emergency response system [121] described above using the CCM and Real-time CORBA capabilities provided by Component-Integrated ACE ORB(CIAO) [145]. CIAO extends our previous work on *The ACE ORB* (TAO) [117] by providing more powerful component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the OMG CCM [88] and D&C [90] specifications. Moreover, CIAO integrates the CCM capabilities outlined below with TAO's Real-time CORBA [117] features, such as thread-pools, lanes, and client-propagated and server-declared policies. In this section, we first briefly explain the motivation for using CCM to develop the UAV portion of the emergency response system. We then describe the challenges in developing this system using CCM, and then show how we resolved these challenges by applying a MDE approach using PICML.

The CORBA Component Model (CCM) is an OMG specification that standardizes the

development of component-based applications in CORBA. Since CCM uses CORBA's object model as its underlying object model, developers are not tied to any particular language or platform for their component implementations. The CIAO project is based on CCM rather than other popular component models, such as EJB or .NET, since CORBA is the only COTS middleware that has made a substantial progress in satisfying the QoS requirements of DRE applications. For instance, the OMG has adopted the following DRE-related specifications in recent years:

- **Minimum CORBA** [94], which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded system applications.
- **Real-time CORBA** [95], which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end.
- **CORBA Messaging** [87], which exports additional QoS policies, such as asynchronous invocations, timeouts, request priorities, and queuing disciplines, to DRE applications.
- **Fault-tolerant CORBA** [92], which uses entity redundancy of objects to support replication, fault detection, and failure recovery.

These QoS specification and enforcement capabilities are essential to support DRE applications.

III.2.1 Challenges in Composing the UAV Application

The components in this UAV application are shown in Figure III.1 and the steps involved in this effort are described below:

1. Identify the components in the system, and define their interfaces, which involves defining component ports and attributes, using the CORBA 3.x IDL features provided by

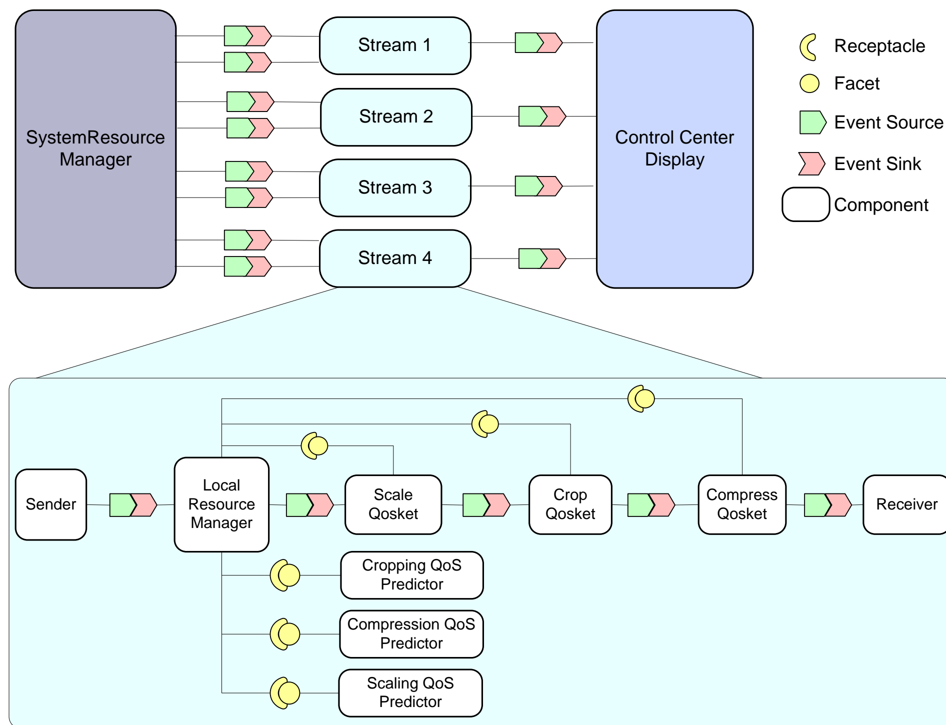


Figure III.1: Emergency Response System components

CCM. In the UAV example, each UAV is associated with a stream of images. Each image stream is composed of `Sender`, `Qosket`, and `Receiver` components. `Sender` components are responsible for collecting the images from each image sensor on the UAV. The `Sender` passes the images to a series of `Qosket` [145] components that perform operations on the images to ensure that the QoS requirements are satisfied. Some Qosket components include `CompressQosket`, `ScaleQosket`, `CropQosket`, `PaceQosket`, and a `DiffServQosket`. The final `Qosket` then passes the images to a `Receiver` component, which collects the images from the UAV and passes them on to a display in the control room of the emergency response team.

Each `Sender`, `Receiver`, and the various `Qosket` components pass images via CCM event source and sink ports. There are also manager components that define policies, such as the relative importance of the different mission modes of each UAV. These policies

in turn modify existing resource allocations by the `Qosket` components. For example, the global `SystemResourceManager` component monitors resource allocation across all the UAVs that are operational at any moment, and is responsible for communicating policy decisions from the control center to each UAV by triggering mode changes. The per-stream `LocalResourceManager` component is responsible for instructing the `Qosket` components to adapt their internal QoS requirements according to the mode in which the UAV is currently operating.

2. Define interactions between components, which involves keeping track of the types of each component's ports and ensuring that components which must be interconnected have matching ports defined. In the UAV example, this involves connecting the different components that comprise a single stream in the correct order since some components (such as `DeCompressQosket`) do the reverse of an operation performed by another component (such as `CompressQosket`). The manager components need to be connected to receive monitoring information about the existing QoS in each stream of image data.

3. Compose the UAV application by defining CCM deployment descriptors, which involves selecting a set of component implementations from a library of available implementations, describing how to instantiate component instances using these component implementations, and specifying connections between component instances. In the UAV example, this first involves combining the different components that comprise a single stream of images into a single assembly, represented by an XML descriptor. The complete UAV application is then created by making copies of this file to represent each UAV in flight.

4. Deploy the UAV application onto its runtime platform, which involves ensuring that the implementation artifacts and the associated deployment descriptors are available on the actual target platform, and initiating the deployment process using the standard OMG D&C [90] framework and tools. In the UAV example, this involves taking the hand-written XML descriptors and deploying the application using these descriptors as input.

5. Refine the component-based UAV application, which involves making changes to

existing component interface definitions or adding new component types, as part of enhancing the initial UAV application prototype. In the UAV example, this involves adding or removing a `Qosket` component in the pipeline for a single stream depending on results from empirical evaluation of the system.

One of the challenges of using just component middleware is that errors often go undetected until late in the development cycle. When these errors are eventually detected, moreover, repairing them often involves backtracking to multiple prior life-cycle steps, which impedes productivity and increases the level of effort. As a result, the advantages of transitioning from DOC middleware to component middleware can be significantly obstructed, without support from higher-level tools and techniques. These observations underscore the importance of enhancing design-time support for DRE systems built using component middleware, as well as the importance of automating the deployment of such systems.

III.2.2 Resolving UAV Composition Challenges with PICML

As discussed in [144], the use of QoS-enabled component middleware to develop the UAV application significantly improved upon an earlier DOC middleware prototype of this application [111]. In the absence of model-driven development (MDE) tool support, however, a number of significant challenges remain unresolved when using component middleware. The remainder of this section describes five key challenges that arose when the UAV application was developed using CCM and CIAO, and examines how key features of PICML can be applied to address the limitations associated with developing QoS-enabled component middleware-based DRE systems, such as the UAV application.

We use CCM and CIAO as the basis for our research because it is layered on top of Real-Time CORBA, which provides significant capabilities for satisfying end-to-end QoS requirements of DRE systems [117]. There is nothing inherent in PICML, however, that limits it to CCM or CIAO. Likewise, the challenges described below are generic to component middleware, and not deficiencies of CCM or CIAO. For example, both J2EE and

Microsoft .NET use XML to describe component assemblies, so the challenges we describe apply to them, as well.

III.2.2.1 Accidental Complexities in Component Interface Definition

IDL for CCM (*i.e.*, CORBA 3.x IDL) defines extensions to the syntax and semantics of CORBA 2.x IDL. Every developer of CCM-based applications must master the differences between CORBA 2.x IDL and CORBA 3.x IDL. For example, while CORBA 2.x interfaces can have multiple inheritance, CCM components can have only a single parent, so equivalent units of composition (*i.e.*, interfaces in CORBA 2.x and components in CCM) can have subtle semantic differences. Moreover, any component interface that needs to be accessed by component-unaware CORBA clients should be defined as a *supported* interface as opposed to a *provided* interface.

In any system that transitions from an object-based architecture to a component-based architecture, there is a likelihood of simultaneous existence of simple CORBA objects and more sophisticated CCM components. Design of component interfaces must therefore be done with extra care. In the UAV application, for example, though the `Qosket` components receive both allocation events from the resource managers and images from the `Sender` and other `Qosket` components, they cannot inherit from base components implementing each functionality. Similarly, the `Receiver` component interface needs to be defined as a *supported* interface, rather than a *provided* interface.

Solution → Visual Component Interface Definition.

A set of component, interface, and other datatype definitions may be created in PICML using either of the following approaches:

- **Adding to existing definitions imported from IDL**. In this approach, existing CORBA software systems can be easily migrated to PICML using its *IDL Importer*, which takes any number of CORBA IDL files as input, maps their contents to the appropriate PICML model elements, and generates a single XML file that can be imported

into GME as a PICML model. This model can then be used as a starting point for modeling assemblies and generating deployment descriptors.

- **Creating IDL definitions from scratch**. In this approach, PICML's graphical modeling environment provides support for designing the interfaces using an intuitive "drag and drop" technique, making this process largely self-explanatory and independent of platform-specific technical knowledge. Most of the grammatical details are implicit in the visual language, *e.g.*, when the model editor screen is showing the "scope" of a definition, only icons representing legal members of that scope will be available for dragging and dropping.

CORBA IDL can be generated from PICML, enabling generation of software artifacts in languages having a CORBA IDL mapping. For each logically separate definition in PICML, the generated IDL is also split into logical file-type units. PICML's interpreter will translate these units into actual IDL files with `#include` statements based on the inter-dependencies of the units detected by the interpreter. PICML's interpreter will also detect requirements for the inclusion of canonical CORBA IDL files and generate them as necessary.

Application to the UAV example scenario. By modeling the UAV components using PICML, the problems associated with multiple inheritance, semantics of IDL are flagged at design time. By providing a visual environment for defining the interfaces, PICML resolves many problems described in Section III.2.2.1 associated with definition of component interfaces. In particular, by modeling the interface definitions, PICML alleviates the need to model a subset of interfaces for analysis purposes, which has the added advantage of preventing a skew between the models of interfaces used by analysis tools and the interface used in implementations. It also removes the effort needed to ensure that the IDL semantics are satisfied, resulting in a reduction in effort associated with interface definition.

III.2.2.2 Defining Consistent Component Interactions

Even if a DRE system developer is well-versed in CORBA 3.x IDL, it is hard to keep track of components and their types using plain IDL files, which are text-based and provide no visual feedback, *i.e.*, to allow visual comparison to identify differences between components. Type checking with text-based files involves manual inspection, which is error-prone and non-scalable. CCM defines the following valid interactions between the *ports* — Facets, Receptacles, Event Sources and Event Sinks — of a component: Facet-Receptacle interactions, and Event Source-Event Sink interactions. However, an IDL compiler will not be able to catch mismatches in the port types of two components that need to be connected together, since component connection information is not defined in IDL. This problem only becomes worse as the number of component types in a DRE system increases. In our UAV application for example, enhancing the UAV with new capabilities can increase the number of component types and inter-component interactions. If a problem arises, developers of DRE systems may need to revise the interface definitions until the types match, which is a tedious and error-prone process.

Solution → Semantically Compatible Component Interaction Definition.

PICML defines the *static semantics* of a system using a constraint language and enforces these semantics early in the development cycle, *i.e.*, at design-time. This type checking can help identify system configuration errors similar to how a compiler catches syntactic errors early in the programming cycle. Static semantics refer to the "well-formedness" rules of the language. The well-formedness rules of a traditional compiler are nearly always based on a language grammar defining valid syntax. By elevating the level of abstraction via MDE techniques, however, the corresponding well-formedness rules of DSMLs like PICML actually capture semantic information, such as constraints on composition of models, and constraints on allowed interactions.

There is a significant difference in the early detection of errors in the MDE paradigm compared with traditional object-oriented or procedural development using a conventional

```
let facets = self.connectedFCOs(invoke) in
  facets->forAll ( i : ProvidedRequestPort |
    let supertypes = i.refersTo().oclAsType(gme::Model).allParents(Set{}) in
     (supertypes->one (k: gme::FCO | k.name() = self.refersTo().name())
        or self.refersTo().name() = i.refersTo().name() ) )
```

Constraint Listing 1: A Receptacle should be connected to a matching Facet

programming language compiler. In PICML, OCL constraints are used to define the static semantics of the modeling language, thereby disallowing syntactically invalid systems to be built using PICML. For example, the constraint shown in Constraint Listing 1 is a PICML constraint, which checks that the type of a receptacle matches either the corresponding facet's type, or that the receptacle is a super type of the facet type. This is a good example of a constraint that ensures the type compatibility of components that are composed to form an assembly.

Constraints in PICML are not necessarily restricted to type conformance. Constraint Listing 2 is an existential constraint, *i.e.*, it checks for the presence of an implementation corresponding to each *instance* of a component used in an assembly. Each component type may have different alternate implementations offering different QoS behavior, and the correct implementation is chosen at deployment time. Section III.2.2.4 describes this process in greater detail. However, in order to select the correct component implementation at deployment time, it is critical that each instance of a component be associated with an implementation, and this constraint checks this invariant for every component instance in the model.

Another example of a constraint in PICML is the ability to restrict the flow of information in a particular direction, *i.e.*, top-down or bottom-up. Attributes of components in CCM can have default values assigned to them in the implementation. Depending on the context, attributes of components can also have values propagated to them from outside. This propagation is done using an "attribute mapping," which is a mechanism to propagate the value of an attribute of a higher-order element like an assembly, to one or more

```
let instances = self.modelParts(Component) in
  let monolithicImpls = project.allInstancesOf (MonolithicImplementation) in
    instances->forAll (x : Component |
      let myType = x.ComponentParentType() in
        monolithicImpls->exists ( impl : MonolithicImplementation |
          let interfaces = impl.connectedFCOs(Implements) in
            interfaces->size() = 1 and
              interfaces->exists (interface : Reference |
                interface.refersTo().name() = myType.name() ) ) )
```

Constraint Listing 2: Every Component should have a corresponding implementation

attributes of one of more components inside the assembly. Constraint Listing 3 restricts this propagation of initial assignment to flow in a strictly top-down fashion, *i.e.*, to give a behavior that matches the behavior of turning a top-level knob affecting the low-level knobs of a system.

```
let mappings = self.referenceParts (AttributeMapping) in
  let children = self.modelParts(ComponentAssembly) in
    mappings->forAll ( x : AttributeMapping |
    let delegates = x.connectedFCOs("dstAttributeMappingDelegate",
                                    AttributeMappingDelegate) in
      delegates->forAll ( y : FCO |
        let  delParent : Model = y.parent() in
          children->exists ( z : ComponentAssembly |
             delParent.name() = z.name() ) ) )
```

Constraint Listing 3: AttributeMappings can only be delegated from a high-level assembly to sub-assemblies, and not vice-versa

By using GME's constraint manager, PICML constraints can be (1) evaluated automatically (triggered by a specified modeling event such as attempting a connection between ports of two components) or on demand, (2) prioritized to control order of evaluation and severity of violation, and/or (3) applied globally or to one or more individual model elements

Application to the UAV example scenario. In the context of our UAV application, the components of a single stream can be modeled as a CCM assembly. PICML enables the

visual inspection of types of ports of components and the connection between compatible ports, including flagging an error when attempting a connection between incompatible ports. For example, PICML will flag attempts to connect incompatible `LocalResourceManager` receptacles with the facets of the `Qoskets` in a stream in the UAV scenario. By constraining the direction of flow of information, PICML also ensures that the global policies that are set by the `SystemResourceManager` are honored by the `LocalResourceManagers` of the individual streams. PICML also differentiates types of connections using visual cues, such as dotted lines and color, to quickly compare the structure of an assembly. By providing a visual environment coupled with rules defining valid constructs, PICML resolves many problems described in Section III.2.2.2 with ensuring consistent component interactions. By enforcing the constraints during creation of component models and interconnections — and by disallowing connections to be made between incompatible ports — PICML completely eliminates the manual effort required to perform these kinds of checks.

III.2.2.3 Generating Valid Deployment Descriptors

Component developers must not only ensure type compatibility between interconnected component types as part of an interface definition, but also ensure the same compatibility between instances of these component types in the XML descriptor files needed for deployment. This problem is of a larger scale than the one above, since the number of *component instances* typically dwarfs the number of *component types* in a large-scale DRE system. Moreover, a CCM assembly file written using XML is not well-suited to manual editing.

In addition to learning IDL, DRE system developers must also learn XML to compose component-based DRE systems. In our example UAV application, simply increasing the number of UAVs increases the number of component instances and component interconnections. The increase in component interconnections is typically not linear with respect to the increase in number of component instances. Any errors in this step are likely to go undetected until the deployment of the system at run-time.

Solution → Automatic Deployment Descriptor Generation

In addition to ensuring design-time integrity of systems built using OCL constraints, PICML also generates the complete set of deployment descriptors that are needed as input to the component deployment mechanisms. The descriptors generated by PICML conform to the descriptors defined by the standard OMG D&C specification [90]. Sidebar 2 shows an example of the types of descriptors that are generated by PICML, with a brief explanation of the purpose of each type of descriptor.

Sidebar 2: Generating Deployment Metadata

PICML generates the following types of deployment descriptors based on the OMG D&C specification:

- **Component Interface Descriptor (.ccd)** — Describes the interfaces — ports, attributes of a single component.
- **Implementation Artifact Descriptor (.iad)** — Describes the implementation artifacts (DLLs, executables etc.) of a single component.
- **Component Implementation Descriptor (.cid)** — Describes a specific implementation of a component interface; also contains component inter-connection information.
- **Component Package Descriptor (.cpd)** — Describes multiple alternative implementations (for different OSes) of a single component.
- **Package Configuration Descriptor (.pcd)** — Describes a component package configured for a particular requirement.
- **Component Deployment Plan (.cdp)** — Plan which guides the run-time deployment.
- **Component Domain Descriptor (.cdd)** — Describes the deployment target *e.g.*, nodes, networks on which the components are to be deployed.

Since the rules determining valid assemblies are encoded into PICML via its meta-model, and enforced using constraints, PICML ensures that the generated XML describes a syntactically valid system. Generation of XML is done in a programmatic fashion by

writing a `Visitor` class that uses the Visitor pattern [40] to traverse the elements of the model and generate XML. The generated XML descriptors also ensure that the names associated with instances are unique, so that individual component instances can be identified unambiguously at run-time.

Application to the UAV example scenario. In the context of the UAV application, the automated generation of deployment descriptors using PICML not only removes the burden of knowing XML from DRE system developers, it also ensures that the generated files are valid. XML is a linear text language; links and containment are hard to manage using plain XML. PICML provides a visual means to deal with these complexities. Adding (or removing) components is as easy as dragging and dropping (or deleting) an element, making the necessary connections, and regenerating the descriptors, instead of hand-modifying the existing XML files as would be done without such tool support. This automation resolves many problems mentioned in Section III.2.2.3, where the XML files were hand-written and modified manually in case of errors with the initial attempts.

For example, it is trivial to make the $\sim$100 connections in a graphical fashion using PICML, as opposed to hand-writing the XML. All the connections between components for the UAV application were made in a few hours, and the XML was then generated instantaneously, *i.e.*, at the click of a button. In contrast, it required several days to write the same XML descriptors manually. PICML also has the added advantage of ensuring that the generated XML files are syntactically valid, which is a task that is very tedious and error-prone to perform manually.

III.2.2.4 Associating Components with the Deployment Target

In component-based systems there is often a disconnect between software implementation related activities and the actual target system since (1) the software artifacts and the physical system are developed independently and (2) there is no way to associate these

two entities using standard component middleware features. This disconnect typically results in failures at run-time due to the target environment lacking the capabilities to support the deployed component's requirements. These mismatches can also often be a source of missed optimization opportunities since knowledge of the target platform can help (1) optimize component implementations, (2) select appropriate component implementations to be deployed and (3) customize the middleware for the appropriate target environment. In our UAV application, components that reside on a single UAV can use collocation facilities provided by ORBs to eliminate unnecessary (de)marshaling. Without the ability to associate components with targets, errors due to incompatible component connections and incorrect XML descriptors are likely to show up only during actual deployment of the system.

Solution → Deployment Planning. In order to satisfy multiple QoS requirements, DRE systems are often deployed in heterogeneous execution environments. To support such environments, component middleware strives to be largely independent of the specific target environment in which application components will be deployed. The goal is to satisfy the functional and systemic requirements of DRE systems by making appropriate deployment decisions that account for key properties of the target environment, and retain flexibility by not committing prematurely to physical resources.

To support these needs, PICML can be used to specify the target environment where the DRE system will be deployed, which includes defining: (1) **Nodes**, where the individual components and component packages are loaded and used to instantiate those components, (2) **Interconnects** among nodes, to which inter-component software connections are mapped, to allow the instantiated components to communicate, and (3) **Bridges** among interconnects, where interconnects provide a direct connection between nodes and bridges to provide routing capability between interconnects. Nodes, interconnects, and bridges collectively represent the target environment.

Once the target environment is specified via PICML, allocation of component instances onto nodes of the target environment can be performed. This activity is referred to as

component placement, where systemic requirements of the components are matched with capabilities of the target environment and suitable allocation decisions are made. Allocation can either be: (1) **Static**, where the domain experts know the functional and QoS requirement of each of the components, as well as knowledge about the nodes of the target environment. In such a case, the job of the allocation is to create a deployment plan comprising the component-node mapping specified by the domain expert, or (2) **Dynamic**, where the domain expert specifies the constraints on allocation of resources at each node of the target environment, and the job of the allocation is to choose a suitable component-node mapping that meets both the functional and QoS requirement of each of the components, as well as the constraints on the allocation of resources.

PICML currently provides facilities for specifying static allocation of components. In order to compute the static deployment plan for the components and the component assemblies, PICML makes use of the following inputs specified in the model:

- **Component Implementation Capabilities.** Component middleware provide multiple implementations with different QoS characteristics for the same component interface. PICML provides mechanisms to annotate component implementations with capabilities at the modeling level.

- **Component Middleware Target Capabilities.** DRE systems are deployed in heterogeneous target environments, each of them exposing various capabilities like processing power and memory for the applications to be operated. PICML allows such capabilities to be specified while modeling the target environment in which the component middleware is going to be deployed.

- **Component Implementation Selection Requirements.** Different uses of the same component might need to perform under differing QoS requirements. PICML allows the system integrators to specify component implementation selection requirements with each use of a component in an assembly.

As shown in Figure III.2, domain experts can visually map the components with the respective target nodes, as well as provide additional hints, such as whether the components need to be process-collocated or host-collocated, provided two components are deployed in the same target node. PICML generates a deployment plan which uses the component

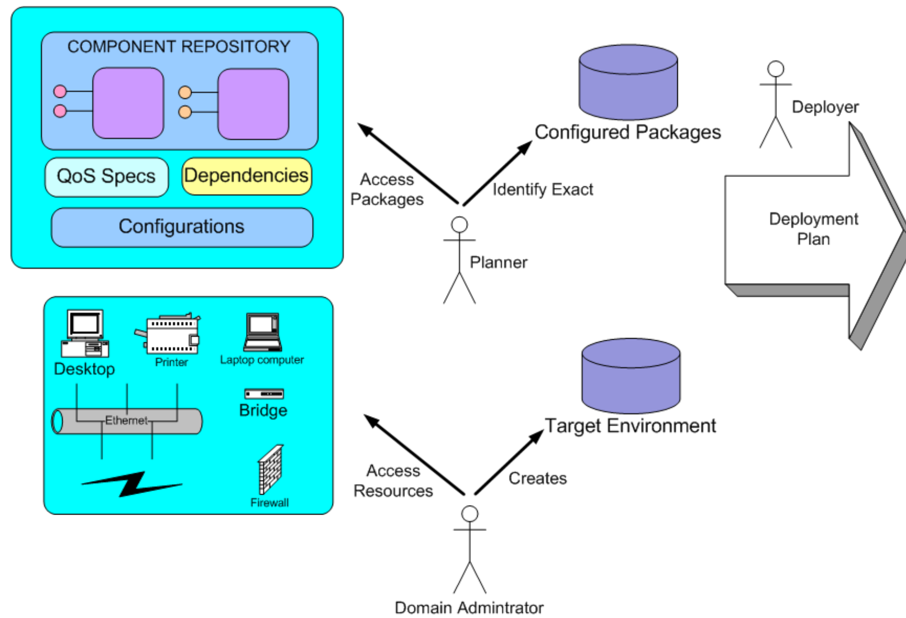


Figure III.2: Component Deployment Planning

implementation capabilities, the target capabilities and the component implementation selection requirements, and generates the mapping of components to nodes. This deployment plan is then used by the CIAO run-time deployment engine to perform the actual deployment of components to nodes.

Application to the UAV example scenario. In the context of the UAV example, PICML can be used to specify the mapping between the different `Qosket` components and the target environment, *i.e.*, the UAVs, in the path from each UAV to the `Receiver` component at the control center. By modeling the target environment in the UAV example using PICML, therefore, the problem with a disconnect between components and the deployment target described in Section III.2.2.4 can be resolved. In the case there are multiple possible

component-node mappings, PICML can be used to experiment with different combinations since it generates descriptors automatically. PICML thus completely eliminates the manual effort involved in creating the deployment plan when there is a need to test different deployment scenarios.

III.2.2.5 Automating Propagation of Changes Throughout a DRE System

Making changes to an existing component interface definition can be painful since it may involve retracing all the steps of the initial development. It also does not allow any automatic propagation of changes made in a base component type to other portions of the existing infrastructure, such as the component instances defined in the descriptors. Moreover, it is hard to test parts of the system incrementally, since it requires hand-editing of XML descriptors to remove or add components, thereby potentially introducing more problems. The validity of such changes can be ascertained only during deployment, which increases the time and effort required for the testing process. In our component-based UAV application, for example, changes to the basic composition of a single image stream are followed by laborious changes to each individual stream, impeding the benefits of reuse commonly associated with component-based development.

Solution → Hierarchical Composition.

In a complex DRE system with thousands of components, visualization becomes an issue because of the practical limitations of displays, and the limitations of human cognition. Without some form of support for hierarchical composition, observing and understanding system representations in a visual medium does not scale. To increase scalability, PICML defines a *hierarchy* construct, which enables the abstraction of certain details of a system into a hierarchical organization, such that developers can view their system at multiple levels of detail depending upon their needs.

The support for hierarchical composition in PICML not only allows DRE system developers to visualize their systems, but also allows them to compose systems from a set

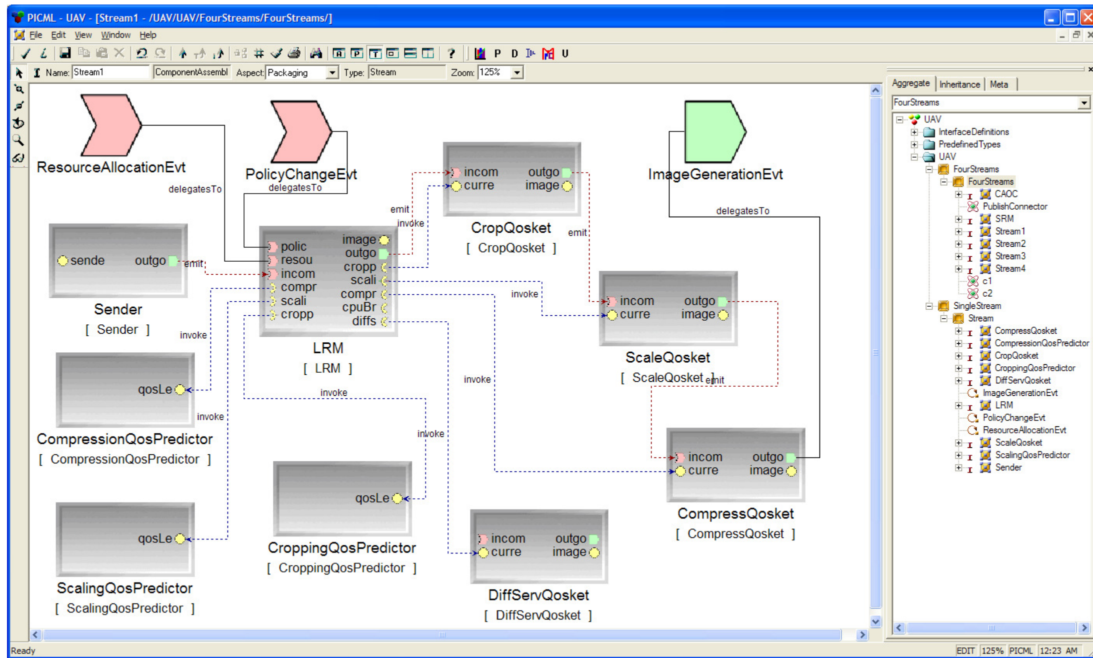


Figure III.3: Single Image Stream Assembly

of smaller subsystems. This feature supports unlimited levels of hierarchy (constrained only by the physical memory of the system used to build models) and promotes the reuse of component assemblies. PICML enables the development of repositories of predefined components and subsystems.

The hierarchical composition capabilities provided by PICML are only a *logical* abstraction, *i.e.*, deployment plans generated from PICML (described in III.2.2.4) flatten out the hierarchy to connect the two destination ports directly (which if not done will introduce additional overhead in the communication paths between the two connected ports), thereby ensuring that at run-time there is no extra overhead that can be attributed to this abstraction. This feature extends the basic hierarchy feature in GME, which allows a user to double-click to view the contents of container objects called "models."

Application to the UAV example scenario. In the UAV example, the hierarchy abstraction in PICML allows the composition of components into a single stream assembly as shown

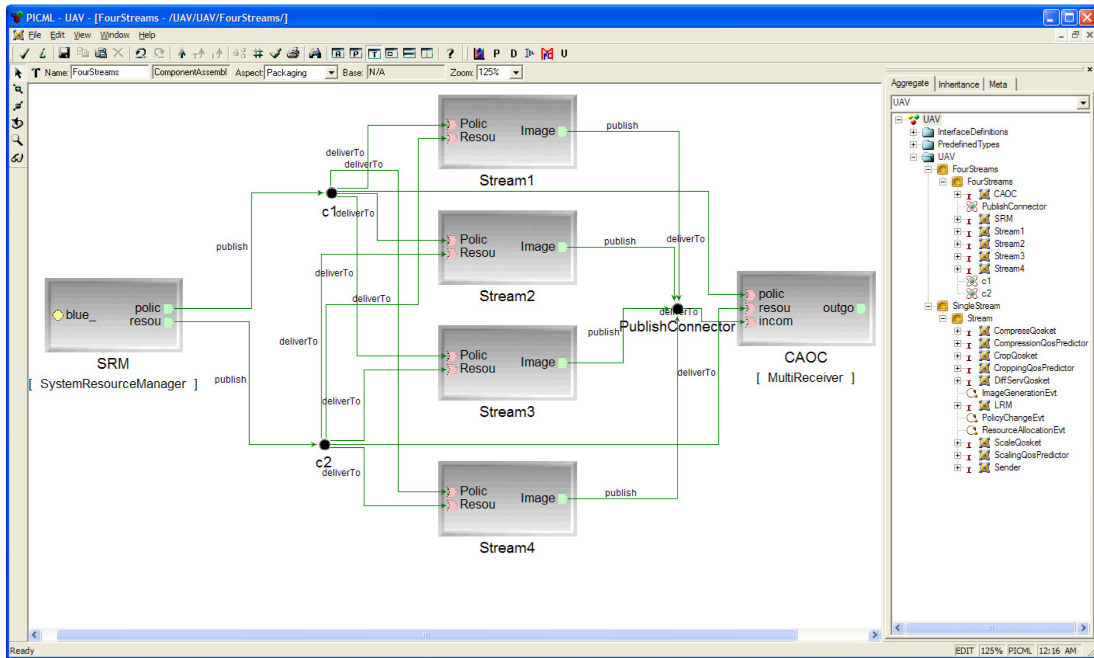


Figure III.4: UAV Application Assembly Scenario

in Figure III.3, as well as the composition of multiple such assemblies into a top-level scenario assembly as shown in Figure III.4.

As a result, large portions of the UAV application system can be built using reusable component assemblies. In turn, this increased reuse allows for automatic propagation of changes made to a subsystem to all portions of the system where this subsystem is used, resolving many problems mentioned in Section III.2.2.5. PICML helps prevent mismatches and removes duplication of subsystems.

Hierarchical assemblies in PICML also help reduce the effort involved in modeling of component assemblies by a factor of $n:1$, since n usages of a basic assembly can be replaced with n instances of the same assembly, as well as providing for automatic generation of descriptors corresponding to the n instances. This technique was used to model a single stream of image from a UAV, and this single assembly was used to instantiate all the four streams of data, as shown in Figure III.4.

III.3 Summary

Although component middleware represents an advance over previous generations of middleware technologies, its additional complexities threaten to negate many of its benefits without proper tool support. To address this problem, we describe the capabilities of the Platform-Independent Component Modeling Language (PICML) in this chapter. PICML is a domain-specific modeling language (DSML) that simplifies and automates many activities associated with developing and deploying component-based DRE systems. In particular, PICML provides a graphical DSML-based approach to define component interface definitions, specify component interactions, generate deployment descriptors, define elements of the target environment, associate components with these elements, and compose complex DRE systems from such basic systems in a hierarchical fashion.

To showcase how PICML helps resolve the complexities of QoS-enabled component middleware, we applied it to model key aspects of an unmanned air vehicle (UAV) application that is representative of emergency response systems. Using this application as a case study, we showed how PICML can support design-time activities, such as specifying component functionality, interactions with other components, and the assembly and packaging of components, and deployment-time activities, such as specification of target environment, and automatic deployment plan generation.

CHAPTER IV

TECHNIQUES FOR OPTIMIZING COMPONENT-BASED SYSTEMS

Component middleware technologies, such as the CORBA Component Model [89] and Enterprise Java Beans [27], have raised the level of abstraction used to develop distributed, real-time and embedded (DRE) systems, such as avionics mission computing [113] and shipboard computing systems [66]. In addition to elevating the level of abstraction, the component middleware also promotes the decomposition of monolithic systems into a number of sub-systems *i.e.*, collections of inter-connected components (called a *component assembly*) which is composed of individual (indivisible) components (called *monolithic component*). An assembly is a set of monolithic components inter-connected using the ports of the components in a particular fashion.

Assemblies of components as defined by standard component middleware like CCM are *virtual*, *i.e.*, the individual components that form the assembly can be spread across multiple machines of the target domain. Monolithic components of virtual assemblies are only mapped onto the target nodes of the domain as part of the deployment process. In contrast to a *virtual* assembly, a *physical* assembly is defined as the set of components created from the monolithic components that are deployed onto a single process of a physical node, as shown in Figure IV.1. A physical assembly is itself a full-fledged component, *i.e.*, it has a component interface as well as an implementation. The implementation of the physical assembly, however, simply delegates to the original implementations of the monolithic components from which the physical assembly is created.

Since component middleware allows developers to expose reusable functionality using the Extension Interface design pattern [118], it can provide better reuse of components across product lines [122]. Component middleware also decouples application logic from

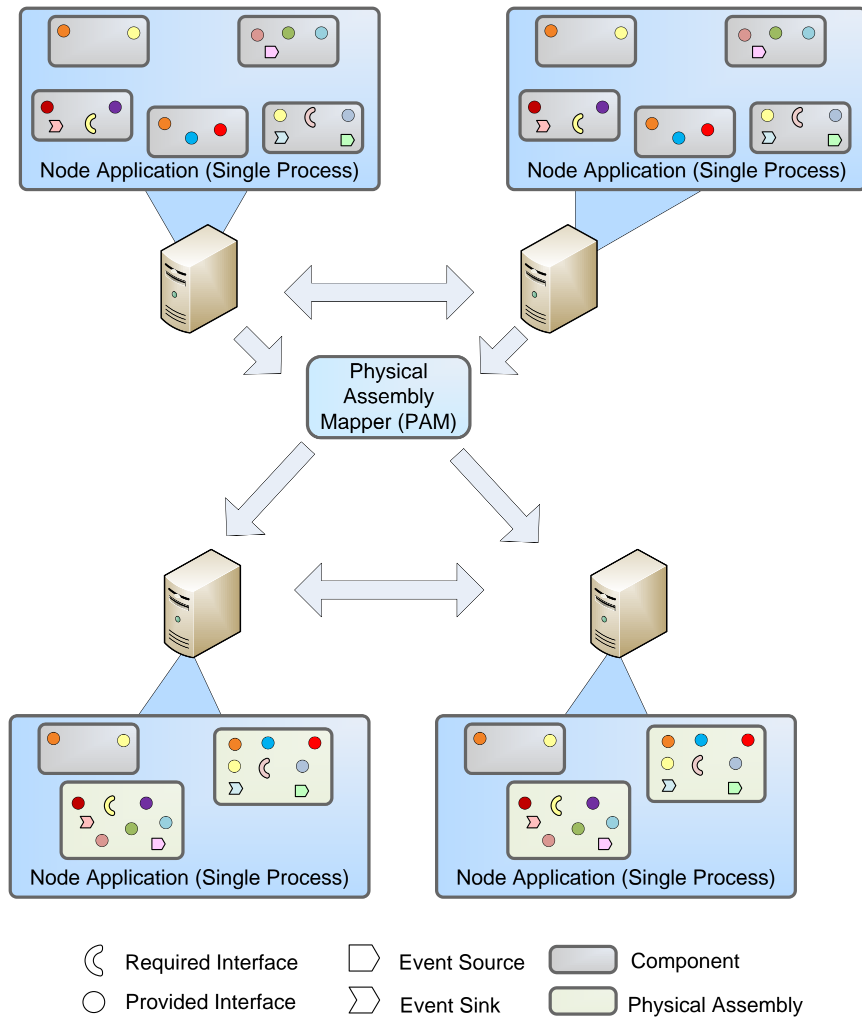


Figure IV.1: Physical Assembly

QoS configuration in DRE systems. For example, traditional DRE system QoS provisioning concerns, such as managing OS resources like threads, thread priorities, socket connections, multi-homed network interfaces and mutual exclusion, can be configured using high-level entities called *containers* and *application servers*.

Component middleware technologies also help decouple the "design and development" phases of DRE systems from the "deployment and management" phases. For example, various techniques [6, 35, 104, 110] extract deployment and configuration information that was previously coupled with the implementation into *declarative* metadata known as "deployment descriptors." As a result, DRE system deployment flexibility is increased because

it is easier to deploy the same version of a single component in a variety of deployment contexts by configuring the component in a context-dependent fashion.

Although component middleware provides the benefits described above, a number of challenges may restrict its use for a large class of DRE systems with stringent footprint and latency requirements. For example, while functional decomposition of DRE systems into component assemblies and monolithic components helps promote reuse across entire product lines, it can also increase the number of components in the system. These many components, in turn, can significantly increase the memory footprint of component-based DRE systems.

Component middleware technologies are also more complex in terms of configuration and provisioning for the QoS requirements of DRE systems. Not only does the correct functioning of DRE systems depend on correct configuration of component middleware features, but misconfiguration of QoS can drastically reduce DRE system performance. The presence of more components also increases complexity during QoS configuration and deployment of DRE systems.

To address the challenges of large-scale component-based DRE systems described above, we have developed model-driven optimization techniques that help reduce the overhead associated with DRE systems containing a large number of components. Our optimization techniques focus on reducing the footprint overhead as well as the QoS mis-configurations in component middleware by optimizing the assembly of components at deployment-time. By combining together multiple components as well as QoS policies (using a technique known as *fusion*), our optimizations reduces the number of components/QoS policies required to deploy a system.

The fusion of multiple components creates a physical assembly. A key enabler in the creation of physical assemblies is the presence of (a) application structure information, *i.e.*, connections between components, (b) application QoS configuration information, *i.e.* QoS

configuration associated with each component, and (c) application deployment information, *i.e.*, the mapping of components onto physical nodes (and processes within nodes). Our optimization techniques obtain such information from the models of application built using DSMLs.

Although models are not a pre-requisite for the creation of physical assemblies, it is hard to both identify as well as optimize component implementations in the absence of a well-defined description of the systems that can be processed by tools automatically. Performing such optimizations of components is hard since the usage of any single component tends to span across multiple compositional hierarchies, *i.e.*, a single component could be connected to different sets of components in different assemblies, in any complex system.

Since components tend to get reused across an entire product line, an optimization that is applicable in one context may not be applicable in another context. Thus, it is not possible to perform these optimizations in isolation, but rather one should perform them based on the requirements of every unique deployment. Finally, performing these optimizations manually by hand becomes infeasible due to the presence of a large number of components as well as due to changes caused by system evolution. The novelty of our approach stems from both identifying and applying these optimizations in an opportunistic and automatic fashion from models of applications, thereby eliminating the difficulties associated with applying these optimizations in a manual fashion.

We have applied these optimization techniques in a model-driven tool called the *Physical Assembly Mapper* (PAM) that optimizes component-based DRE systems. PAM is built using the Generic Modeling Environment (GME) [67], a meta-programmable environment for creating DSMLs. PAM utilizes both the connectivity information between components modeled and the QoS policies to create physical assemblies.

By operating at the high-level abstraction of models, PAM allows optimizing virtual assemblies across two dimensions—*footprint* and *performance*—and at multiple levels—*local* (deployment plan-specific) and *global* (application-wide). A particular useful feature

of the PAM optimizer is that it operates at deployment-time. Changes are not required to the functional decomposition or structure of component-based DRE systems.

IV.1 Challenges in Large-scale Component-based DRE systems

This section describes key features of component middleware programming models and describes the cost of these features along two dimensions—memory footprint and invocation latency—for DRE systems. To make our discussion concrete, we use the CORBA Component Model (CCM) as an example of component model for our discussion. The sources of overhead, however, are generally applicable to any layered component middleware, such as Enterprise Java Beans (EJB), Boeing's PRiSM [124] and OpenCOM [23].

Although the features of CCM described in Section I.2 provide a number of benefits they can also consume excessive time/- resource overhead in large-scale DRE systems. In particular, we describe key sources of memory footprint and latency overhead.

IV.1.1 Key Sources of Memory Footprint Overhead

The contribution to the memory footprint of a component DRE system can be classified into two categories: *static* and *dynamic*. Increase in the static footprint results from code generated to integrate the implementation of a component with the middleware run-time environment; code generation is specific to each unique component type in the system. Dynamic footprint increases are due to the creation of run-time infrastructural elements like component homes and component context on a per-component basis. We discuss both types of memory footprint overhead in this sub-section.

Static footprint. For every component type in a DRE system, the CCM platform mapping requires generation of code for various infrastructure elements, including:

- **Component context**. A component context class is generated corresponding to each component interface to allow each component to be reused in multiple execution contexts.

- **Component base interfaces**. Each component interface derives from a number of base interfaces, *e.g.* `SessionComponent` and `EntityComponent` in CCM, which classify the category of a particular component.

- **Component home**. A component home is generated corresponding to each component interface. Each component home provides not only factory operations that allow customization of creating components but also provides finder operations that clients use to locate a component managed by a component home.

- **Navigation operations**. Each component also contains a number of pre-defined navigation operations. The navigation operations of a component interface allow clients of a component to query and obtain references to the ports of a component in a standardized fashion.

In addition to the interfaces and operations describe above, each component implementation is typically split into multiple shared libraries. For example, component implementations are often split into three shared libraries: *stub*, *servant*, and *executor*. The stub library contains the automatically generated client-side proxy code necessary for each component type to connect to other component types, the servant contains automatically generated code that registers a component with an Object Request Broker (ORB), and the executor contains the business logic of a component written by application developers.

The drawbacks of designing DRE systems using multiple shared libraries are well-known [32] including increased code size, increase in the number of dependencies between shared libraries and number of relocations at load time, all of which result in increased dynamic memory footprint. Developers are thus forced to make a choice with respect to the granularity of the component functionality as well as the component implementations. The design trade-off is between (1) a single monolithic shared library, which can increase the footprint of components that only need to connect to it, compared to (2) a number of shared libraries, which can increase the overall footprint and time taken to load the libraries

into memory. Section IV.2.1 describes how our optimization techniques defer the time that this design trade-off has to be made to deployment-time rather than design/development-time.

The overhead due to the static footprint increases with the number of component types. This overhead, which can be as high as 45%, becomes apparent in the presence of a large number of types or in resource constrained environments, which are common in DRE systems.

Dynamic Footprint. The code generated per component interface that allows components to be hosted by the containers not only adds to the static footprint, it also creates a number of auxiliary middleware infrastructural elements corresponding to each component instance at runtime, including:

- **Component home**. Since a component home can manage only one type of component, the CCM runtime infrastructure creates a separate component home instance for every component type loaded into a system. This component home is then used to create multiple component instances. Naïve implementations could also create a component home instance per component instance. CCM allows clients to create components dynamically by obtaining a reference to its component home. In many classes of DRE systems, these sophisticated features of component homes are seldom used and impose additional time/space overhead corresponding to each component instance created at run-time.

- **Component context**. The runtime infrastructure creates a component context corresponding to each component instance that is deployed. The component context contributes to the increase in the dynamic footprint corresponding to the increase in the number of component instances.

- **Component servant**. Each component instance must also be registered with the underlying middleware infrastructure to communicate with other components. A

component servant is created at run-time corresponding to each instance of a component and allows it to be registered with the middleware. Although inevitable, each component servant created contributes to the increase in the dynamic footprint of the system.

Each component instance consumes a certain amount of memory in the runtime environment. In the presence of a large number of components, it is imperative to reduce the number of component instances/types to reduce the memory consumption of the system as a whole. In order to reduce the dynamic memory footprint of the system due to auxiliary middleware infrastructure elements, the designers are forced to make a decision with respect to the number of components as well as the granularity of the components during the creation of assemblies, *i.e.*, design/development-time. It is non-trivial to keep track of any redundant component instances during component assembly creation, since each such component instance can be spread across multiple assemblies, *i.e.*, sub-systems. Forcing the designer to pay attention to issues like number of component instances created and redundancy in component instances, during component assembly design distracts the designer from the high-level issues like functionality of the assembly. The design trade-off here is between (1) fine-grained decomposition of the system into a number of component types/instances, which can increase memory footprint, compared to (2) monolithic architectures that are strongly coupled, brittle, and discourage reuse, but which reduces the memory consumption of the auxiliary middleware infrastructure elements (by creating few of them). Section IV.2.1 describes how the creation of physical assemblies reduces the number of components in the system without requiring the use of monolithic architectures at design/development-time.

Although static overhead of a component increases its lower limit of the memory requirement, this type of overhead does not grow as the number of components increases on a single node. Dynamic overhead, in contrast, increases linearly as the number of components grows. In a large-scale scenario with thousands of components, reducing the

dynamic overhead is essential to reduce memory footprint requirements of an integrated system. Section IV.2.1 describes how our optimization techniques reduce the total number of components in the system.

IV.1.2 Key Sources of Latency Overhead

As shown in Figure I.1, containers are central to the QoS policy configuration and enforcement in CCM. In particular, containers are responsible for configuring policies related to middleware resources, such as threads/threadpools, thread priority assignment, mapping priorities between different operating systems, and propagation of priorities between clients and servers. In a typical CCM configuration, containers are homogeneous, *i.e.*, containers host components with similar QoS policies. The runtime infrastructure can create separate containers for each type of QoS policy.

Invocations between components in the same container usually are collocated, *i.e.*, they do not traverse the local loopback and incur no (de)marshaling overhead. In the presence of Real-time CORBA [95] features (such as priority propagation, threadpools and threadpools with lanes), however, the ORB performs a series of checks including matching priority, matching threadpools, matching lanes within a threadpool, to determine if the invocation can be done in a collocated fashion. The difference in invocation latency between the case when it is collocated and when it is done over the local loopback is order of magnitudes slower [143].

Each container has exclusive access to middleware resources, such as threadpools, that it manages. Components that are hosted on different containers will often not take advantage of collocation optimizations done by the middleware, and hence will exhibit increased invocation latencies.

Components in a DRE system typically go through schedulability analysis [1], verification [74], and various kinds of resource planning [26] before they are deployed. The outcome of these tasks is usually a schedule and the assignment of priorities and other QoS

policies to the various components. A direct translation of the results of such analysis activities into a middleware configuration will result in the creation of a multitude of QoS policies.

For example, RT-CORBA defines QoS policies such as priorities assigned to each component (*PriorityModelPolicy*, mapping of the CORBA priority to native OS priority (*PriorityMapping*), total number of threads as well as the partitioning of threads with same priorities (*ThreadPoolPolicy*) associated with each component, whether connections between components are shared (*PrivateConnectionPolicy*), the granularity of shared connections including the priority(-ies) associated with each such shared connection (*PriorityBandedConnectionPolicy*), properties of the protocol used for communication between a source component (*ClientProtocolPolicy*) and a destination component (*ServerProtocolPolicy*). As a result, as many containers will be created as there are unique QoS policies.

Even within a single QoS policy, there may be multiple instances of middleware resources, such as thread pools. Such QoS configurations have the unfortunate effect that despite careful schedulability analysis and resource allocation, components end up with sub-optimal invocation latency. To reduce the negative effects due to QoS configuration, the designer is forced to keep track of the number of QoS policies assigned to components across multiple assemblies, and whether they need to be unique or could potentially be combined during design/development time.

Keeping track of such issues distracts the designer from the more important issues of the exact QoS configuration of each individual component. The trade-off here is between (1) ease of QoS configuration, *e.g.*, by assigning desired QoS policies to components based on functional decomposition, *i.e.*, a component assembly alone leading to sub-optimal performance, compared to (2) increased complexity in analysis and QoS configuration due to exposing platform-specific details into the high-level functional analysis, but with possibly better performance.

It is important to note that extracting commonality among the different QoS policies and

allocating middleware resources in an optimal fashion at runtime can be expensive in terms of the time taken to deploy an application and the memory required to manage the different QoS policies that are in effect across all containers in a component server. In a large-scale, component-based DRE system scenario, it is crucial to eliminate the invocation latency overhead caused due to sub-optimal QoS policy configuration. Section IV.2.1 describes how our optimization techniques help ensure optimal QoS configuration by merging QoS policies.

IV.2 Deployment-time Optimization Techniques

As described in Section IV.1.1, a key source of footprint overhead is the number of peripheral infrastructure elements, such as component home and component context, created for each monolithic component. An approach that reduces the number of components deployed should reduce the number of peripheral infrastructure elements, thereby reducing the static and dynamic footprint of the component-based DRE systems. Several possible techniques are available to reduce the number of components in a DRE system, with each technique have its own trade-offs. These techniques can be categorized along the following time spectrum, *i.e.*, the time that such optimization techniques are applied:

- **Design/development-time techniques.** One of the easiest techniques is to design the system to not use many components. Using this technique, the design of the system is changed so that each component performs possibly more than one functionality, and the goal is to reduce the number of components in the whole system. This approach, however, can significantly reduce component reuse, thereby limiting a key benefit of component middleware. As a result, this approach is essentially equivalent to traditional DRE system development architectures and lies at one end of the optimization time spectrum.

- **Run-time techniques.** Another way to reduce the total number of components is to

load monolithic components into memory when required and unload them when not in use. Although this technique is viable for certain classes of systems, DRE systems often cannot tolerate the jitter of dynamic loading. This approach is the opposite of the static approach and lies at the other end of the optimization time spectrum.

- **Deployment-time techniques.** One benefit of component middleware is the clear separation of the design and implementation phases from the deployment and configuration phases. Treating deployment as a separate phase facilitates a class of optimizations that neither require modifications to the design of the system (static approach) nor come with the associated overhead of loading/unloading components at runtime (dynamic), but instead are performed in accordance with the requirements of a specific DRE system deployment.

The approach presented in this chapter uses deployment-time optimization techniques. This section first describes the model-driven optimization techniques that help reduce the time and overhead in large-scale component-based DRE systems. It then presents the structure and functionality of the *Physical Assembly Mapper* (PAM), which is a tool that automates deployment-time optimization techniques in the context of the CORBA Component Model (CCM).

IV.2.1 Deployment-time Optimization Algorithms

The central theme of our component assembly optimizations is the notion of "fusion." By fusion, we refer to the merging of multiple elements into a semantically equivalent element. One of the key differences between the various optimization techniques described in this section, is the type of elements fused, the scope at which such fusion is performed, as well as the rules governing which elements are fused.

The optimization technique described in Section IV.2.1.4 fuses multiple components into a single physical assembly at the level of a single deployment plan; the technique

described in Section IV.2.1.5 also fuses components into a single physical assembly but the scope of such fusion spans an entire application; the technique described in Section IV.2.1.6 fuses multiple QoS policies into a single aggregate QoS policy.

IV.2.1.1 Assumptions and Challenges in Component/QoS Policy Fusion

A physical assembly is defined as the set of components created from the monolithic components that are deployed onto a single process of a physical node. Our optimization techniques creates one or more physical assemblies by *fusing* monolithic components deployed into the same process on each node of the target domain. To ensure that our component fusion technique for creating physical assemblies does not degenerate to the static technique described in Section IV.2, our approach operates under the following assumptions:

1. Physical assembly creation should not require changes to the existing implementations of monolithic components.
2. Physical assembly creation should not impact existing clients of fused components.

At the core of the component fusion technique is the capability to merge multiple components into a single physical assembly. As described in Section I.2, components interact with the external world using ports. Fusing multiple components into a single component requires merging the ports of all the individual components. There are, however, the following challenges in fusing multiple components into a single physical assembly in a DRE system:

1. **Ports of a component are identified using their names.** Each component interface defines a namespace; each kind of port (*e.g.*, facets, receptacles etc.) defines its own unique namespace within a component. Port names are also used to locate the services provided by each component and affect the middleware glue code generated for each component. Since ports are the externally visible points of interaction, port

names of a component must be unique within the corresponding port kind namespace. Although this holds true for each individual component, it need not be true when merging multiple components into a single component. Section IV.2.2.2 describes how we address this challenge in PAM.

2. **Each component relies on being supplied a component context.** This context is needed to connect the component with the services of other components that it depends upon. If multiple components are fused together, each component in the fused physical assembly must be provided with a context that is compatible with each monolithic component's context. Section IV.2.2.3 describes how we address this challenge in PAM.

3. **Each component maintains its externally visible state through its component attributes.** When fusing multiple components together, it is necessary to ensure that the states of the individual components are maintained separately. It is also necessary to allow modification to such state from external clients. Section IV.2.2.2 describes how we address this challenge in PAM.

4. **Each component must be identified uniquely.** To obtain the services of a component through its ports, external clients must be able to locate the component via directory services, such as the CORBA Naming Service, LDAP servers, and Active Directories. If multiple components are fused into a single component, the external clients should still be able to lookup the individual components using their original names. Section IV.2.2.2 describes how we address this challenge in PAM.

5. **Components can be associated with QoS policies.** The QoS policies of different components might be different and sometimes incompatible with each other. For example, in the RT-CORBA programming model, the *PriorityModel* policy determines the priority at which a component will execute as described in Section IV.1.2. The two possible values for this policy: CLIENT_PROPAGATED and SERVER_DECLARED

are incompatible, and thus two components that have these policies cannot be fused into one. Any fusion of multiple components should take into account the change in execution semantics caused by the fusion, *i.e.*, compatible QoS policies is a prerequisite to any fusion. Section IV.2.2.2 describes how we address this challenge in PAM.

IV.2.1.2 Common Characteristics of Fusion Algorithms

Our fusion algorithms perform a series of checks to evaluate "mergeability," *i.e.*, whether multiple elements such as components and QoS policies can be merged into a single element. The property of mergeability of two elements is non-transitive. Every pair of elements must be examined to determine if they can be merged together.

If n is the number of candidate elements for each algorithm, *e.g.*, set of components deployed in a single process, set of QoS policies associated with components of a single process, k is the number of elements that result from merging components together, then the number of elements will be reduced by $\frac{n-k}{n}$. Of the elements that can be merged into a single element, our goal is to find the largest set of elements because the larger the number of elements that we can merge, the greater the reduction in the number of elements. The best case is when $k = 1$, *i.e.*, the savings will be $\frac{n-1}{n}$.

Given an undirected graph $G = (V, E)$, where V is the set of candidate elements, and E is the set of edges such that if two elements are connected then they can be merged together, the problem of finding the largest set of elements that can be merged together is equivalent to the problem of finding a maximum clique in the undirected graph G. The maximum clique determination problem is well-known to be NP-complete [56] and a survey of the maximum clique problem appears in [12].

One can find a maximum clique by enumerating all the maximal cliques and choosing the largest. An efficient algorithm for enumerating the maximal cliques is by Bron and

Kerbosch [15], which is *Algorithm 457* in the ACM collection. The worst-case time complexity for enumerating all maximal cliques has recently been proven [137] to be $O(3^{n/3})$, where n is the number of vertices in the graph. Our component fusion algorithm, therefore, does not calculate maximum clique by enumerating all maximal cliques and choosing the largest.

For our first implementation, we chose to trade-off the time savings from calculating just maximal cliques (as opposed to a maximum clique) and using these to creating physical assemblies, over the benefits of the footprint savings from creating physical assemblies out of maximum cliques. We, therefore, use a variation of the algorithm by Bron and Kerbosch due to Koch [63] to calculate the maximal cliques. This algorithm has the desirable property that it enumerates the larger maximal cliques first. In our preliminary testing of the algorithm with some representative DRE systems, as shown in Section IV.3, we found that the maximal cliques chosen by our current algorithm tend to also be maximum size cliques. This, however, does not hold true for all systems. We intend to make the choice between maximal and maximum clique as an option to our tool that implements the algorithms.

IV.2.1.3 Terminology

We now define some terms used in our algorithms: a *node* is the physical machine on which one or more components are deployed. A *domain* is the target environment composed of independent nodes and their inter-connections. A *collocation group* is defined as the set of components that are deployed in a single process of a target node. Each collocation group corresponds to a single OS process and is always associated with one target node.

A *deployment plan* is a mapping of a configured system into a target domain; it includes mapping of monolithic components to collocation groups, list of monolithic component implementations, list of implementation artifacts corresponding to the component implementation, connectivity information between the different components as well as configuration

information such as QoS policies associated with component instances. A deployment plan serves as the blueprint to be used by the middleware to deploy an application. The algorithms use several auxiliary functions that are briefly described below:

- $components(cg)$ Returns the set of components that belong to the collocation group cg.
- $types(I)$ Returns the set of types corresponding to the component instances in I.
- $collocationgroups(P)$ Returns the set of collocation groups that are defined in the deployment plan P.
- $nodes(P)$ Returns the set of nodes that are defined in the deployment plan P.
- $physicalassembly(C)$ Returns the physical assembly created by merging the components in set C. The physical assembly returned is also a component, *i.e.*, it has its own interface definition, ports and QoS policies.
- $maximalclique(G)$ Returns a maximal clique from G. This version returns the larger maximal cliques first.
- $qospolicies(c)$ Returns the set of QoS policies associated with component c.
- $facets(c)$ Returns the set of facets defined in component c.
- $receptacles(c)$ Returns the set of receptacles defined in component c.
- $publishers(c)$ Returns the set of publishers, *i.e.*, event sources defined in component c.
- $consumers(c)$ Returns the set of consumers, *i.e.*, event sinks defined in component c.

IV.2.1.4 Local Component Fusion Algorithm

We developed two versions of the component fusion algorithm, both of which operate under the assumption that all high-level deployment planning (*e.g.*, resource allocation) has been completed and the set of associations of components to nodes is finalized. The two algorithms differ in the scope at which they are applied. Algorithm 1 is called *Local Component Fusion*, where "local" refers to the fact that this version of the algorithm operates at the level of a single deployment plan. Algorithm 6 is called *Global Component Fusion*, where "global" refers to the fact that this algorithm operates at the level of an entire application.

Algorithm 1: Local Component Fusion

Input: **DeploymentPlan** IP
Result: **DeploymentPlan** OP
begin
  CollocationGroup cg;
  Component c; **set of Component** I;
  ComponentType t; **set of ComponentType** T;
  set of set of Component K;
  foreach $cg \in collocationgroups(IP)$ **do**
    $I \leftarrow \{c \mid c \in components(cg)\}$
    $T \leftarrow \{t \mid t \in types(I)\}$
    $K \leftarrow K \cup$ CreatePhysicalAssemblies (T, I)
  end
  $OP \leftarrow$ UpdateDeploymentPlan (IP, K)
end

Smaller DRE systems might use a single deployment plan to deploy the whole application, whereas large-scale DRE systems are usually deployed using multiple deployment plans. The local fusion algorithm initially collects the list of components that are deployed onto the different collocation groups (possibly on multiple nodes) and creates physical assemblies from the set of components that are local to that deployment plan.

Algorithm 1 delegates much of its work to several auxiliary functions. Algorithm 2 creates physical assemblies by calculating the cliques in the set of components passed as

Algorithm 2: CreatePhysicalAssemblies (T, I)

Input: **set of ComponentType** T
Input: **set of Component** I
Result: **set of set of Component** K
begin
  while $I \neq \emptyset$ **do**
    Component c; **set of Component** SC;
    $SC \leftarrow \bigcup_{U_{i,j} \subset I, \forall u_i \in U_i, u_j \in U_j} type(u_i) \neq type(u_j) \wedge |U_i| = |U_j|$
    $I = I - SC$
    $K \leftarrow K \cup$ EnumerateCliques (SC)
  end
end

input. Algorithm 1 uses a domain-specific heuristic to construct the set of components which are passed to Algorithm 3. Instead of creating a clique directly out of the all component instances belonging to a collocation group, we create a set of component instances that occur the same number of times. Thus, the heuristic will result in selecting components that occur only once for the first call to Algorithm 3, components that occur twice.

Algorithm 3: EnumerateCliques(I)

Input: **set of Component/QoSPolicy** I
Result: **set of set of Component/QoSPolicy** K
begin
  Graph G;
  set of Component/QoSPolicy V, C; **set of Edge** E;
  $G = (V, E) \mid V = \{i \in I\} \wedge E = \{(u, v) \mid u, v \in V, u \neq v \wedge \text{CanMerge}(u, v)\}$
  while $V \neq \emptyset$ **do**
    $C \leftarrow maximalclique(\text{G})$
    $K \leftarrow K \cup C$
    $V \leftarrow V - C$
  end
end

As a result of our heuristic, either all instances of a single component type are merged into one or more physical assemblies, or it is left alone. The algorithm never creates a

component type that appears both in some physical assembly *and* stand-alone. Without this heuristic, the static footprint of the process will be significantly worse compared to the original footprint. The reason for this overhead is because we will load both the original implementation libraries of the component as well as the new physical assembly into the same process; components which end up being stand alone, as well as part of a physical assembly will contribute to the static footprint twice (or more if they are part of multiple physical assemblies).

Algorithm 3 does not enumerate all the cliques in the graph passed as input to it. Instead it calculates a maximal clique and immediately removes the vertices that are part of that clique from the graph. This removal operation is safe since the vertices that are returned cannot be part of a larger clique because the maximal clique algorithm we use returns the larger cliques first. Thus, the set of vertices passed to the maximal clique algorithm keeps shrinking. Each maximal clique (with more than one vertex) corresponds to a physical assembly.

Algorithm 4: CanMerge(a, b)

Input: **Component/QoSPolicy** a
Input: **Component/QoSPolicy** b
Result: **boolean**
begin
  set of Facet F_a, F_b; **set of Receptacle** R_a, R_b;
  set of Publisher P_a, P_b; **set of Consumer** C_a, C_b;
  set of QoSPolicy QoS_a, QoS_b;
  $QoS_a \leftarrow qospolicies(a)$; $QoS_b \leftarrow qospolicies(b)$;
  $F_a \leftarrow facets(a)$; $F_b \leftarrow facets(b)$;
  $R_a \leftarrow receptacles(a)$; $R_b \leftarrow receptacles(b)$;
  $P_a \leftarrow publishers(a)$; $P_b \leftarrow publishers(b)$;
  $C_a \leftarrow consumers(a)$; $C_b \leftarrow consumers(b)$;
  if $(F_a \cap F_b = \emptyset) \wedge (R_a \cap R_b = \emptyset) \wedge (P_a \cap P_b = \emptyset)$
    $\wedge (C_a \cap C_b = \emptyset) \wedge (P_a \cap R_b = \emptyset) \wedge (P_b \cap R_a = \emptyset)$
    $\wedge (QoS_a \simeq QoS_b)$ **then**
     return ***true***
  else return ***false***
end

In the description of Algorithm 3 (and others) we use a notation **Component/QoSPolicy** to denote the fact that these algorithms are generic, *i.e.*, the same algorithm is applicable to both Component and QoSPolicy (described in Algorithm 7) elements. Algorithm 3 relies on Algorithm 4 to decide if two components can be merged together into a physical assembly. Algorithm 4 performs a series of checks (not all of which is shown) to decide if two components can be merged together. As with Algorithm 3, Algorithm 4 is applicable to both component and QoSPolicy elements (we show a separate function for QoSPolicy in the interest of space). Algorithm 4 is essentially a predicate that will vary from one component middleware domain to another. Each component middleware domain will have a series of conditions that will determine if two components can be merged together. We have shown a version (partial) that applies to CCM in Algorithm 4.

Algorithm 5: UpdateDeploymentPlan (IP, K)

```
Input: DeploymentPlan IP
Input: set of set of Component K
Result: DeploymentPlan OP
begin
    CollocationGroup cg; set of CollocationGroup CG
    Component c,i; set of Component I,C,O;
    OP ← nil
    nodes(OP) ← nodes(IP)
    collocationgroups(OP) ← collocationgroups(IP)
    foreach cg ∈ collocationgroups(IP) do
        foreach i ∈ {c | c ∈ components(cg)} do
            if i ∈ {C | C ∈ K} then
                O ← O ∪ physicalassembly(C)
            else O ← O ∪ i
        end
        components(cg) ← O
    end
end
```

Algorithm 1 creates an updated deployment plan using Algorithm 5, which is primarily responsible for replacing all references to components that are now merged into a physical

assembly with a reference to the physical assembly. It is also responsible (not shown in the algorithm) for updating the "virtual" assembly such that all the connections (within the "virtual" assembly) from/to components merged into a physical assembly are replaced with connections from/to the physical assembly.

IV.2.1.5 Global Component Fusion Algorithm

The second version of the component fusion algorithm is called "Global Component Fusion" and is shown in Algorithm 6. "Global" refers to the fact that this version of the algorithm uses system-wide deployment information and is not constrained to a single deployment plan. The benefits of applying the algorithm at the global scope is measured and analyzed in Section IV.3.

Algorithm 6: Global Component Fusion

```
Input: set of DeploymentPlan IP
Result: DeploymentPlan OP
begin
    Node n; set of Node N; DeploymentPlan p;
    CollocationGroup cg;
    set of CollocationGroup cgs;
    set of set of Component K;
    Component c; set of Component I;
    ComponentType t; set of ComponentType T;
    foreach p ∈ IP do
        N ← {n | n ∈ nodes(p)}
    end
    foreach n ∈ N do
        cgs ← {cg | cg ∈ collocationgroups(n)}
        foreach cg ∈ cgs do
            I ← {c | c ∈ components(cg)}
            T ← {t | t ∈ types(I)}
            K ← K∪ CreatePhysicalAssemblies (T, I)
        end
    end
    foreach p ∈ IP do
        OP ← OP∪ UpdateDeploymentPlan (p, K)
    end
end
```

The global fusion algorithm is similar to the local except that it operates across a set of deployment plans. The global algorithm can find more opportunities for creating physical assemblies. Global fusion is different from local fusion since it merges all deployment plans of a DRE system, instead of updating the individual plans like the local algorithm. The two versions of our component fusion algorithm are focused on reducing the footprint of a component-based DRE system.

IV.2.1.6 QoS Policy Fusion Algorithm

In addition to the component fusion algorithms, we also designed a QoS policy fusion algorithm that helps improve the performance of component-based DRE systems. As described in Section IV.1.2, component-based DRE system performance can be degraded significantly by sub-optimal configuration and assignment of QoS policies.

To remedy the ill-effects of mis-configurations, Algorithm 7, fuses multiple compatible QoS policies into an aggregate QoS policy thereby reducing the number of different QoS policies needed to configure the system. Algorithm 7 is similar to the Algorithm 1 in the concepts employed to fuse the QoS policies. The problem of fusing multiple QoS policies into a minimum number of QoS policies is equivalent to determining the maximum clique on a graph. The main difference is in elements that form the clique, components in Algorithm 1 as opposed to QoS policies here. The other difference is in the construction of the graph itself, *i.e.*, the set of predicates to evaluate to determine if two QoS policies are compatible with each other will vary in a domain-specific fashion.

Algorithm 7 can be combined with Algorithm 1 or Algorithm 6. Conversely, it can be used alone, in which case it only optimizes the performance by merging the QoS policies. If the two algorithms are combined, Algorithm 4 takes into account compatibility of QoS policies (shown as $\simeq$) among components when deciding whether two components can be merged into a single physical assembly.

Algorithm 7 describes the QoS policy fusion technique. As with the component fusion

Algorithm 7: QoS Policy Fusion

Input: **DeploymentPlan** IP
begin
  CollocationGroup cg;
  Component c; **set of Component** I;
  ComponentType t; **set of ComponentType** T;
  set of QoSPolicy Q; **set of set of QoSPolicy** K;
  foreach $cg \in collocationgroups(IP)$ **do**
    $I \leftarrow \{c \mid c \in components(cg)\}$
    foreach $i \in I$ **do**
      $Q \leftarrow Q \cup qospolicies(i)$
    end
    $K \leftarrow K \cup$ EnumerateCliques (Q)
  end
end

algorithm, it collects the set of components that are deployed into the same collocation group. It then collects all the QoS policies that are associated with these components and proceeds to enumerate the maximal cliques among these QoS policies.

A key difference between Algorithm 1 and Algorithm 7 is that when merging QoS policies it is unnecessary to apply the heuristic of merging elements that appear the same number of times. The set of constraints (namely increased footprint due to duplicate contributions) that played a role in using that heuristic no longer apply to this case. QoS policies that belong to the same clique are replaced with an aggregated QoS policy in the virtual assembly in which these policies are members.

One key characteristic of the fusion algorithms described in this section is that they do not require component developers to change the design or the structure of the DRE system. All the algorithms use the deployment context information and perform optimizations that are applicable for a specific deployment. Instead of forcing changes to the design that are applicable to only one deployment context, the algorithms work in the other direction, *i.e.*, by changing the optimizations depending on the constraints of a deployment scenario.

Although our approach works well for systems whose compositions are static, (*i.e.*, the

system composition does not change after deployment) or semi-static (*i.e.*, changes at run-time are restricted to a set of fixed alternatives), it introduces some complications in case of "highly dynamic" systems. By "highly dynamic," we refer to the fact that both the decisions of which component should be connected to which other components and the nodes these components should be deployed to, are done at run-time. In this scenario, the requirements in our algorithm to calculate the physical assemblies (*i.e.*, determination of maximal cliques), to dynamically generate the implementations for the physical assemblies and to compile them, may not be feasible in severely resource constrained run-time environments.

IV.2.2 Design and Functionality of the Physical Assembly Mapper

The algorithms described in Section IV.2.1 are sufficiently complicated that attempting to perform them manually will not scale for large-scale DRE systems. We can effectively rule out any manual attempt to perform these optimizations for large-scale DRE systems. The automation of the algorithms using existing methodologies like writing *ad hoc* scripts tools, results in a very brittle tool-chain. The main reason for the brittleness is that the vocabulary used to describe information such as the interface definition files of the components, the various deployment metadata like deployment plans, QoS configuration files necessary to perform these optimizations are disparate.

For example, plain text is typically used for interface definition files, XML is used for deployment descriptors and QoS configuration are usually platform-specific, *i.e.* usually a mix of both proprietary plain text as well as XML. Writing software that deals with such disparate sources of input is tedious and error-prone. There is a need for a higher-level abstraction that allows dealing with these disparate sources of information in a unified fashion. Model-Driven Engineering [116] is a promising approach to providing this much sought after high-level abstraction.

Although models are not a pre-requisite for the creation of physical assemblies, it is hard to both identify as well as optimize component implementations in the absence of

a well-defined description of the systems that can be processed by tools automatically. Performing such optimization of components is challenging since the usage of any single component tends to span across multiple compositional hierarchies, *i.e.*, a single component could be connected to different sets of components in different assemblies, in any complex system.

Since components tend to get reused across an entire product line, an optimization that is applicable in one context may not be applicable in another context. It is not possible to perform these optimizations in isolation, but rather one should perform them based on the requirements of every unique deployment. Finally, performing these optimizations manually by hand becomes infeasible due to the presence of a large number of components as well as due to changes caused by system evolution.

IV.2.2.1 Implementation of Fusion Algorithms in Physical Assembly Mapper

As described in Chapter III, we developed *Platform-Independent Component Modeling Language* (PICML) to enable developers of component-based DRE systems to define component interfaces, inter-connect components, define QoS policies and also generate valid deployment metadata that enable automated system deployment. Other aspects of our work on model-driven tools include *Component QoS Modeling Language* (CQML) [60], which models the QoS configuration options required to implement the QoS policies of the application specified in PICML.

To demonstrate our optimization techniques, we developed a prototype optimizer called Physical Assembly Mapper(PAM), which builds upon our previous work on both PICML and CQML to implement the fusion algorithms described in Section IV.2.1. In particular, PICML enables functional composition of DRE systems from monolithic components by connecting them together into a component assembly. These assemblies are *virtual* in the sense that the components that make up such assemblies can be deployed across many nodes of the target deployment domain. CQML complements and extends PICML

by enabling non-functional decomposition by allowing specification of QoS policies on individual components. PAM automatically creates a physical assembly from the set of virtual assemblies that define the application structure. PAM utilizes both the connectivity information between components modeled as well as the QoS policies to create *physical* assemblies.

PAM is implemented as a model interpreter, a DSML-specific tool written using C++ for use with GME. As described in Sidebar 1 of Chapter III, GME is a meta-programmable environment for creating DSMLs. PAM also utilizes the Boost Graph Library [125] to implement the maximal clique algorithm discussed in Section IV.2.1. Figure IV.2 presents an overview of the optimization process performed by PAM. Optimization using PAM consists of three phases: a model transformation phase described in Section IV.2.2.2, a glue code generation phase described in Section IV.2.2.3 and a configuration files generation phase described in Section IV.2.2.4. Along with each phase, we also describe how the phase solves the challenges described in Section IV.2.1.1.

IV.2.2.2 Model Transformation in PAM

The input to PAM is the input model that captures the application structure and the QoS configuration options. The input model of the DRE system contains information about the individual component interface definitions, their corresponding monolithic implementations, collections of components connected together in a system-specific fashion to form virtual assemblies, associations of components with QoS configuration options.

PAM implements Algorithm 1, the local component fusion, Algorithm 6, the global component fusion as well as Algorithm 7, the QoS policy fusion algorithm to rewrite the input model into a functionally equivalent model. As part of this model transformation, PAM creates physical assemblies including interface definitions for the physical assemblies. Since the algorithms perform a series of checks before deciding to merge components

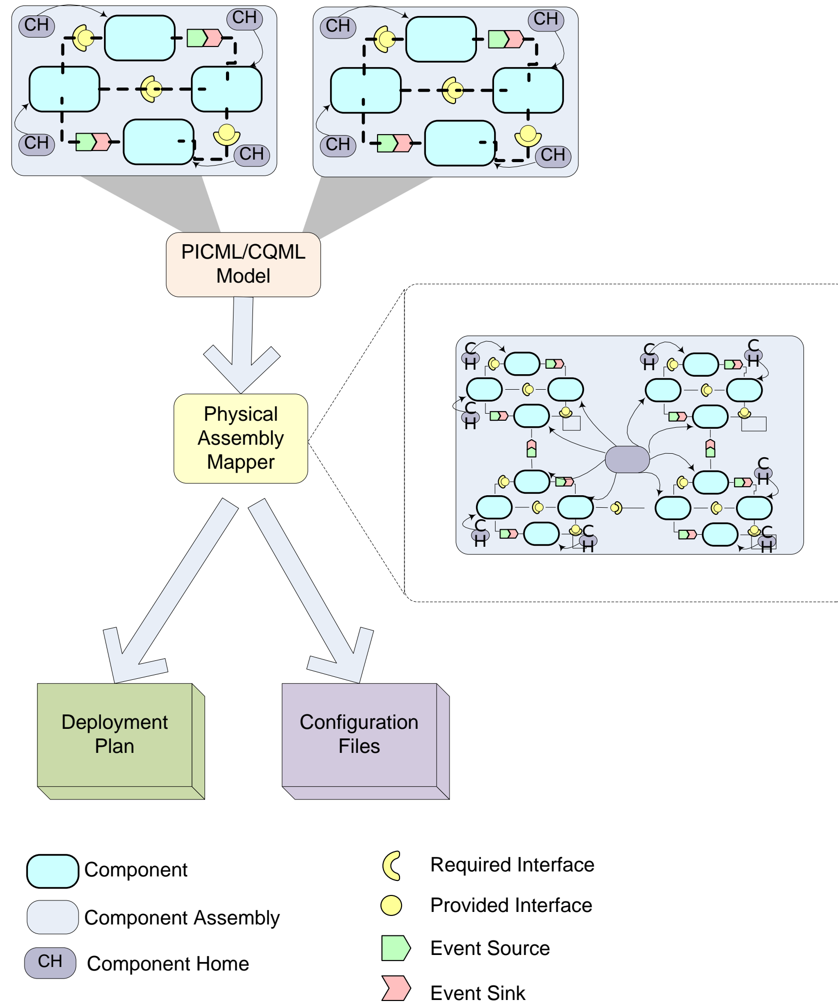


Figure IV.2: Physical Assembly Mapper

together, the issues with ensuring unique port names described in challenge 1, as well as resolving incompatibility between QoS policies described in challenge 5 of Section IV.2.1.1 are non-existent when using PAM.

For each such physical assembly created, PAM replaces the original set of component instances with an instance of the newly created physical assembly. This writing replaces all the connections to/from the original components with connections to/from the physical assembly. PAM also creates new attributes corresponding to each attribute of all the individual components ensuring that there is no clash in the attribute names within the physical

assembly namespace. PAM solves the problem with maintaining the state of the individual components separately described in challenge 3 of Section IV.2.1.1.

To facilitate the lookup of the original components by external clients using mechanisms such as CORBA Naming Service, LDAP and Active Directory servers, PAM creates configuration properties in the model associated with each physical assembly. These configuration properties create multiple entries, one corresponding to each unique name used by the original components in lookup services, and ensures that all these names point to the physical assembly. PAM solves challenge 4 described in Section IV.2.1.1.

IV.2.2.3 Generation of Glue-code in PAM

Once the model has been rewritten into a functionally equivalent optimized model, PAM utilizes a number of model interpreters to generate various artifacts related to the middleware glue code. This middleware glue code is necessary to use the physical assemblies created in the model with the existing monolithic implementations of the components. The glue code generated by PAM creates a composite context by inheriting from the individual contexts of the components that make up the physical assembly. This derived context is compatible (due to inheritance) with each monolithic component's context and can be supplied to the individual component implementations at run-time by the container.

The glue code generated for the physical assemblies can be compiled and deployed with the implementations of the other components in the system. PAM solves the challenge associated with providing a compatible context to the original component implementations described in challenge 2 of Section IV.2.1.1. Since PAM performs the generation without requiring modifications to individual component implementations, our original goal of not imposing a burden on the component developer by requiring changes to the original implementation is also achieved.

IV.2.2.4 Generation of Configuration Files in PAM

In addition to the middleware glue code, PAM also generates modified metadata such as deployment plans and QoS policy configuration files. When Algorithm 1 is applied, PAM generates deployment plans in which the components that have been merged to form physical assemblies are replaced with the physical assemblies. All references to the original components are also replaced with references to the physical assemblies. The replacement of components (and their references) is done at the scope of a single deployment plan by the implementation of Algorithm 1 in PAM.

When Algorithm 6 is applied, PAM generates a single deployment plan. Since the optimizations are applied at the scope of the entire application, PAM merges the different deployment plans to create a single aggregate deployment plan. PAM then replaces the original components merged together to form physical assemblies with the physical assemblies including the replacement of references as done for Algorithm 1.

When Algorithm 7 is applied (either stand-alone or in combination with Algorithm 1 or Algorithm 6), PAM creates aggregate QoS policies which serve roles similar to the physical assemblies. PAM replaces the original QoS policies associated with the components with these new aggregate QoS policies in each assembly where such aggregate QoS policies are created. The deployment plans generated refer to these aggregate QoS policies instead of the original QoS policies. PAM can be thus viewed as a "deployment compiler" that takes the application components, their QoS policies and their deployment information as input, and produces as output a set of physical assemblies and aggregate QoS policies that eliminate the overhead due to auxiliary middleware infrastructure elements or QoS misconfiguration.

IV.3 Empirical Evaluation and Analysis

To evaluate the benefits of our fusion algorithms described in Section IV.2.1, we applied PAM on several representative DRE applications: (1) an application from the shipboard

computing domain [66] and (2) Boeing's Boldstroke Single Processor [122] scenario from the avionics mission computing domain. This section describes the characteristics of the two applications, explains the experiment testbed architecture, presents the experiments to evaluate footprint improvement, followed by the experiments to evaluate the invocation latency. Our experiments compare the time/space properties of applications developed using standard CCM configurations against the execution of these applications after applying PAM to optimize the application.

IV.3.1 Experimental Platforms

IV.3.1.1 Shipboard Application

Our first application is from the domain of shipboard computing. A shipboard computing environment is a metropolitan area network (MAN) of computational resources and sensors that provides on-demand situational awareness and actuation capabilities for human operators, and responds flexibly to unanticipated runtime conditions. To meet such demands in a robust and timely manner, the shipboard computing environment uses component-based services to bridge the gap between shipboard applications and the underlying operating systems and middleware infrastructure to support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization. We use the shipboard computing application to evaluate the footprint benefits due to our component fusion algorithms.

The shipboard computing environment that we used for our experiments was developed using the CIAO middleware [55]. This application consists of a number of components grouped together into multiple *operational strings*. As shown in Figure IV.3, an operational string is composed of a sequence of components connected together. Each operational string contains a number of sensors (*e.g.*, `ed1_A, ed2_A` shown on the far left) and system monitors (*e.g.*, `sm1_A, sm2_A` shown at the top) that publish data from the physical devices as well as the overall system state to a series of planners. After analyz-

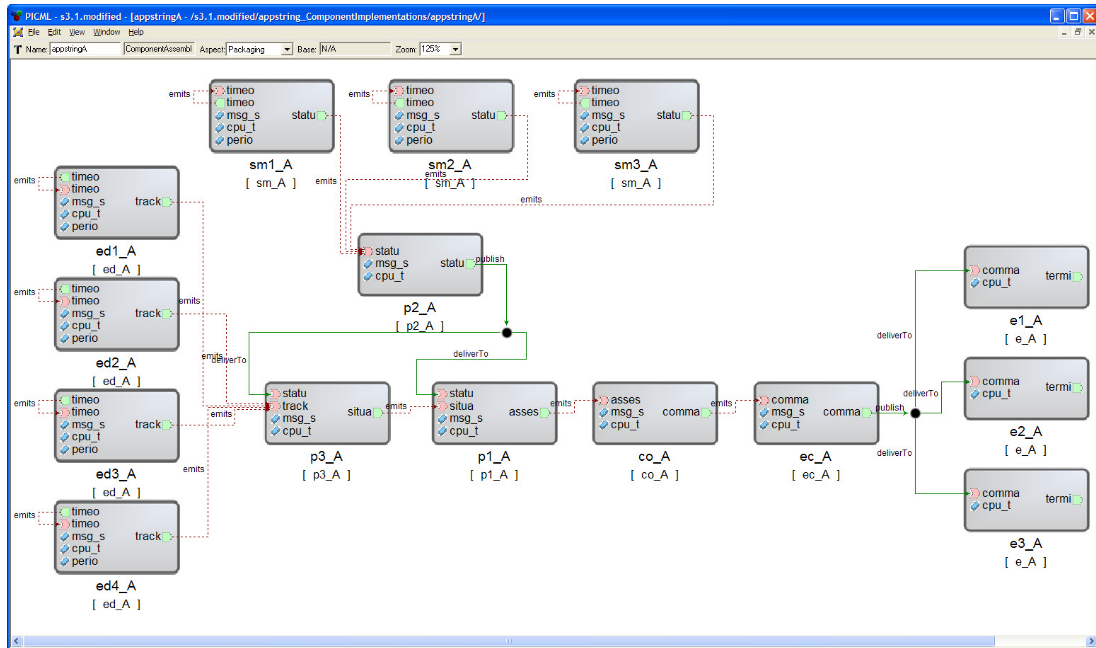


Figure IV.3: Sample Operational String

ing the sensor data and the inputs from system monitors, the planners (*e.g.*,`p1_A, p2_A` shown in the center) perform control decisions using the effectors (*e.g.*, `e1_A, e2_A` shown on the far right). Each operational string contains 15 components altogether, and the application used in our experiments is made up of 10 such operational strings, for a total of 150 components. Operational strings are at different importance levels. In case of a resource contention, the higher importance operational strings receive priority when accessing a resource.

The application itself is deployed using 10 different deployment plans across 5 different physical nodes named bathleth, scimitar, rapier, cutlass, and saber. The assignment of components to nodes was determined *a priori* using high-level resource planning algorithms, and was available as input to our algorithms. Each node had a variable number of components, ranging from 20 to as high as 80 components assigned to it.

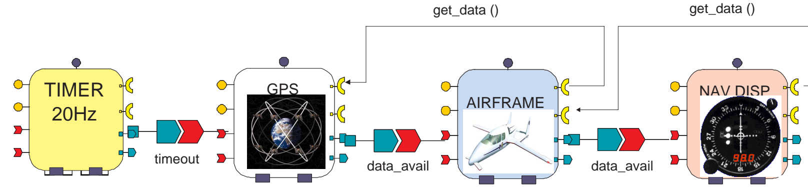


Figure IV.4: Basic Single Processor Scenario

IV.3.1.2 Boeing Basic Single Processor

The Basic Single Processor (BasicSP) scenario from the Boeing Bold Stroke component avionics mission computing project, uses a *push event and pull data* publisher/-subscriber communication paradigm [123] atop a QoS-enabled component middleware platform. In conjunction with colleagues at The Boeing Company [124] and Washington University [144], we have developed a prototype of the BasicSP application described above using the CCM and Real-time CORBA capabilities provided by CIAO [145]. We use the BasicSP example to evaluate the invocation latency benefits due to our policy fusion algorithms.

The BasicSP application comprises of four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays. As shown in Figure IV.4, a Timer component triggers a GPS navigation sensor component, which in turn publishes position information to an Airframe component.

Upon receiving the data availability event, the Airframe component pulls data from the GPS, and informs a Nav_Display component. The Nav_Display component then updates the display by pulling position data from the Airframe component. The application requests new inputs from the GPS component at a rate of 20 Hz, and updates the display outputs with new aircraft position data at a rate of 20 Hz. The latency between the inputs to the application and the output display should be less than a single 20 Hz frame.

For the BasicSP scenario to satisfy the QoS requirement of ensuring display refresh rate of 20 Hz, it is necessary to examine the end-to-end critical path and configure the

components appropriately. In particular, the latency between the Airframe and Nav_Display components should be minimized.

IV.3.2 Experimental Setup

We used ISISlab (`www.dre.vanderbilt.edu/ISISlab`) which is an open testbed for experimentation on distributed real-time and embedded (DRE) systems and distributed continuous quality assurance. ISISlab has 56 dual-CPU blades with 2.8 GHz XEONs, 1 GB memory, 40 GB disks, and 4 Network Interface Cards (NICs) per blade. It is assembled as 4 blade centers with 14 blades per center with 6 Cisco 3750G switches consisting of 24 10/100/1000 Mbps ports per switch. ISISlab uses the Emulab [147] software from the University of Utah to support the installation of various versions of Linux, BSD UNIX, Windows, and Solaris.

Our experiments used version 0.5.10 of CIAO running on Windows XP SP2 and Linux with Ingo Molnar's real-time pre-emption patches [84]. For the footprint experiments using the shipboard computing application described in Section IV.3.1.1, we used 5 blades running Windows XP SP2. All the machines were connected on the same local network and connected to each other using Gigabit ethernet.

We measured the footprint of the components in the deployed shipboard computing application using Virtual Address Dump (VaDump)[1] distributed with the Windows Resource Kit Tools [80]. VaDump is a command-line tool that creates a list containing information about the memory usage of a specified process. VaDump examines the virtual address of a running process. Depending on the options specified, the output of VaDump can include: (1) each address, along with its size, state, protection, and type, (2) total committed memory for the image, the executable file, and each shared library, including system shared libraries, and (3) total mapped committed, private committed, and reserved memory. We used VaDump to measure both static (code and static data) and dynamic (heap memory)

[1] VaDump is similar to the utility "pmap" that exists on Linux and UNIX, but provides more fine grained detail, including the amount of stack and heap committed to a process.

footprint of the components by taking a snapshot of the process that creates the container hosting the components on each machine.

For the latency experiments using the Basic Single Processor application, we used Linux kernel version 2.6.20-rt8 with real-time pre-emption patches. All processes hosting the components were run in the POSIX scheduling class, SCHED_FIFO, which enables first-in-first-out scheduling semantics based on the priority of the process. In our experiments, we measured the end-to-end invocation latency from the Timer component to the Nav_Display component by repeating the invocations for 250,000 iterations. The Timer component was configured to run at a fixed priority using the SERVER_DECLARED RT-CORBA *PriorityModel* policy, whereas the rest of the components were configured with the CLIENT_PROPAGATED policy.

IV.3.3 Empirical Footprint Results

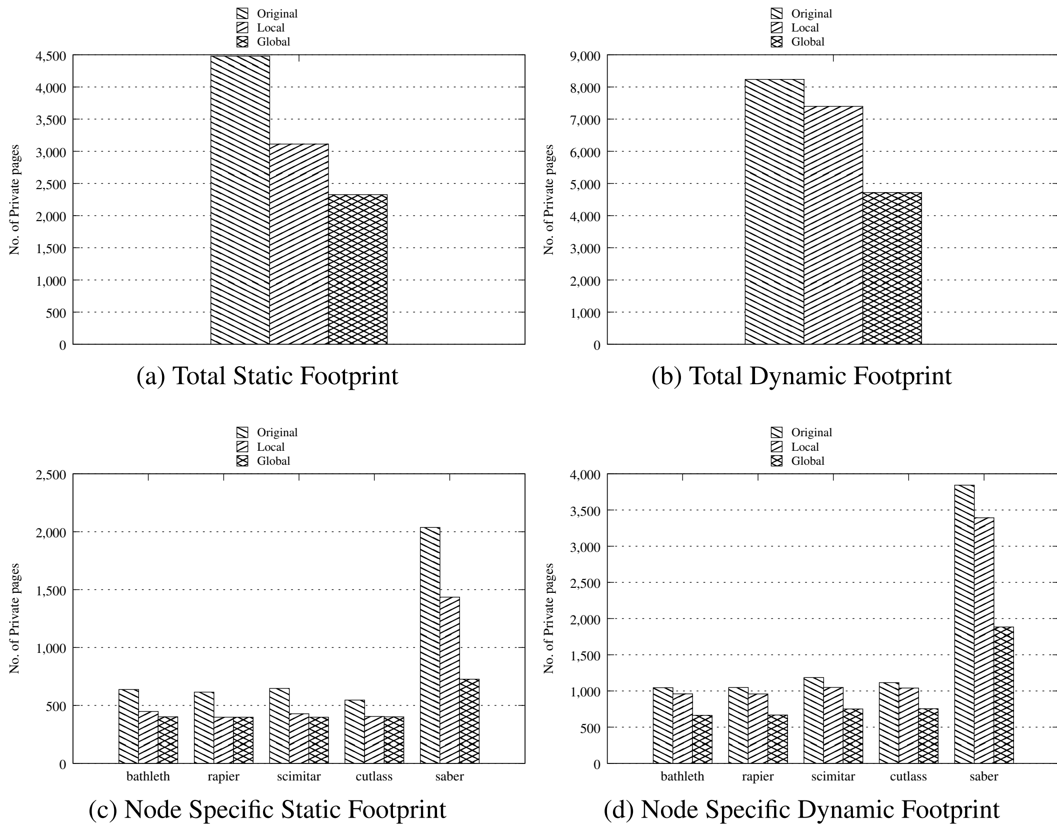
Experiment design. To measure the footprint of the shipboard computing application, we deployed the 10 operational strings across the 5 nodes using 10 deployment plans. We allowed the application to execute for 5 minutes and measured the footprint of the components by running VaDump on the process hosting the components on each node. We refer to this run of the experiment as *Original* in the graphs shown below.

We used PAM (off-line) on the input model by invoking it to use the local component fusion algorithm described in Section IV.2.1.4 and repeated the experiment using the 10 locally optimized deployment plans generated. We refer to this run of the experiment as *Local* in the graphs shown below. Finally, we used PAM (also done off-line) on the input model by invoking it to use the global component fusion algorithm described in Section IV.2.1.5 and repeated the experiment using the single global deployment plan generated. We refer to this run of the experiment as *Global* in the graphs shown below.

Analysis of results – Static Footprint. Figure IV.5a compares the static footprint, which includes the footprint contribution from code and the static data of the whole application

deployed across all the 5 nodes. We measure the footprint of the application as the sum of the number of private and shareable pages (as opposed to shared) of the processes hosting the components using VaDump. The three runs of the experiment did not include the contributions from the operating system and middleware shared libraries, since they were unaffected by our optimizations.

As shown in Figure IV.5a, the original static footprint of the application was 4,478 pages and the application of the local component fusion algorithm reduced it to 3,110, which is an improvement of 31%. Applying the global fusion algorithm reduced the static footprint further to 2,324 pages, which is an improvement of 49%. The creation of physical assemblies by the component fusion algorithms therefore significantly reduced the static footprint of the application.



(a) Total Static Footprint (b) Total Dynamic Footprint

(c) Node Specific Static Footprint (d) Node Specific Dynamic Footprint

Figure IV.5: Static and Dynamic Footprint

Analysis of results – Dynamic Footprint. Figure IV.5b compares the dynamic footprint of the application. The contributions here are primarily from the dynamic allocation of memory by the application in the three runs. Unlike the static footprint measurements, measuring the dynamic footprint of the application captures the heap usage of the whole process, since VaDump does not provide the heap usage of individual shared libraries. Since we could not precisely pinpoint the heap usage of individual shared libraries, our dynamic footprint results are not as fine-grained as the static footprint results.

As shown in Figure IV.5b, the original dynamic footprint of the application was 8,231 pages. Application of the local fusion algorithm reduced it to 7,393 pages, which is an improvement of 11%. Application of the global fusion algorithm reduced it to 4,713 pages, which is an improvement of 43%. The reduction in dynamic memory stems primarily from reducing the number of homes and component context created in the physical assemblies. The increased reduction in the global compared to local is due to increased opportunities for creating physical assemblies (*i.e.*, the scope is across the entire application), as well as the merging of multiple deployment plans into a single deployment plan, which reduces the number of processes required to deploy the application.

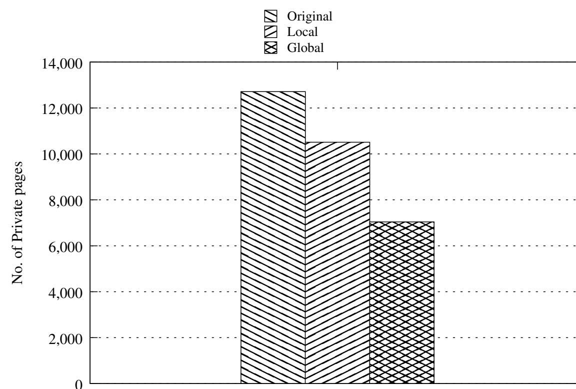


Figure IV.6: Total Footprint

Figure IV.6 shows the combined footprint of the application. As shown in Figure IV.6, the combined footprint of the original application was 12,709 pages, the application of local

fusion algorithm reduced it to 10,503 pages, which is an improvement of 18%. The application of the global fusion algorithm reduced it to 7037 pages, which is an improvement of 45%.

Figure IV.5c and Figure IV.5d provide the breakup of the total footprint across the different nodes. The increased footprint in the case of node saber in the three runs is due to the number of components deployed on that node.

IV.3.4 Empirical Latency Results

Experiment design. The purpose of this experiment was to demonstrate the benefits of our QoS policy fusion algorithm. As described in Section IV.1.2, the effects of invocations between components residing in different containers is orders-of-magnitude slower than components that reside in the same container. Since each container is associated with a unique QoS policy, our QoS policy fusion algorithm merges the QoS policies to reduce the effective number of containers necessary for a given application.

In all the experiments involving latency, we show four subgraphs that plot (1) the *mean latency*, which is a measure of the normal behavior, (2) the *maximum latency*, which is a measure of the absolute worst-case behavior, (3) the *standard deviation* of the latency, which is a measure of the jitter, and (4) the *99% latency*, which is a measure of the statistical worst-case behavior. All of these numbers are for components in the same address space, *i.e.*, the source and the target components/objects are in the same process.

Analysis of results – Vanilla CORBA objects. First, we show the difference in invocation latency between two normal CORBA objects (not CCM components), when the source and the target objects are configured with different QoS policies. These experiments were conducted using TAO, our real-time QoS enabled implementation of CORBA (CIAO is built on top of TAO). Figure IV.7 shows the difference in the latency when the source and the target objects were configured with different QoS policies. Table IV.1 describes the 8 different policy set combinations used in this experiment.

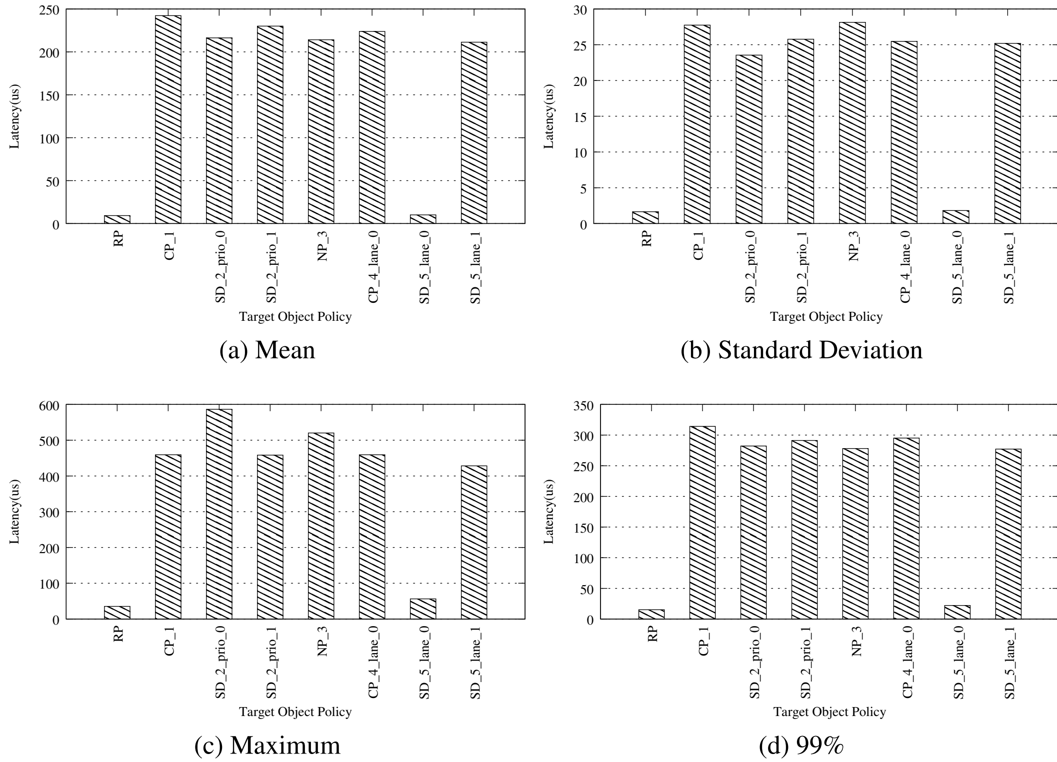
Table IV.1: QoS Policy Configuration

Object	POA	ThreadPool (Lane Priority)	PriorityModel (Priority)
RP	RootPOA	n/a	n/a
CP_1	1	yes	CLIENT_PROPAGATED
SD_2_0	2	yes	SERVER_DECLARED (0)
SD_2_1	2	yes	SERVER_DECLARED (1)
NP_3	3	yes	n/a
CP_4_0	4	yes(0)	CLIENT_PROPAGATED
SD_5_0	5	yes(0)	SERVER_DECLARED (0)
SD_5_1	5	yes(1)	SERVER_DECLARED (1)

The source object in Figure IV.7 was configured with policy set `SD_5_lane_0`. As shown in Figure IV.7, when the target object is in a POA with a different QoS policy set the latency is ∼210µs. When the target object is in a POA without a QoS policy (data set 1), the latency is ∼9µs. When the target object is in a POA with the same QoS policy (data set 7) the latency is ∼10µs. An order-of-magnitude difference (210 vs. 10) in invocation latency is exhibited by these results.

The large difference in Figure IV.7 is due to TAO not using the collocated invocation path, *i.e.*, the calls were going through the local loopback interface, when the source and the target object policies were different. Our initial attempt to fix the problem involved enhancing the TAO so it used the collocated invocation path even when the source and the target objects have different QoS policies. We call this the "cross-pool/lane optimization" because the invocations are collocated even when they cross over different thread pools and/or thread pool with lanes.

Analysis of results – Cross-pool/lane optimizations. Figure IV.8 compares the invocation latency before and after applying the cross-pool/lane optimizations. This figure shows how the cross-pool/lane optimizations reduce invocation latencies across the board, *i.e.*, irrespective of the policy of the target object to about ∼50µs. In the absence of the cross-pool/lane optimizations, TAO treated objects which used different thread pools/thread-pool with lanes as equivalent to a remote object even if they were collocated, *i.e.*, were part of



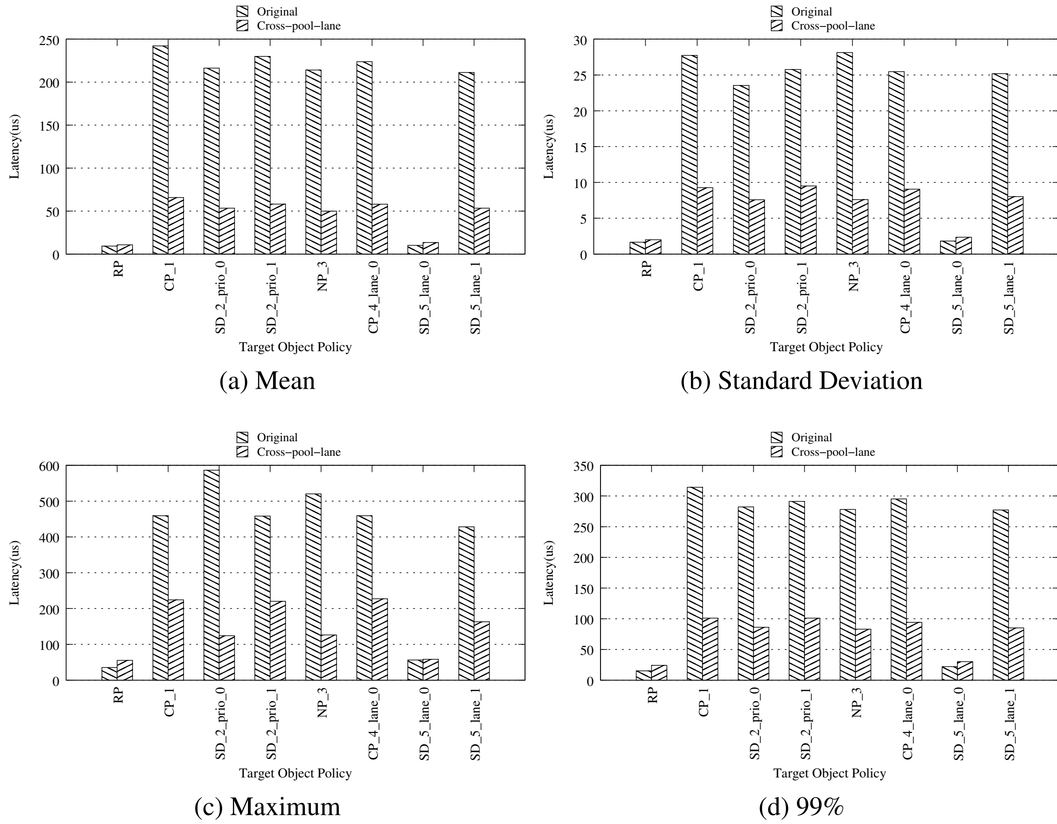
(a) Mean (b) Standard Deviation (c) Maximum (d) 99%

Figure IV.7: Original Latency Measurements

the same address space. This was done to ensure that the requests on objects using different thread pools were invoked with the correct set of priorities (which could be different for different thread pools/thread pool with lanes) so that any conditions for priority inversion are prevented [109].

We performed the cross-pool/lane optimizations with an eye on reducing the overhead due to un-necessary (de-)marshaling as well the overhead due to going through the local loopback network interface, for such objects that were deemed to be remote even if they were collocated. The cross-pool/lane optimizations implement a novel inter-thread pool/lane signaling protocol using pipes between threads thereby avoiding both (de-)marshaling as well as data copying between user-space and kernel-space. The cross-pool/lane optimizations thus, eliminates the overhead due to (de-)marshaling overhead and avoids going through the local network stack as well, while ensuring that the requests are still handled

by threads corresponding to the thread-pool/lane with the correct priority. Although it is better than the original implementation, the signaling protocol still causes a context switch between threads. The context switch is responsible for the increase in invocation latency to 50µs compared to the 10µs for invocations between objects with same QoS policy sets.



(a) Mean (b) Standard Deviation

(c) Maximum (d) 99%

Figure IV.8: Cross-pool/Lane Latency Measurements

Analysis of Results – QoS policy fusion and cross-pool/lane optimizations. Finally, we show the application of our QoS policy fusion algorithm combined with the "cross-pool/lane" optimizations to our Basic Single Processor scenario. As described in Section IV.3.2, the original application has 4 components each with a separate QoS policy configured. As described in Section IV.2.1.6, the QoS policy fusion algorithm merges

compatible QoS policies into aggregate QoS policies. By using the implementation of this algorithm in PAM, we reduced the number of unique QoS policy sets in the application.

The reduction in QoS policy sets results in components that use the aggregate QoS policies created to be placed in the same container. Since all components within the same container share the middleware resources like thread-pools/thread-pool with lanes, the latency of invocation between these components is reduced (as desired) to levels comparable to the collocated case. To alleviate the increased latencies in case the QoS policy fusion could not combine the QoS policies, we used a version of TAO that implemented our cross-pool/lane optimizations. The combination of the application of the QoS policy fusion via PAM and the cross-pool/lane optimizations in the middleware resulted in transforming all calls that are made between components in the same process to either use collocated invocation or use the cross-pool/lane invocation. Only when two components are in a different processes (on the same machine) or on a different machine, the remote invocation path is used.

Figure IV.9 compares the end-to-end invocation latency from the Timer component to the Nav_Display component. Figure IV.9 shows that the cross-pool/lane optimizations done at the middleware level reduces the invocation latency from about 1,509µs to about 554µs, which is a decrease of 64%. After applying the QoS policy fusion algorithm (denoted by PAM in Figure IV.9), moreover, the latency reduced to 293µs, which is a decrease of 81%.

The significant reduction is latency stems from reducing the number of unique QoS policies, which in turn, reduces the number of different containers created, thereby resulting in multiple components being hosted within the same container. In particular, in the case of the Basic Single Processor scenario, our optimizations resulted in the creation of two containers as opposed to the 4 containers originally present. The primary factor that determines the invocation latency between components is whether the local path is chosen or if the remote path is chosen.

Our latency experiment results demonstrate the benefits of placing components into

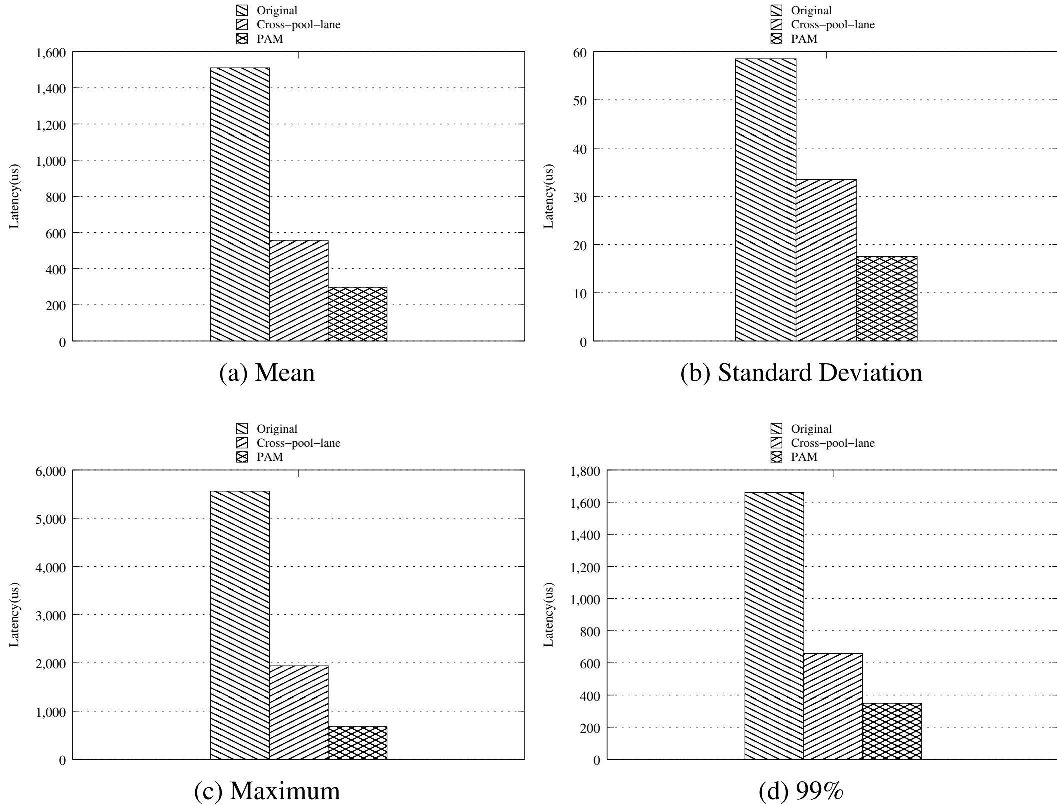


Figure IV.9: Basic Single Processor Latency Measurements

the same container (when the QoS policies are compatible) and the benefits of the cross-pool/lane optimizations (when the QoS policies are incompatible), both on a per-hop basis (shown in Figure IV.8) as well as across the end-to-end application invocation path (shown in Figure IV.9. In general, we expect our results will apply for large-scale component-based applications. The primary difference will be an increase in the absolute value of the latency depending on the number of components in the end-to-end invocation chain. By fusing QoS policies into aggregate QoS policies automatically using deployment information available in the model, our algorithm can reduce the number of QoS policies required to deploy an application contributing to significant performance benefits.

IV.4 Summary

Component middleware technologies like EJB and CCM allow developers to build systems that have a large number of components. The increase in the number of components not only imposes increased demands on DRE system resources but also increases the complexity of QoS configuration significantly. Without sophisticated tools and techniques for managing the complexity of large-scale component-based DRE systems, the benefits of using component middleware may be negated by the excessive resource demands and complexity of QoS configuration. Thus, there is a need for optimization techniques in large-scale component-based DRE systems to ensure that the productivity benefits of component middleware are realized without compromising the overall requirements of the system.

Although optimizing component middleware to satisfy the requirements of applications is not new, component middleware technologies offer some new avenues for optimization of middleware. Unlike the traditional dichotomy of design/development-time vs. run-time optimization of middleware, the presence of an explicit application "deployment" step in component middleware, opens the possibilities for a new class of optimizations called "deployment-time" optimizations.

In this chapter, we described a model-driven engineering (MDE) approach to performing "deployment-time" optimizations. Our approach includes a family of related optimization techniques, all of which use the notion of "fusion" – combining multiple elements into a single element – to reduce the number of elements without affecting the original semantics. We described three algorithms – Local Component Fusion, Global Component Fusion, QoS Policy Fusion – which differ not only in the type of elements they operate on but also in the scope at which they operate.

We implemented the three algorithms in a prototype MDE tool called Physical Assembly Mapper (PAM), which is an enhancement to Platform-Independent Component Modeling Language (PICML), a Domain-Specific Modeling Language (DSML) that supports development of component-based DRE systems using the CORBA Component Model

(CCM). We conducted experiments on applying the techniques implemented in PAM on a couple of representative DRE systems. Our results indicate that the "deployment-time" optimization techniques in PAM provides 45% improvement in footprint and 81% improvement in latency when compared to conventional component middleware technologies.

PAM, PICML, and CQML are open-source and available for download at www.dre.vanderbilt.edu/cosmic.

CHAPTER V

TECHNIQUES FOR INTEGRATING COMPONENT-BASED SYSTEMS

With the maturation of commercial-off-the-shelf (COTS) component middleware technologies, such as Enterprise Java Beans (EJB) [134], CORBA Component Model (CCM) [88], and Microsoft .NET Framework [81], software developers are increasingly faced with the task of integrating heterogeneous enterprise distributed systems built using different COTS technologies, rather than just integrating proprietary software developed in-house. Although there are well-documented patterns [49] and techniques [14] for integrating systems via various component middleware technologies, system integration is still largely a tedious and error-prone manual process. To improve this process, component developers and system integrators must understand key properties of the integration technologies they are applying and the systems[1] they are integrating.

There are multiple levels at which system integration is achieved [139], including:

Data integration, which integrates systems at the logical data layer, typically using some form of data transfer/sharing. Example technologies that allow data integration include commercial databases (such as IBM DB2, Oracle, and Microsoft SQL Server) and tools (such as Microsoft BizTalk Mapper and IBM WebSphere Integration Developer) that provide database schema mapping between different databases.

Functional integration, which integrates systems at the logical business layer, typically using distributed objects/components, service-oriented architectures, or messaging middleware. Examples of technologies that allow functional integration include the Java Connector Architecture and Service-Oriented Integration adapters available in commercial products, such as IBM's Websphere.

[1] In the remainder of this chapter "system" or "application" refers to an enterprise distributed system built using component middleware like EJB, Microsoft .NET, or CCM.

Presentation integration, which allows access to an application's functionality through its user interface by simulating a user's input and by reading data from the screen. This "screen scraping" is usually done via programming languages like Perl that use regular expressions to parse the screen output of legacy systems.

Portal integration, which creates a portal application that displays information retrieved from multiple applications via a unified user interface, thereby allowing users to perform required tasks. Examples of technologies that allow portal integration include Microsoft ASP.NET and Java portlets combined with Java Server Pages (JSP), which provide technologies to build web-based portals for integrating information from a variety of sources.

Process integration, which defines a business process model that describes the individual steps in a complex business function and coordinates the execution of long-running business functions that span multiple disparate applications. Example technologies that support process integration include implementations of Business Process Execution Language (BPEL) and its web services incarnation (WS-BPEL).

This chapter describes technologies that help simplify the *functional integration* of systems built using component middleware. This type of integration operates at the logical business layer, typically using distributed objects/components, exposing service-oriented architectures, or messaging middleware, and is responsible for delivering services to clients with the desired quality of service (QoS). We focus on functional integration of systems in this chapter since:

- Component middleware is typically used to implement the core business logic of a system. In this context it is inappropriate to use portal integration since there may be no direct user interaction and because component middleware usually resides in the second tier of a typical three-tier enterprise architecture. In contrast, the entities that make up a "portal," *e.g.*, portlets, belong in the first tier, *i.e.*, front-end.

- Unlike legacy systems, component middleware technologies usually expose an API to access functionality. Employing presentation integration to integrate systems built

using component middleware technologies is problematic. For example, techniques used in typical presentation integration (such as parsing the output of a system to enable integration) are *ad hoc* compared with using the well-defined APIs exposed by component middleware technologies.

- Updates to data at the business logic layer occur frequently during system execution. Due to the cost of remote data access operations and the rate at which such operations are generated by the business logic components in the second tier of a three-tier enterprise architecture, it is infeasible to employ data integration to keep the data consistent among the different systems. Data integration is usually appropriate for the back-end (*i.e.*, third tier) of a three-tier enterprise architecture, where the data is long-lived and not transient.

- The business logic of a system is often proprietary and organizations tightly control the interfaces exposed by the system. It is often unnecessary to employ process integration, which usually applies to inter-organizational integration where loose-coupling is paramount. Process integration is a superset of functional integration and usually relies on functional integration within autonomous organizational boundaries.

Functional integration of systems is hard due to the variety of available component middleware technologies, such as EJB and CCM. These technologies differ in many ways, including the protocol level, the data format level, the implementation language level, and/or the deployment environment level. In general, however, component middleware technologies are a more effective technology base than the brittle proprietary infrastructure used in legacy systems [122], which have historically been built in a vertical, stove-piped fashion.

Despite the benefits of component middleware, key challenges in functional integration of systems remain unresolved when integrating large-scale systems developed using heterogeneous COTS middleware. These challenges include (1) *integration design*, which

involves choosing the right abstraction for integration, (2) *interface mapping*, which reconciles different datatypes, (3) *technology mapping*, which reconciles various low-level issues, (4) *deployment mapping*, which involves planning the deployment of heterogeneous COTS middleware, and (5) *portability incompatibilities* between different implementations of the same middleware technology. The lack of simplification and automation in resolving these challenges today significantly hinders effective system integration.

A promising approach to address the functional integration challenges outlined above is *Model-Driven Engineering* (MDE) [116], which involves the systematic use of models as essential artifacts throughout the software lifecycle. At the core of MDE is the concept of *domain-specific modeling languages* (DSMLs) [59], whose type systems formalize the application structure, behavior, and requirements within particular domains. DSMLs have been developed for a wide range of domains, including software defined radios [138], avionics mission computing [58], warehouse management [28], and even the domain of component middleware [148] itself.

Although DSMLs have been used to help software developers create homogeneous systems [58, 129], enterprise distributed systems are rarely homogeneous. A single DSML developed for a particular component middleware technology, such as EJB or CCM, may not be applicable to model, analyze, and synthesize key concepts of web services. To integrate heterogeneous systems successfully, system integrators need automated tools that provide a unified view of the entire enterprise system, while also allowing fine-grained control over specific subsystems and components.

Our approach to integrating heterogeneous systems is called *(meta)model composition* [68], where the term "(meta)model" conveys the fact that this technique can be applied to both metamodels *and* models. At the heart of this technique is a method for

- Creating a new DSML (a composite DSML) from multiple existing DSMLs (component DSMLs) by adding new elements or extending elements of existing DSMLs,

- Specifying new relationships between elements of the component DSMLs, *e.g.*, relationships that capture the semantics of the interaction between elements of the two previously separate component DSMLs,
- Defining relationships between elements of the composite DSML and elements of the component DSMLs, *e.g.*, relationships that define containment of elements of component DSMLs inside elements of composite DSMLs.

A key benefit of (meta)model composition is its ability to add new capabilities while simultaneously leveraging prior investments in existing tool-chains, including domain constraints and generators of existing DSMLs. A combination of DSMLs and DSML composition technologies can help address the challenges associated with functional integration of component middleware technologies, without incurring the drawbacks of conventional approaches. Common drawbacks include (1) requiring expertise in all of the domains corresponding to each subsystem of the system being integrated, (2) writing more code in third-generation programming languages to integrate systems, (3) the lack of scalability of such an approach, and (4) the inflexibility in (re-)targeting integration code to more than one underlying middleware technology during system evolution.

This chapter describes the design and application of the *System Integration Modeling Language* (SIML) [5]. SIML is our open-source DSML that enables functional integration of component-based systems via the (meta)model composition mechanisms provided by the Generic Modeling Environment (GME) [67], which is an open-source meta-programmable modeling environment. The SIML composite DSML combines the following two existing DSMLs:

- The CCM profile of the *Platform-Independent Component Modeling Language* (PICML) [3], which supports the model-driven engineering of CCM-based systems,
- The *Web Services Modeling Language* (WSML), which supports model-driven engineering of web services-based systems.

Since SIML is a composite DSML, it has complete access to the semantics of PICML and WSML (sub-DSMLs), which simplifies and automates various tasks associated with integrating systems built using CCM and web services.

V.1 Functional Integration - A Case Study

To motivate the need for MDE-based functional integration capabilities, this section describes an enterprise distributed system case study from the domain of shipboard computing environments [48], focusing on its functional integration challenges. The shipboard computing environment that forms the basis for our case study was originally developed using one component middleware technology: OMG CCM implemented using the CIAO middleware [55]. It was later enhanced to integrate with components written using another middleware technology: W3C web services implemented using Microsoft's .NET web services.

V.1.1 Shipboard Enterprise Distributed System Architecture

The enterprise distributed system in our case study consists of the components shown in Figure V.1 and outlined below:

- **Gateway component**, which provides the user interface and main point of entry into the system for operators,
- **Naming Service components**, which are repositories that hold locations of services available within the system,
- **Identity Manager components**, which are responsible for user authentication and authorization,
- **Business logic components**, which are responsible for implementing business logic, such as determining the route to be taken as part of ship navigation, tracking the work allocation schedule for sailors,

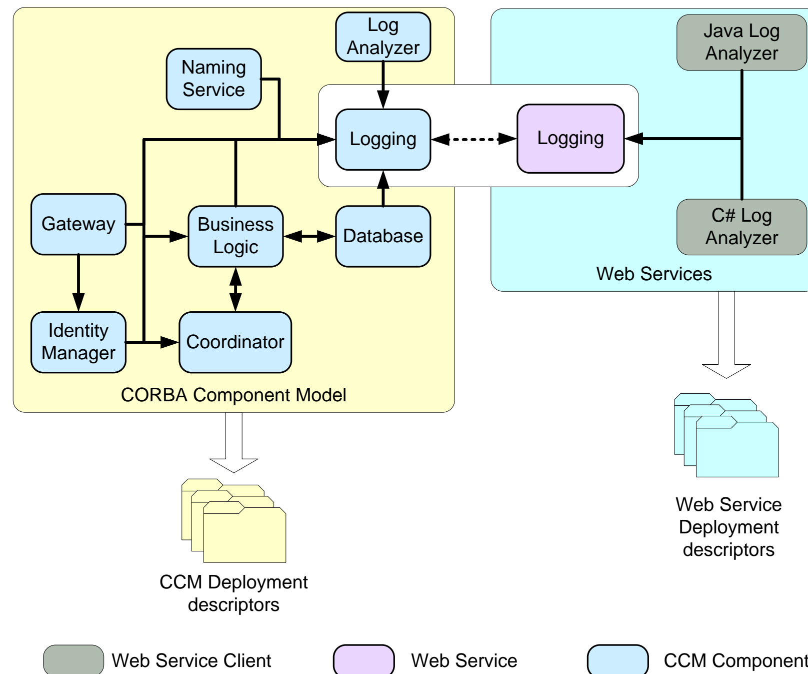


Figure V.1: Enterprise Distributed System Architecture

- **Database components**, which are responsible for database transactions,
- **Coordinator components**, which act as proxies for business logic components and interact with clients,
- **Logging components**, which are responsible for collecting log messages sent by other components,
- **Log Analyzer components**, which analyze logs collected by Logging components and display results.

Clients that use these component services first connect to a Naming Service to obtain the Gateway's location. They then request services offered by the system, passing their authentication/authorization credentials to a Gateway component, which initiates the series of interactions shown in Figure V.1. The system provides differentiated services depending on the credentials supplied by clients. Areas where services can be differentiated

between various clients include the maximum number of simultaneous connections, maximum amount of bandwidth allocated, and maximum number of requests processed in a particular time period.

To track the performance of the system—and the QoS the system offers to different clients—application developers originally wrote Log Analyzer components to obtain information by analyzing the logs. Based on changes in the COTS technology base and user requirements, a decision was made to expose a web service API to Logging components so that clients could also track the QoS provided by the system to their requests by accessing information available in Logging components. Since the original system was written using CCM, this change request introduced a new requirement to integrate systems that were not designed to work together, *i.e.*, CCM-based Logging components with the Web Service clients.

The flow of control—and the number and functionality of the different participants—in this case study is representative of enterprise distributed systems that require authentication and authorization from clients, and provide differentiated services to clients, based on the credentials offered by the client. In this chapter, we examine this system from an *integration* perspective, *i.e.*, how can this system, which initially had a homogeneous, stand-alone design, be integrated with other middleware. Note that this chapter is *not* studying the system from the perspective of system functionality or the QoS provided by the Business Logic components.

V.1.2 Functional Integration Challenges

Functional integration of systems is challenging and involves activities that map between various levels of abstraction in the integration lifecycle, including design, implementation, and use of tools. We describe some of the key challenges associated with integrating older component middleware technologies, such as CCM and EJB, with newer middleware

technologies, such as web services, and relate them to our experiences developing the shipboard computing case study described in Section V.1.1. The following list of challenges is by no means complete, *i.e.*, we focus on challenges addressed by our approach.

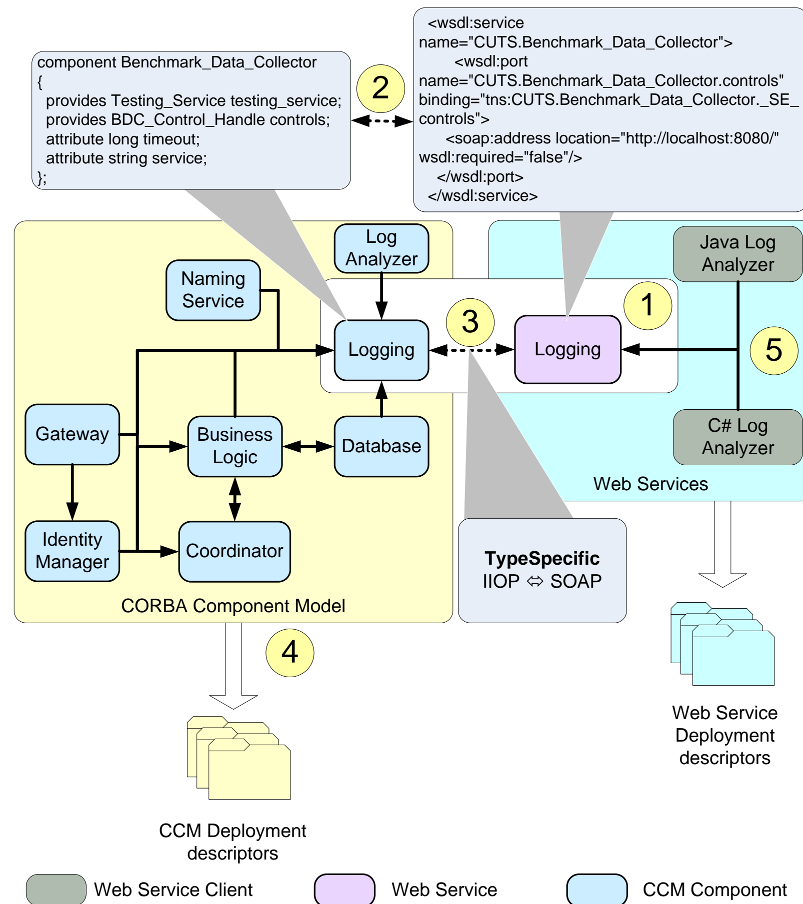


Figure V.2: Functional Integration Challenges

Challenge 1: Choosing an appropriate level of integration. As shown in Step 1 of Figure V.2, a key activity is to identify the correct level of abstraction at which functional integration of systems should occur, which involves selecting elements from different technologies being integrated that can serve as conduits for exchanging information. Within the different possible levels at which integration can be performed, the criteria for determining the appropriate level of integration include:

- The number of normalizations (*i.e.*, the conversion to/from the native types) required to ensure communication between peer entities being integrated,
- The number (and hence the overhead) and the flexibility of deployment (*e.g.*, in-process vs. out-of-process) of run-time entities required to support functional integration,
- The number of required changes to the integration architecture corresponding to changes in the peers being integrated, and
- Available choices of platform-specific infrastructure (*e.g.*, operating systems, programming languages) associated with performing integration at a particular level.

Attempting integration at the wrong level of abstraction can yield brittle integration architectures. For instance, the portions of the system implementing the integration might require frequent changes in response to changes in either the source or the target system being integrated.

In our shipboard computing case study example, we need to integrate Logging components so that web service clients can access their services. The programming model of CCM prescribes component *ports* as the primary component interconnection mechanism. Web Services also define ports as the primary interconnection mechanism between a web service and its clients. During functional integration of CCM with web services, a mapping between CCM component ports and web services ports offers an appropriate level of abstraction for integration. Although mapping CCM and web services ports is relatively straightforward, determining the right level of abstraction to integrate *arbitrary* middleware technologies can be much harder since a natural mapping between technologies may not always exist. In general, it is hard for system integrators to decide the right level of abstraction, and requires expertise in all the technologies being integrated.

Challenge 2: Reconciling differences in interface specifications. After the level of abstraction to perform functional integration is determined, it is necessary to map the interfaces exposed by elements of the different technologies as shown in Step 2 of Figure V.2. COTS middleware technologies usually have an interface definition mechanism that is separate from the component/service implementation details, *e.g.*, CCM uses the OMG Interface Definition Language (IDL), whereas web services use W3C Web Services Definition Language (WSDL). Older technologies (such as COBOL or C) may not offer as clear a separation of interfaces from implementations, so the interface definition itself may be tangled. Irrespective of the mechanism used to define interfaces, mapping interfaces between any two technologies involves at least three tasks:

- **Datatype mapping**, which involves mapping a datatype (both pre-defined and complex types) from source to target technology.
- **Exception mapping**, which involves mapping exceptions from source to target technology. Exceptions are defined separately from datatypes since the source or target technologies may not support exceptions (*e.g.*, Microsoft's COM uses a *HRESULT* to convey errors instead of using native C++ exceptions).
- **Language mapping**, which involves mapping datatypes between two technologies while accounting for differences in languages at the same time. Functional integration is limited when attempting this mapping, which is often done via inter-process communication at runtime to work around limitations in hosting these technologies "in-process," *i.e.*, within the same process.

In our shipboard computing case study example, Logging components handle CORBA datatypes, (which offer a limited subset of datatypes) whereas web service clients exchange XML datatypes (which provide a virtually unlimited set of datatypes due to XML's flexibility). Similarly, Logging components throw CORBA exceptions with specific minor/major codes containing specific fault and retry semantics. In contrast, web service clients must

convert these exceptions to SOAP "faults," which have a smaller set of exception codes and associated fault semantics. Performing these mappings is non-trivial, requires expertise in both the source and target technologies, and can incur scalability problems due to tedium and error-proneness if not automated.

Challenge 3: Managing differences in implementation technologies. The interface mapping described above addresses the high-level details of how information is exchanged between different technologies being integrated. As shown in Step 3 of Figure V.2, however, low-level technology details (such as networking, authentication, and authorization) are responsible for *delivering* such integration, *i.e.*, make it possible for the actual exchange of information between the different technologies being integrated. This adaptation involves a technology mapping and includes the following activities:

- **Protocol mapping**, which reconciles the differences between the protocols used for communication between the two technologies. For example, the Internet Inter-ORB Protocol (IIOP) binary protocol is used for communication in CCM, whereas the SOAP XML-based text protocol is used in web services.

- **Discovery mapping**, which allows bootstrapping and discovery of components/-services between source and target technologies. For example, CCM uses the CORBA Naming Service and CORBA Trading Service, whereas web services use Universal Description Discovery and Integration (UDDI) to discover other web services.

- **Quality of Service (QoS) mapping**, which maps QoS mechanisms between source and target technologies to ensure that service-level agreements are maintained. For example, CCM uses the CORBA Notification Service to enable anonymous publish/subscribe communication between components, whereas Web Services use WS-Notification and WS-Eventing to handle event-based communication between services.

In our shipboard computing case study example, Logging components only understand IIOP. Unfortunately, IIOP is not directly interoperable with the SOAP protocol understood by web service clients. To communicate with Logging components, therefore, requests must be converted from SOAP to IIOP and vice-versa.

There are also differences between how components and services are accessed. For example, the Logging component is exposed to CCM clients as a CORBA Object Reference registered with a Naming Service. In contrast, a web service client typically expects a Uniform Resource Identifier (URI) registered with a UDDI service to indicate where it can obtain a service. Converting from an Object Reference to a URI is non-trivial and the URI must be synchronized if Logging components are redeployed to different hosts.

In general, mapping of protocol, discovery, and QoS technology details not only requires expertise in the source/target technologies, it also requires intimate knowledge of the implementation details of these technologies. For example, developers familiar with CCM may not understand the intricacies of IIOP, which is usually known to only a handful of middleware technology developers, rather than application developers. This expertise is needed because issues like QoS are so-called non-functional properties, which require input from domain and platform experts, in addition to application developers.

Challenge 4: Managing deployment of subsystems. Component middleware technologies use declarative notations (such as XML descriptors, source-code attributes, and annotations) to capture various configuration options. Example metadata include EJB deployment descriptors, .NET assembly manifests, and CCM deployment descriptors. This metadata describes configuration options on interfaces and interconnections between these interfaces, as well as implementation entities, such as shared libraries and executables.

As shown in Step 4 of Figure V.2, system integrators must track and configure metadata correctly during integration and deployment. In many cases, the correct functionality of the integrated system depends on correct configuration of the metadata. Moreover, the development-time default values for such metadata are often different from the values at

integration- and deployment-time. For instance, configuration of web servers that are exposed to external clients are typically stricter, *e.g.*, they impose various limits on resource usage like connection timeouts, interfaces listened on, maximum number of simultaneous connections from a single client to prevent denial-of-service attacks, compared to the ones that developers use when creating web applications.

In our shipboard computing case study example, Logging components are associated with CCM descriptors needed to configure their functionality, deployed using the CIAO CCM deployment infrastructure, and run on a dedicated network testbed. If web service clients need to access functionality exposed by Logging components, however, certain services (such as a Web Server to host the service and a firewall) must be configured. This coupling between the deployment information of Logging components and the services exposed to Web Service clients means that changes to the Logging component necessitates corresponding changes to Logging web service. Failure to keep these elements synchronized can result in loss of service to clients of one or both technologies.

Challenge 5: Dealing with interoperability issues. Unless a middleware technology has only one version implemented by one provider (which is unusual), there may be multiple implementations from different providers. As shown in Step 5 of Figure V.2, differences between these implementations will likely arise due to non-conformant extension to standards, different interpretations of the same (often vague) specification, or implementation bugs. Regardless of the reasons for incompatibility, however, problems arise that often manifest themselves only during system integration. Examples of such differences are highlighted by efforts like the Web Services-Interoperability Basic Profile (WS-I) [7] standard, which is aimed at ensuring compatibility between web services implementations from different vendors.

In our shipboard computing case study example, not only must Logging components expose their services in WSDL format, they must also ensure that web service clients developed using different web services implementations (*e.g.*, Microsoft .NET or Java) are

equally capably of accessing their services. Logging components therefore need to expose their services using an interoperable subset of WSDL defined by WS-I, so clients are not affected by incompatibilities, such as using SOAP RPC encoding.

Due to the five challenges described in this section, significant integration effort is spent on configuration activities, such as modifying deployment descriptors and configuring web servers to ensure that system runs correctly. Significant time is also spent on interoperability activities, such as developing and configuring protocol adapters to link different systems together. Depending on the number of technologies being integrated, this activity does not scale up due to the number of adaptations required and the complexity of the adapter configuration. For example, it took several weeks to develop and configure the gateway (needed to bridge the communication between CCM and web services) described in Section V.3.2.

In general, problems discovered at the integration stage often require implementation changes, thereby necessitating interactions between developers and integrators. These interactions are often inconvenient, and even infeasible (especially when using COTS products), and can significantly complicate integration efforts. The remainder of this chapter shows how our GME-based (meta)model composition framework and associated tools help address these challenges.

V.2 DSML Composition using GME

This section describes the (meta)model composition framework in the Generic Modeling Environment (GME) [67]. DSMLs are defined by metamodels, hence, DSML composition is defined by (meta)model composition. The specification of how metamodels should be composed (*i.e.*, what concepts in the metamodels that are composed relate to each other and how) can be specified via association relationships and additional composition operators, as described in [57]. GME provides the following operators that assist in composition:

- The *equivalence* operator defines a full union between two metamodel components.

The two are no longer separate, but instead form a single concept. This union includes all attributes and associations, including generalization, specialization, and containment, of each individual component.

- The *interface inheritance* operator does not support attribute inheritance, but does allow full association inheritance, with one exception: containment associations where the parent functions as the container are not inherited. In other words, the child inherits its parent's external interface, but not its internal structure.

- The *implementation inheritance* operator makes the child inherit all of the parent's attributes, but only the containment associations where the parent functions as the container. No other associations are inherited. In other words, the child inherits the parent's internal structure, but not its external interface. The union of interface and implementation inheritance is the normal inheritance operator of the GME metamodeling language, and their intersection is null.

Together, these three operators allow for a semantically rich composition of metamodels.

A key property of a composite DSML is that it supports the *open-closed* principle [78], which states that a class should be open for extension but closed with respect to its public interface. In GME, elements of the sub-DSMLs are *closed*, *i.e.*, their semantics cannot be altered in the composite DSML. The composite DSML itself, however, is *open*, *i.e.*, it allows the definition of new interactions and the creation of new derived elements. All tools that are built for each sub-DSML work without any modifications in the composite DSML and all the models built in the sub-DSMLs are also usable in the composite DSML.

We use the following GME (meta)model composition features to support the SIML-based integration of systems built using different middleware technologies, as described in Section V.3.1:

1. **Representation of independent concepts.** To enable complete reuse of models and

tools of the sub-DSMLs, the composition must be done in such a way that all concepts defined in the sub-DSMLs are preserved. Step 1 of Figure V.3 shows how no elements from either sub-DSMLs should be merged together in the composite DSML. GME's composition operators [68] can be used to create new elements in

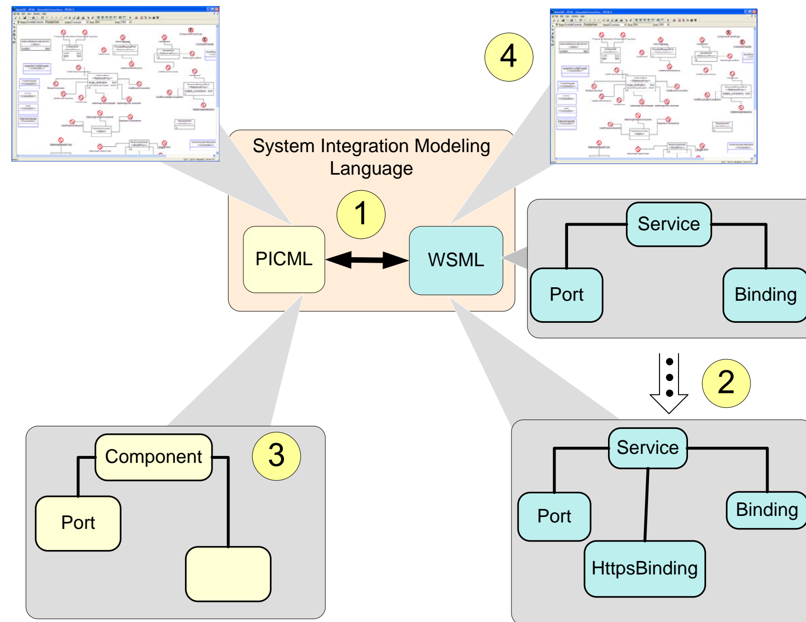


Figure V.3: Domain-Specific Modeling Language Composition in GME

the composite DSML, but the sub-DSMLs as a whole must remain untouched. As a consequence, any model in a sub-DSML can be imported into the composite language, and vice versa. All models in the composite language that are using concepts from the sub-DSMLs can be imported back into the sub-DSML. Existing tools for sub-DSMLs can be reused as well in the composite environment. This technique of composing DSMLs is referred to as *metamodel interfacing* [34] since we create new elements and relationships that provide the interface between the sub-DSMLs.

2. **Supporting (meta)model evolution.** DSML composition enables reuse of previously defined (sub-)DSMLs. Just like code reuse in software development, (meta)model reuse can benefit from the concept of *libraries*, which are read-only projects imported

to a host project. GME libraries ensure that if an existing (meta)model is used to create a new composite (meta)model, any changes or upgrades to the original will propagate to the places where they are used. Step 2 of Figure V.3 shows how if the original (meta)model is imported as a library, GME provides seamless support to update it when new versions become available (libraries are supported in any DSML with GME, not just the metamodeling language).

Components in a host project can create references to—and derivations of—library components. The library import process creates a copy of the reused project, so subsequent modifications to the original project are not updated automatically. To update a library inside a host project, a user-initiated refresh operation is required. To achieve unambiguous synchronization, elements inside a project have unique ids, which ensures correct restoration of all relationships that are established among host project components and the library elements.

3. **Partitioning (meta)model namespaces.** When two or more (meta)models are composed, name clashes may occur. To alleviate this problem, (meta)model libraries (and hence the corresponding components DSMLs) can have their own namespaces specified by (meta)modelers, as shown in Step 3 of Figure V.3. External software components, such as code generators or model analysis tools that were developed for the composite DSML, must use the fully qualified names. But tools that were developed for component DSMLs will still work because GME sets the context correctly before invoking such a component.

4. **Handling constraints.** The syntactic definitions of a metamodel in GME can be augmented by static semantics specifications in the form of Object Constraint Language (OCL) [146] constraint expressions. When metamodels are composed together, the predefined OCL expressions coming from a sub-DSML should not be

altered. GME's Constraint Manager uses namespace specifications to avoid any possible ambiguities, and these expressions are evaluated by the Constraint Manager with the correct types and priorities as defined by the sub-DSML, as shown in Step 4 of Figure V.3. The composite DSML can also define new OCL expressions to specify the static semantics that augment the specifications originating in the metamodels of the sub-DSMLs.

V.3 Integrating Systems with SIML

This section describes how we created and applied the *System Integration Modeling Language* (SIML) to solve the challenges associated with functional integration of systems in the context of the shipboard computing scenario described in Section V.1.1. SIML is our open-source composite DSML that simplifies functional integration of component-based systems built using heterogeneous middleware technologies. First, we describe how SIML applies GME's (meta)model composition features described in Section V.2 to compose DSMLs built for CCM and web services. We then describe how the challenges described in Section V.1.2 are resolved using features in SIML.

V.3.1 The Design and Functionality of SIML

Applying GME's (meta)model composition features to SIML. To support integration of systems built using different middleware technologies, SIML uses the GME (meta)model composition features described in Section V.2 as shown in Figure V.4. SIML is thus a composite DSML that allows integration of systems by composing multiple DSMLs, each representing a different middleware technology. Each sub-DSML is responsible for managing the metadata (creation, as well as generation) of the middleware technology it represents.

The composite DSML produced using SIML defines the semantics of the integration, which might include reconciling differences between the diverse technologies, as well as

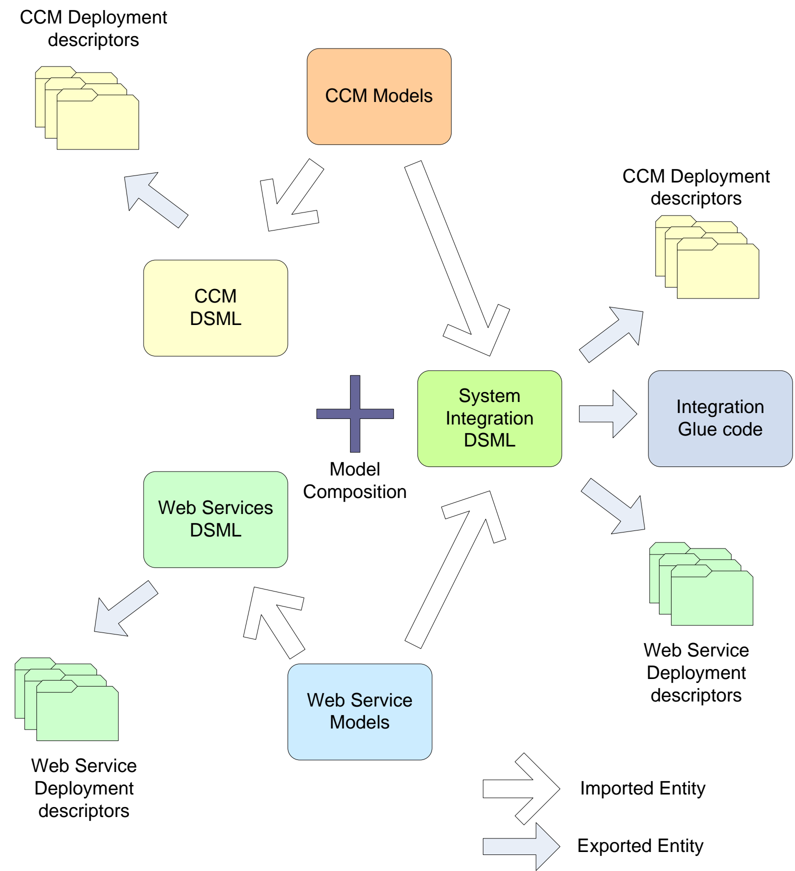


Figure V.4: Design of System Integration Modeling Language (SIML) Using Model Composition

representing characteristics of various implementations. The result is a single composite DSML that retains all the characteristics of its sub-DSMLs, yet also unifies them by defining new interactions between elements present in both DSMLs. System integrators therefore have a single MDE environment that allows the creation and specification of elements in each sub-DSML, as well as interconnecting them as if they were elements of a single domain.

For example, SIML is designed to support composite DSMLs that could represent different resource adaptations required to connect an EJB component with a web service. Likewise, it could be used to represent the differences between implementation of web services in the Microsoft .NET framework or an implementation in IBM's WebSphere.

The problems with functional integration of systems outlined in Section V.1.2 can be resolved by generating metadata directly from the composite DSML since the tools of the sub-DSMLs work seamlessly in the composite.

Applying SIML to compose CCM and web services. Our initial use of SIML was to help integrate CCM with web services in the context of the shipboard computing case study described in Section V.1. The two sub-DSMLs we needed to integrate to support the new requirements for this case study were:

- The **Platform-Independent Component Modeling Language** (PICML) [3], which enables developers of CCM-based systems to define application interfaces, QoS parameters, and system software building rules described in Chapter III. PICML can also generate valid XML descriptor files that enable automated system deployment.
- The **Web Services Modeling Language** (WSML), which supports key activities in web service development, such as creating a model of a web services from existing WSDL files, specifying details of a web service including defining new bindings, and auto-generating artifacts required for web service deployment.

These two sub-DSMLs were developed independently for earlier projects. The case study described in Section V.1.1 provided the motivation to integrate them together using GME's (meta)model composition framework.

Since SIML is a composite DSML, all valid elements and interactions from both PICML and WSML are valid in SIML. It is possible to design both CCM components (and assemblies of components), as well as web services (and federations of web services) using SIML, just as if either PICML or WSML were used independently. The whole is greater than the sum of its parts, however, because SIML defines new interactions that allow connecting a CCM component (or assembly) with a web service and automates generation of necessary gateways, which are capabilities that exist in neither PICML nor WSML.

V.3.2 Resolving Functional Integration Challenges using SIML

We now show how we applied SIML to resolve the functional integration challenges discussed in Section V.1.2 in the context of our shipboard computing case study example described in Section V.1.1. Although we focus on the current version of SIML that supports integration of CCM and web services, its design is sufficiently general that it can be applied to integrate other middleware technologies (such as EJB) without undue effort. Figure V.5

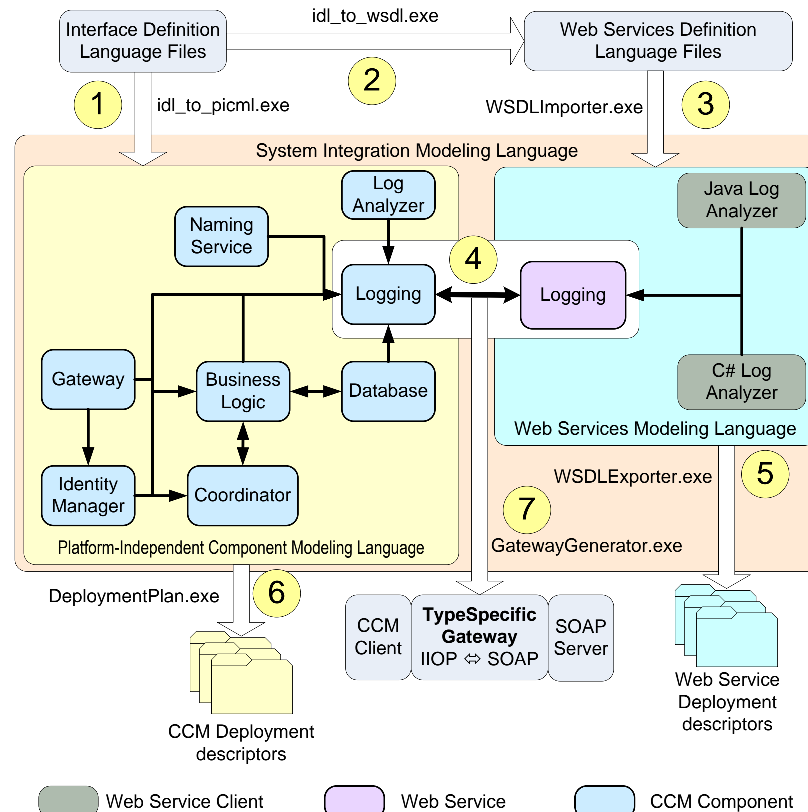


Figure V.5: Generating a Web Service Gateway Using SIML

shows how SIML resolves the following challenges to generate a gateway given an existing CCM application:

Resolving challenge 1: Choosing an appropriate level of integration. As mentioned in Section V.1.2, determining the correct level of integration requires expertise in all different technologies being integrated. To allow interactions between CCM components and web

services, SIML defines interactions between ports of CCM components and ports exposed by the web services. SIML also automates the generation of the glue code, so some choices with respect to the level of integration (*e.g.*, mapping of a CCM port to a web service port) are pre-determined, while other decisions, (*e.g.*, aggregation of more than one CCM component into a single web service) are customizable.

SIML extends the list of valid interactions of both CCM components and web services, which is an example of a composite DSML defining interactions that do not exist in its sub-DSMLs. SIML can also partition a large system into hierarchies via the concept of "modules," which can be either CCM components (and assemblies of CCM components) or web services.

In our shipboard computing case study example, a user of SIML needs to import the IDL files describing the shipboard system, as show in step 1 of Figure V.5. Similarly, WSDL files can also be imported into SIML, as shown in step 3 of Figure V.5. After the interfaces of the systems have been imported, users can define the interactions between the subsystems, *i.e.*, the interaction between the CCM and Web Service logging capabilities can be defined by connecting the ports of the CCM Logging Component to the ports of the Logging web service.

The interactions between subsystems automate a number of activities that arise during integration, including generation of resource adapters, such as the gateways shown in step 7 of Figure V.5 and described in the forthcoming resolution of Challenge 3. SIML provides a middleware technology-specific integration framework that allows system integrators to define the points of interaction in their system. SIML allows the system integration to be done in a "declarative" fashion, *i.e.*, the system integrators specify the points of integration at a high-level using connections between model elements. Using this information, SIML takes care of the translation of the integrator's intent (policy) into the low-level mechanics

needed to achieve the integration. The approach taken by SIML, is different from an "imperative" approach to integration, where the system integrator needs to specify not only the high-level integration design, but also the low-level details of the integration.

SIML's architecture can be enhanced to support integration of other middleware technologies by extending the list of interactions defined by SIML to integrate new technologies. For example, SIML could be extended to support interactions between CCM and EJB or between Web Services and EJB. Extending SIML to support EJB requires specification of a DSML that describes the elements and interactions of EJB. Once the DSML for EJB is specified, it can be imported into SIML as a library while also assigning a new namespace to it. The creation of a new namespace prevents any clashes between the type systems, *e.g.*, between a CCM component and EJB component. Interactions between elements of CCM and EJB can then be defined in the composite DSML. From these new interactions, generative techniques (as explained in resolution to Challenge 3) can be applied to automate the integration tasks.

Resolving challenge 2: Reconciling differences in interface specifications. To map interfaces between CCM and web services, SIML provides a tool called IDL2WSDL that automatically converts any valid CORBA IDL file to a corresponding WSDL file. As part of this conversion process, IDL2WSDL performs (1) *datatype mapping*, which maps CORBA datatypes to WSDL datatypes and (2) *exception mapping*, which maps both CORBA exceptions to WSDL faults. IDL2WSDL relieves system integrators from handling the intricacies of this mapping.

Figure V.5 shows how both IDL and WSDL can be imported into the DSML environment corresponding to CCM (PICML) and web services (WSML). This capability allows integrators to define interactions between CCM components and web services as shown in step 4 of Figure V.5. SIML also supports *language mapping* between ISO/ANSI C++ and Microsoft C++/CLI, which is the .NET framework extension to C++.

In our example scenario, IDL2WSDL can automatically generate the WSDL files of

the Logging web service from the IDL files of the Logging Component as shown in step 2 of Figure V.5. The generated WSDL file can then be imported into SIML, and annotated with information used during deployment. As shown in step 5 of Figure V.5, SIML can also generate a WSDL file back from the model, so that WSDL stubs and skeletons can be generated. SIML automates much of the tedious and error-prone details of mapping IDL to WSDL, thereby allowing system integrators to focus on the business logic of the application being integrated.

Resolving challenge 3: Managing differences in implementation technologies. The rules defined in SIML allow definition of interaction at the modeling level. This feature, however, is not useful if these definitions cannot be translated into runtime entities that actually perform the interactions. SIML therefore generates *resource adapters*, which automatically bridge the differences between protocol formats by performing the necessary conversions of one format into another, such as converting SOAP requests into IIOP requests, and vice-versa.

A *resource adapter* in SIML is implemented as a gateway. A gateway sits between web service clients and encapsulates access to the CCM component by exposing it as a web service. SIML allows system integrators to define connections between ports of a CCM component and a web service, as shown in step 4 of Figure V.5. These connections are then used by a SIML *model interpreter*, which automatically determines the operation/method signatures of operations/methods of the ports on either end of a connection, and uses this information to generate a gateway automatically. As shown in step 7 of Figure V.5, the generated gateway is composed of three entities:

1. A **CCM client**, which uses the stubs (client-side proxies) generated from the IDL files and handles the communication with other CCM components using IIOP as the protocol,

2. A **SOAP server**, which uses the skeletons (server-side proxies) generated from the

WSDL files and handles the communication with web service clients using SOAP as the protocol,

3. A **IIOP-to-SOAP translator**, which operates at the level of programming language (as opposed to the on-wire-protocol level) handling the delegation of web service requests to the CCM client component as well as dealing with the conversion of replies from the CCM client into a web service reply to be sent back to the Web Service clients.

The generated gateway encapsulates the resource adapter, which contains all the "glue code" necessary to perform datatype mapping, exception mapping, and language mapping between CCM and web services. SIML's gateway generator is configurable and can currently generate web service gateways for two different implementation of web services: gSOAP [140] and Microsoft ASP.NET. The generated gateway also performs the necessary *protocol mapping* (*i.e.*, between IIOP and SOAP) and *discovery mapping* (*i.e.*, automatically connecting to a Naming Service to obtain object references to CCM components). Our initial implementation does not yet support *QoS mapping*, which is the focus of future work, as described in Chapter VII.

In our shipboard computing case study example, SIML can automatically generate the Logging web service gateway conforming to GSOAP and/or Microsoft ASP.NET, by running the SIML model interpreter as shown in step 7 of Figure V.5. Auto-generation of gateways eliminates the tedious and error-prone programming effort that would have otherwise been required to integrate CCM components with Web Services. In general, given a pair of technologies to integrate, auto-generation of gateways eliminates the need for both writing code required to perform the technology mapping, as well as the repetitive instantiation of such code for each of the interfaces that need to be integrated. Auto-generation also masks the details of the configuration of the technology-specific *resource adapter*s used in the integration.

Resolving challenge 4: Managing deployment of subsystems. After the necessary integration gateways have been generated, system integrators also need to deploy and configure the application and the middleware using metadata, *e.g.*, in the form of XML descriptors. Since SIML is built using (meta)model composition it can automatically use the tools developed for the sub-DSMLs directly from within SIML. For instance, PICML can handle deployment of CCM applications and WSML can handle deployment of web services.

In our shipboard computing case study example, SIML can be used to automatically generate the necessary deployment descriptors for all CCM components, as well as the Logging web service as shown in steps 5 and 6 of Figure V.5. SIML shields system integrators from low-level details of the formats of the different descriptors. It also shields them from manually keeping track of the number of such descriptors required to deploy a CCM component or a Web Service.

By encapsulating the required resource adapters inside a web service or CCM component, SIML allows reuse of deployment techniques available for both CCM and web services. System integrators need not deploy resource adapters separately. Although this approach works for in-process resource adapters (such as those generated by SIML), out-of-process resource adapters need support from a deployment descriptor generator. Since SIML is a DSML itself, this support could be added to SIML so it can generate deployment support for out-of-process resource adapters.

Resolving challenge 5. Dealing with interoperability issues. Since knowledge of the underlying middleware technologies is built into SIML, it can compensate for certain types of incompatibilities, such as differences in interface definition styles during design time. For example, IDL2WSDL allows generation of WSDL that supports an interoperable subset of WSDL as defined in the WS-I Basic Profile. System integrators are better prepared to avoid incompatibilities that would have traditionally arisen during integration testing.

SIML can also define constraints on WSDL definition as prescribed by the WS-I Basic

Profile, so that violations can also be checked at modeling time. Similarly, gateway generation can automatically add workarounds for particular implementation quirks, such as defining the correct set of values for XML namespaces of the interfaces defined in WSDL files depending upon the (observed) behavior of the target middleware implementation. System integrators are once again shielded from discovering these problems during final integration testing. In our shipboard computing case study example, SIML can generate a Logging web service gateway that either supports a WS-I subset or uses SOAP RPC encoding.

SIML's DSML composition-based approach to integrating systems relieves system integrators from developing more code during integration. The automation of gateway generation allows integration of systems that have a large number of components since developers need not write system specific integration code. In addition, SIML supports evolution of the integrated system by incrementally adding more components or by targeting different middleware implementations as future needs dictate.

V.3.3 Evaluating SIML

To evaluate the benefits of SIML, we first define a taxonomy for evaluating technologies that assist the functional integration of CCM and Web Services. We then use this taxonomy to compare SIML with tools that are supplied by vendors for either technology, referred to in Table V.1 as *Native tools*. Examples of native tools include the Microsoft Visual Studio and the IBM Eclipse suite, which developers using middleware technologies like .NET and EJB are likely to use. This table depicts the different mapping activities described in Section V.3.2 that are typical in functional integration of middleware systems. For each activity the table describes the level of support in SIML and whether the activity is automated. It also describes the level of automation measured as the number of distinct steps performed by a system integrator. Table V.1 further decomposes the level of automation into three broad categories: (1) *design*, which denotes that system integrators

Table V.1: Evaluating Functional Integration using SIML

Integration Activity	Supported?	Automated?	Level of Automation (# of distinct steps)					
			Using SIML			Using Native Tools		
			Design	Implementation	Tool Use	Design	Implementation	Tool Use
Integration Design	Yes	No	0	0	1	1	1	0
Interface Mapping								
DataType Mapping	Yes	Yes	0	0	1	1	1	0
Exception Mapping	Yes	Yes	0	0	1	1	1	0
Language Mapping	Yes	Yes	0	0	1	1	1	0
Technology Mapping								
Protocol Mapping	Yes	Yes	0	0	1	1	1	0
Discovery Mapping	Yes	Yes	0	0	1	1	1	0
QoS Mapping	No	No	1	1	0	1	1	0
Deployment Mapping								
Descriptor Generation	Yes	Yes	0	0	1	0	0	1
Gateway Placement	No	No	1	1	0	1	1	0
Interoperability Mapping	Yes	Yes	0	0	1	0	1	0

need to perform a design activity that might include domain analysis, requirement analysis, etc., (2) *implementation*, which denotes that system integrators need to implement some functionality usually by writing code, and (3) *tool use*, which denotes that a tool needs to be used by the system integrators to perform that activity. This categorization assigns a weight commensurate to the skills of the individual responsible for carrying out the task in a particular organization.

Our taxonomy also assumes that design and implementation are orders of magnitude more difficult/time-consuming than tool use. In Table V.1, multiple activities of the same category are considered equal, since the magnitude difference will likely dwarf any small number of steps of any particular category. Table V.1 uses 1 to indicate one or more, *i.e.*, $1 \ldots n$ steps, and 0 to indicate that the effort is automated. To estimate the amount of effort required, we sum up each of the three columns (*i.e.*, design, implementation, and tool) and then multiply the result by the weight assigned to each category. For example, a reasonable assignment of weight for these activities might be 10, 5 and 1, for each of design, implementation and tool use. With this assignment, we can see that using SIML requires $2 \times 10 + 2 \times 5 + 8 \times 1 = 38$ distinct steps to achieve functional integration. In comparison, using just the native tools would result in $8 \times 10 + 9 \times 5 + 1 \times 1 = 126$ distinct steps to achieve the same. It should be noted that the number of steps will get reduced drastically as (and when) native tools add support for integration activities.

The numbers in Table V.1 are for each unique unit of work per unique pair of source and target technologies, *i.e.*, for a single datatype mapping, a single exception mapping, a single protocol mapping. To calculate the total cost of integration, we must take into account both the number of distinct types/exceptions/languages, and the number of unique pairs of technologies being integrated.

Since SIML allows hierarchical composition of the integration infrastructure, the integration architecture scales along with the increase in the number of components. Although the generative techniques applied to generate the gateways scale with the number

of components in the system, when the number of components increases to thousands of components, the limitations of visual design tools tend to show up. To overcome the issues with scalability of modeling techniques, we have applied techniques like aspect-oriented weaving of domain-specific models [4] in prior efforts. Such techniques can be applied to automate the modeling activities in SIML in the presence of large number of components, since SIML itself is a domain-specific language for integration.

Table V.1 shows that SIML helps reduce the effort by reducing the design and/or implementation activities associated with integration to ordinary tool usage activities. For example, SIML effectively reduces the design and implementation effort required to perform the datatype, exception and language mapping, to a single step of tool use. This table also shows that similar gains can be achieved for complex tasks, such as protocol mapping (conversion between IIOP and SOAP in this case) and discovery mapping (conversion between CORBA Object References and Web Service URIs). Finally, the table reveals current gaps in our toolchain, *i.e.*, SIML does not perform QoS mapping or help with placement of resource adapters (or gateways), which remains as future work.

V.4 Summary

The development of enterprise distributed systems increasingly involves more integration of existing commercial-off-the-shelf (COTS) software and less in-house development from scratch. As the capabilities of COTS component middleware technologies grow, the complexity of integration of systems built upon such frameworks also grows. This chapter showed how a model-driven engineering (MDE) approach to functional integration of systems built using component middleware technologies enhances conventional tedious, error-prone, and non-scalable approaches to integration of enterprise distributed systems. In particular, we showed how domain-specific modeling language(DSML)s and (meta)model composition can help MDE tools address these limitations.

To demonstrate the viability of our approach, we enhanced support for composition

of DSMLs in the Generic Modeling Environment (GME). Using this new capability, we developed the *System Integration Modeling Language* (SIML), which is a DSML composed from two other DSMLs: the CORBA Component Model (CCM) profile of *Platform-Independent Component Modeling Language* (PICML) and the *Web Services Modeling Language* (WSML). We then demonstrated how composing DSMLs can solve functional integration problems in an enterprise distributed system case study by reverse engineering an existing CCM system and exposing it as web service(s) to web clients who use these services. Finally, we evaluated the benefits of our approach by generating a Logging component gateway from the model, which automates key steps needed to functionally integrate CCM components with web services.

Instructions for downloading and building the open-source SIML and GME MDE tools are available at `www.dre.vanderbilt.edu/cosmic`.

CHAPTER VI

COMPARISON WITH RELATED RESEARCH

Our research on MDE-based composition, optimization and integration techniques has resulted in improved tool-support for component middleware. This chapter compares and contrasts our work with respect to other alternate research approaches.

VI.1 Related Research: Composition Techniques

This section summarizes related efforts associated with developing DRE systems using an MDE approach and compares these efforts with our work on PICML.

Cadena Cadena [45] is an integrated environment developed at Kansas State University (KSU) for building and modeling component-based DRE systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Cadena also provides a component assembly framework for visualizing and developing components and their connections. Unlike PICML, however, Cadena does not support activities such as component packaging and generating deployment descriptors, component deployment planning, and hierarchical modeling of component assemblies.

VEST and AIRES The *Virginia Embedded Systems Toolkit* (VEST) [129] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [64] are MDD analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. Components are selected from pre-defined libraries, annotations for desired real-time properties are added, the resulting code is mapped to a hardware platform, and real-time and schedulability analysis is done. In contrast, PICML allows component modelers to model the complete functionality of components and intra-component interactions, and doesn't rely on predefined libraries. PICML also allows DRE

system developers the flexibility in defining the target platform, and is not restricted to just processors.

ESML The *Embedded Systems Modeling Language* (ESML) [58] was developed at the Institute for Software Integrated Systems (ISIS) to provide a visual metamodeling language based on GME that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. The user-created models can be fed to analysis tools (such as Cadena and AIRES) to perform schedulability and event analysis. Using these analyses, design decisions such as component allocations to the target execution platform are performed. Unlike PICML, ESML is platform-specific since it is heavily tailored to the Boeing Boldstroke PRiSm QoS-enabled component model [113, 124]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Boldstroke component model. ESML was the inspiration, however, for a number of features in PICML.

Ptolemy II Ptolemy II [16] is a tool-suite from the University of California Berkeley (UCB) that supports heterogeneous modeling, simulation, and design of concurrent systems using an actor-oriented design. Actors are similar to components, but their interactions are controlled by the semantics of models of computation, such as discrete systems. The set of available actors is limited to the domains that are natively defined in Ptolemy. Using an actor specialization framework, code is generated for embedded systems. In contrast, PICML does not define a particular model of computation. Also, since PICML is based on the metamodeling framework of GME, it can be customized to support a broader range of domains than those supported by Ptolemy II. Finally, PICML targets component middleware for DRE systems and can be used with any middleware technology, as well as any programming language, whereas Ptolemy II is based on Java, with preliminary support for C.

VI.2 Related Research: Optimization Techniques

In this section, we compare our deployment-time optimizations to other component middleware optimization techniques. As described in Section II.2.1, optimization techniques to improve application performance can be categorized along two dimensions: (1) the layer at which the optimization is applied, *e.g.*, whether the optimization is restricted to the middleware layer alone or spans multiple layers, including applications above the middleware, (2) the time at which such optimization techniques are applied, *i.e.*, design/-development-time, run-time or deployment-time. Our research in PAM falls into the application layer and is done at deployment-time.

Design/development-time approaches. Design-time approaches to component middleware optimization include eliminating the dynamic loading of component implementation shared libraries and establishing connections between components done at run-time, as described in static configuration of CIAO [131]. Our PAM approach is different since it uses models of applications to modify the structure of the assembly by creating physical assemblies, *i.e.*, new components, at deployment time. Our approach is not restricted to optimizing just the inter-connections between components. Moreover, the static configuration approach can be applied in combination to our deployment-time optimizations.

Another approach to optimizing the middleware at design/development-time employs context-specific middleware specializations for product-line architectures [65], which exploits "invariant properties"— application, middleware and platform-level properties that remain fixed during system execution — to reduce the overhead caused by excessive generality in middleware frameworks. Researchers have also employed Aspect-Oriented Programming (AOP) techniques to automatically derive subsets of middleware based on use-case requirements [50] and modify applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [101].

In addition, middleware has been synthesized in a "just-in-time" fashion by integrating source code analysis, and inferring features and synthesizing implementations [151]. The

key difference between our approach in PAM and the various context-specific specializations and AOP-based techniques is that the optimizations performed by PAM do not require any input from the application developer, *i.e.*, the application developer need not design his application tuned for a specific deployment scenario. Our approach in PAM is, however, complementary to these approaches, since not all optimizations done via modification of application advocated by these approaches are possible to perform at deployment-time using PAM.

Run-time approaches. Research on approaches to optimizing middleware at run-time has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context [31]. QuO [153] is a dynamic QoS framework that allows dynamic adaptation of desired behavior specified in *contracts*, selected using proxy objects called *delegates* with inputs from run-time monitoring of resources by *system condition* objects. QuO has been integrated into component middleware technologies, such as CIAO/CCM [121].

Other aspects of run-time optimization of middleware include domain-specific middleware scheduling optimizations for DRE systems [42], using feedback control theory to affect server resource allocation in Internet servers [152] as well as to perform real-time scheduling in Real-time CORBA middleware [70]. Our work in PAM is targeted at optimizing the middleware resources required to host composition of components in the presence of a large number of components, whereas the main focus of these efforts is to either build the middleware to satisfy certain performance guarantees, or effect adaptations via the middleware depending upon changing conditions at run-time. Our work in PAM is complementary to these approaches to application optimization.

Run-time approaches to application-specific optimizations have focused on data replication for edge services, *i.e.*, replicating servers at geographically distributed sites [41]. Significant performance improvements in the latency and availability has been achieved

by relaxing the consistency of data that is replicated at the edge servers using application-specific semantics. Other research on optimizing web services has focused on utilizing reflective techniques encapsulated in the request metadata [85] for dynamic negotiation of best communication mechanisms between any requester and provider of a service. Other research [126] on dynamic optimization approaches include improving algorithms for event ordering within component middleware by making use of application context information available in models.

The approaches outlined above optimize the middleware/on-the-wire protocol using knowledge of the computations performed by the application. Our work in PAM makes use of the application deployment information on each node of the target domain and is focused on optimizing the execution of the components at each end-system as opposed to optimizing the on-the-wire protocol.

Deployment-time approaches. Deployment-time optimizations research includes BluePencil [69], which is a framework for deployment-time optimization of web services. BluePencil focuses on optimizing the client-server binding selection using a set of rules stored in a policy repository and rewriting the application code to use the optimized binding. Although conceptually similar, our work in PAM differs from BluePencil because it uses models of application structure and application deployment to serve as the basis for the optimization infrastructure. In contrast, BluePencil uses approaches like *configuration discovery* that extract deployment information from configuration files present in individual component packages. By operating at the level of individual client-server combinations, the kind of global optimizations performed by PAM are non-trivial to perform in BluePencil. Another key difference is that PAM focuses on optimizing multiple dimensions, *e.g.*, footprint as well as performance, whereas BluePencil only uses optimized binding to affect performance. BluePencil relies on modification of the application source code (to rewrite the application code), while PAM is non-intrusive and does not require application source code modifications.

Finally, our approach of fusing multiple component into a single physical assembly shared among the components is similar to how web servers like Apache support creation of multiple virtual hosts [38] in a single instance of a web server. The resource savings obtained from running multiple virtual hosts on a single instance of a web server, are replicated by the resource savings obtained as a result of hosting multiple component implementations in a single physical assembly in our approach using PAM. Our optimizations in PAM, focus on increasing the scalability of applications built using components in a manner similar to the scalability benefits of virtual hosts.

VI.3 Related Research: Integration Techniques

This section surveys the technologies that provide the context of our work on system integration in the domain of enterprise distributed systems. We classify techniques and tools in the integration space according to the role played by the technique/tool in system integration.

Integration evaluation tools enable system integrators to specify the systems/technologies being integrated and evaluate the integration strategy and tools used to achieve integration. For example, IBM's WebSphere [54] supports modeling of integration activities and runs simulations of the data that is exchanged between the different participants to help predict the effects of the integration. System execution modeling [128] tools, such as CUTS [48], help developers conduct "what if" experiments to discover, measure, and rectify performance problems early in the lifecycle (*e.g.*, in the architecture and design phases), as opposed to the integration phase.

Although these tools help identify potential integration problems and evaluate the overall integration strategy, they do not replace the actual task of integration itself since these tools use simulation-/emulation-based abstractions of the actual systems. SIML's role is complementary to existing integration evaluation tools. In particular, after the integration

evaluation has been done using integration evaluation tools, SIML can be applied to design the integration and generate various artifacts required for integration, as discussed in Section V.3.1.

Integration design tools. OMG's UML profile for Enterprise Application Integration (EAI) [91] defines a Meta Object Facility (MOF) [93] for collaboration and activity modeling. MOF provides facilities for modeling the integration architecture, focusing on connectivity, composition and behavior. The EAI UML profile also defines a MOF-based standardized data format intended for use by different systems to exchange data during integration. Data exchange is achieved by defining an EAI application metamodel that handles interfaces and metamodels for programming languages (such as C, C++, PL/I, and COBOL) to aid the automation of transformation.

Although standardizing on MOF is a step in the right direction, in practice there are various problems, such as the lack of widespread support for MOF by various tools, and the differences between versions of XML Metadata Interchange (XMI) [97] support in tools. Existing integration design tools provide limited support for interface mapping by generating stubs and skeletons. Moreover, key activities like discovery mapping, and deployment mapping must still be programmed manually by system integrators. The primary difference between SIML and integration design tools is that SIML not only allows such integration design, but it also automates the generation of key integration artifacts, such as gateways. Gateways encapsulate the different adaptations required to bridge the differences in the underlying low-level mechanisms of heterogeneous middleware technologies like network protocols and service discovery, reducing the amount of effort required to develop and deploy the systems, as discussed in Section V.3.2.

Integration patterns TrowBridge *et al.* [139] provides guidance to system integrators in the form of best patterns and practices, with examples using a particular vendor's products. Hohpe [49] catalogs common integration patterns, with an emphasis on system integration via asynchronous messaging using different commercial products. These efforts do not

directly provide tools for integration, but instead provide pattern-based guidance to apply existing tools to achieve more effective integration. A future goal of SIML is to add support for modeling integration patterns so that users can design integration architectures using patterns. We also plan to enhance SIML's generative capabilities to incorporate integration pattern guidelines in gateway generation, as discussed in Section V.3.2.

Resource adapters are used during integration to transform data and services exposed by service producers to a form amenable to service consumers. Examples include *data transformation* (mapping from one schema to another), *protocol transformation* (mapping from one network protocol to another), or *interface adaptation* (which includes both data and protocol transformation). The goal of resource adapters is to provide integrated, reusable solutions to common transformation problems encountered in integrating systems built using different middleware technologies.

Existing standards (such as the Java Messaging Specification [132] and J2EE Connector Architecture Specification [82]) and tools (such as IBM's MQSeries [53]), however, approach the integration from a middleware and programming perspective, *i.e.*, system integrators must still handcraft the "glue" code that invokes the resource adapter frameworks to connect system components together. In contrast, SIML uses syntactic information present in the DSMLs to automate the required mapping/adaptation by generating the necessary "glue" code, as discussed in Section V.3.2. Moreover, SIML relies on user input only for tool use, as opposed to requiring writing code in a programming language to configure the resource adapters.

Integration frameworks. The semantic web and the Web Ontology Language (OWL) [22] have focused on the composition of services from unambiguous, formal descriptions of capabilities as exposed by services on the Web. Research on service composition has focused largely on automation and dynamism [107], integration on large-scale "system-of-systems," such as the GRID [37]. Other work has focused on optimizing service compositions such that they are "QoS-aware" [149]; in such "QoS-aware" compositions, a service

is composed from multiple other services taking into account the QoS requirements of clients. Since these automated composition techniques rely on unambiguous, formal representations of capabilities, system integrators must make their legacy systems available as web services. Likewise, system integrators need to provide formal mappings of system capabilities to integrate, which may not always be feasible.

SIML's approach to (meta)model composition, however, is not restricted to a single domain, though the semantics are bound at design time, as discussed in Section V.3.1. Although both approaches rely on metadata, SIML uses metadata to enhance the generative capabilities during integration. Automated composition techniques, in contrast, focus on extraction of *semantic knowledge* from metadata, which is then used as the basis for producing compositions that satisfy user requirements.

Integration quality analysis. As the integration process evolves, it is necessary to validate whether the results are satisfactory from functional and QoS perspectives. Research on QoS issues associated with integration has yielded languages and infrastructure for evaluating *service-level agreements*, which are contracts between service providers and consumers that define the obligations of the parties involved and specify what measures to take if service assurances are not satisfied. Examples include (1) the Web Service-Level Agreement language (WSLA) [71] framework, which defines an architecture to define service-level agreements using an XML Schema, and provides an infrastructure to monitor the conformance of the running system to the desired service-level agreement, (2) [100], which allows monitoring user-specific service level agreements within the WS-Agreement framework, and (3) Rule-Based Service Level Agreement [106], which is a formal multi-layer approach to describing and monitoring service level agreements. Other efforts have focused on defining processes for distributed continuous quality assurance [142] of integrated systems to identify the impact on performance during system evolution. Information from these analysis tools should be incorporated into future integration activities.

Although quality analysis tools can provide input to design-time integration activities,

they do not support automated feedback loops. In particular, they do not provide mechanisms to modify the integration design based on results of quality analysis. SIML, in contrast, is designed to model service-level agreements to allow their evaluation before and/or after integration, as discussed in Section V.3.1.

VI.4 Summary

In this chapter we provided a comparison of our work with other approaches in the research community, and showed how our research on MDE-based techniques for component-based development provide capabilities in composition, optimization and integration tool-chain. Our research on PICML, SIML and PAM provides novel contributions to address deficiencies in the tool-chain in certain areas while complementing existing research in others.

CHAPTER VII

CONCLUDING REMARKS

Component middleware is an emerging paradigm whose success is crucial to realizing the vision of Software Factories [43]. There are, however, significant gaps in the component middleware development, optimization and integration toolchain, which if left unresolved have the potential to negate the benefits of using component middleware. This dissertation applied Model-Driven Engineering (MDE) techniques to various facets of component-based DRE system development. Chapter III applied MDE techniques to create the *Platform-Independent Component Modeling Language* (PICML), a Domain-Specific Modeling Language (DSML) to design, develop and deploy component-based DRE systems; Chapter IV presented a new class of optimization techniques, "deployment-time" optimizations, to optimize the overhead in component middleware technologies, and demonstrated the efficacy of the techniques in the *Physical Assembly Mapper*(PAM), an extension to PICML; Chapter V presented a novel approach to functional integration of component-based DRE systems by applying (meta)model composition to create, the *System Integration Modeling Language* (SIML), a composite DSML for integration created out of PICML and *Web Services Modeling Language* (WSML); and Chapter VI compared our work on PICML, PAM and SIML with other research in the design, development, deployment and integration of component-based systems.

VII.1 Lessons Learned

We now summarize the lessons learned from our work on MDE-based composition, optimization and integration techniques for component-based DRE systems.

VII.1.1 Composition Techniques

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

1. **Component and platform modeling improves DRE systems reasoning** The results of applying PICML to build a variety of DRE systems in diverse DRE system domains including avionics mission computing, unmanned-air vehicle surveillance and shipboard computing by users, underscore the importance of the level of comprehension brought about by the high-level of abstraction in PICML relative to conventional component middleware approaches. In particular, PICML's DSML-based approach provides DRE system developers with a system-level view and brings focus to system-level design decisions in addition to low-level issues at the granularity of individual components.

2. **Early detection of errors improves productivity significantly** Most of the errors that PICML eliminates at design- and deployment-time are discovered only at run-time with conventional component development techniques, due to a combination of complexities in development of components, coupled with *out-of-band* specification (using XML) of component interconnections. This finding underscores the importance of PICML's MDE approach, which helps increase the effectiveness of applying QoS-enabled component middleware technologies to the DRE systems domain.

3. **Addressing *ad hoc* approaches of configuration** PICML alleviates key complexities in understanding the impact of middleware configurations on application QoS and brings rigor [60] to otherwise *ad hoc* processes used by developers to configure and deploy component middleware for DRE systems.

VII.1.2 Optimization Techniques

The following is a summary of lessons learned from our work developing and applying PAM to optimize component-based DRE systems at deployment-time:

1. **Deployment phase should be treated with equal importance.** The benefits provided by component middleware significantly alter the lifecycle of DRE system development with system deployment achieving importance similar to design, development and analysis/verification. The presence of a separate, well-defined deployment phase in DRE system development allows deferring key system decisions to an intermediate stage between the traditional design/development-time vs. run-time. By using information available at deployment-time (not available at design/development-time and information that is too late to be useful at run-time), the deployment phase opens up possibilities for system optimizations impossible in previous generations. In addition to system optimizations, deferring key system decisions until deployment-time in turn increases reuse by de-coupling deployment-time variability from component functionality.

2. **Application-specific optimizations are critical to building large-scale systems.** Although general-purpose optimizations serve to improve the performance of all systems, application-specific optimizations have the potential for going one-step beyond. Our approach illustrated the performance and footprint benefits of performing deployment-time optimizations in an application-specific fashion. An alternative approach is to perform these optimizations at run-time. For example, the middleware could try to dynamically place components with similar QoS policies into the same container instead of our deployment-time application-specific approach. Dynamic placement of components would necessitate the middleware to keep state identifying the different QoS policies in effect, track changes to the QoS policies and to mediate access to QoS policy changes. Such an approach, however, will quickly become

infeasible in large-scale components due to the excessive state the middleware needs to maintain to track the QoS policy relationships.

By performing the optimizations in an application-specific fashion, we can obtain the benefits of such placement without the overhead of maintaining state. Large-scale systems thus start exhibiting an interesting inversion of the traditional process: instead of the application conforming to the characteristics of the middleware, the middleware needs to conform to the characteristics of the application.

3. **Optimizations should be performed across layers in any layered architecture.** Apart from reducing the latency, our results indicate that irrespective of the number and kind of optimizations performed at the middleware layer, the middleware is ultimately restricted to the context information available to it. By using higher-level abstractions like the MDE approach used in PICML, we can perform a class of optimizations that are not possible at the middleware layer alone. By effectively combining the deployment information with the QoS policy information, our QoS policy fusion algorithm can optimize the configuration of DRE systems without affecting the component application logic. Our results show that any optimizations performed on a system with a layered architecture can significantly benefit from propagation of context information freely across the different layers. In addition to the propagation of deployment to the middleware, there is a need to be able to propagate information from levels both above the application deployment (*i.e.*, application functionality) and below the middleware (*i.e.*, operating system and system hardware). Our approach currently only unifies two of these layers and needs to be extended to encompass all layers in a DRE system.

4. **MDE has the potential to serve as the unifying foundation for building DRE systems.** In order to be able to separate the different phases of DRE system lifecycle, but also achieve the benefits gained from propagating information from one phase to

another, it is necessary to create a pipeline for DRE system development. Just like a pipeline in a micro-processor relies on a common instruction set and allows processing instructions by splitting each instruction into a number of different stages, models and model-driven engineering can provide the basis for building a DRE system development pipeline. By using models as the carriers of information across the different stages of such a DRE system development pipeline, we can realize the goals of exposing information across the different stages, thereby leading to creation of optimized DRE systems.

VII.1.3 Integration Techniques

The following is a summary of lessons learned from our work developing and applying the SIML (meta)model composition MDE tool-chain to integrate heterogeneous middleware technologies:

1. **Integration tools are becoming as essential as design tools.** SIML is designed to bridge the gap between existing *component technologies* (in which the majority of software systems are built) and *integration middleware* (which facilitate the integration of such systems). SIML elevates the activity of integration to the same level as system design by providing MDE tools that support integration design of enterprise distributed systems built with heterogeneous middleware technologies.

2. **Representation and evaluation of service-level agreements is a crucial aspect of integration.** Since SIML is a DSML, it can potentially be used as the infrastructure to define constraints on the integration process itself, thereby allowing evaluation of service-level agreements prior to the actual integration. For example, the MDE-based

approach used in SIML allows extensions to support associating service-level agreements (SLAs) on sub-systems being integrated, and evaluate such SLAs at design-time itself. The integration can be evaluated from the perspective of "quality-of-integration," in addition to evaluation for feasibility of integration from a functional perspective.

3. **Automating key portions of the integration process is critical to building large-scale distributed systems.** Compared with conventional approaches, SIML's MDE approach to system integration automates key aspects of system integration, including gateway "glue code" generation, metadata management, and design-time support for expressing unique domain and/or implementation assumptions. It supports seamless migration of existing investment in models and allows incremental integration of new systems. SIML also helps integrate applications based on middleware technologies other than CCM and web services.

4. **Standards-based inter-operability of design-time tools is key to realizing the benefits of such tools.** Although our implementation of SIML is done using GME as the underlying modeling environment, our MDE approach is general-purpose and can be applied to tool-chains other than GME. For example, by adding support for import/export for XML Metadata Interchange (XMI) [97], models developed using tools such as IBM's Rational Software Architect, Objecteering UML and MagicDraw UML, could be imported into SIML, which can then be used to perform the key integration activities. Application developers and integrators can seamlessly realize the benefits of system development and integration using SIML's MDE-based approach.

5. **QoS integration is a complex problem, and requires additional R&D advances.** Though SIML helped map functional aspects of a system from a source technology to a target technology, the non-functional, QoS-related aspects of a system should

also map seamlessly. For example, technologies like the Real-time CORBA Component Model (RT-CCM) [142] support many QoS-related features (such as thread pools, lanes, priority banded connections, and standard static/dynamic scheduling services) that allow system developers to configure the middleware to build systems with desired QoS features. When systems based on RT-CCM are integrated with other technologies, it is critical to automatically map the QoS-related features used by an application in the source technology to the set of QoS features available in the target technology. For example, a number of specifications have been released for web services that target QoS features, such as reliable messaging, security, and notification. The focus of our future efforts in functional integration of systems will involve extending SIML to map QoS features automatically from one technology to another using DSMLs, such that the integration is automated in all aspects – both functional and non-functional.

6. **Integration design tools should become a part of the end-to-end software development cycle.** Ultimately, there is a need for integration design tools that help with functional integration, as well as other forms of integration, including data, presentation, and process integration. These design tools themselves require integration into the software development lifecycle to to provide an "application integration platform," similar to how software testing tools (such as JUnit [75] and NUnit [51]) have gained widespread acceptance and have become an integral part of the software development lifecycle.

VII.2 Summary of Research Contributions

In summary, this dissertation has made the following contributions to the study of techniques to improve the design/development, optimization and integration of component-based DRE systems:

- **Composition Techniques.** Our research resulted in the creation of a domain-specific modeling language, PICML, that provides a platform for expressing domain-specific constraints of COTS middleware technologies. PICML brings rigor to the otherwise unorganized and complex process of composing systems using standards-based component middleware like CCM. In particular, PICML provides a graphical DSML-based approach to define component interface definitions, specify component interactions, generate deployment descriptors, define elements of the target environment, associate components with these elements, and compose complex DRE systems from basic systems in a hierarchical fashion. PICML served (and continues to serve) as the foundation for a variety of MDE-based optimization and integration techniques, and other related research efforts.

- **Optimization Techniques.** Our research resulted in the creation of a new class of optimization techniques, "deployment-time" optimization techniques (with three variants), for reducing the overhead in large-scale COTS component middleware technologies. Our approach includes a family of related optimization techniques, all of which use the notion of "fusion" – combining multiple elements into a single element – to reduce the number of elements without affecting the original semantics. We described three algorithms – Local Component Fusion, Global Component Fusion, QoS Policy Fusion – which differ not only in the type of elements they operate on but also in the scope at which they operate. We demonstrated the effectiveness of our techniques by implementing them in PAM which resulted in a reduction of latency of about 81% and a reduction in the footprint of about 45%. The novelty of our techniques are that they are automatic (*i.e.*, do not require user intervention,) non-intrusive (*i.e.*, do not require changes to existing systems or implementations) and are standards compliant.

- **Integration Techniques.** Our research contributes a novel application of (meta)model

composition to perform functional integration of COTS component middleware technologies. We demonstrated the effectiveness of our techniques, by composing PICML and Web Services Modeling Language (WSML), two domain-specific modeling languages, to create the System Integration Modeling Language (SIML), which provides increased automation of functional integration by automatically generating "all" of the integration glue code directly from the models. We demonstrated the generality and benefits of our approach in SIML by targeting integration glue code generation for two different implementations of Web Services, gSOAP and Microsoft .NET Web Services.

The central theme of this dissertation has been the application of model-driven engineering (MDE) tools and techniques to solve a variety of design/development, optimization, integration and deployment problems with standards-based COTS middleware. We demonstrated the benefits and implications of such a MDE-based approach in the context of the CORBA Component Model.

VII.3 Future Research Directions

This section presents some future research directions based on our experience in applying MDE to compose, optimize and integrate component-based DRE systems. Our thoughts on new directions can be summarized as follows:

- **Incremental composition/validation environments.** One of the key capabilities of our composition technique in PICML is that the domain semantics are captured via constraints. Developers of DRE systems can check the system for violations at design-time as opposed to deployment-time or run-time. Although the design-time validation is beneficial, currently there is no feedback/guidance provided to the user as he/she progressively creates a large system. In order to enable composition/validation in an incremental fashion, designers of domain-specific modeling languages need to define a notion of "model validity" at any given instant of time.

Using these descriptions, tool environments like GME need to support enforcing the validity in an incremental fashion. Ensuring validity in an incremental fashion is hard due to the fact that a change to one element in a model could cascade a series of checks, which in turn could cascade more checks, and so on indefinitely. Depending on the scale of the system, and the number of constraints needed to be checked to ensure validity, this could prove to be computationally expensive. In addition to checking in an incremental fashion, the tool environment should also provide visual design cues about the severity of violations and what is needed to fix the violation. In order to provide design cues to fix violations, the tool environment needs to perform a "design-space exploration" [86]. Enabling composition of component-based systems with incremental validation and design cues is a hard problem and requires further research.

- **Integrating design-time models with components at runtime.** Our current approach to performing checks at design-time ensures that the DRE systems are valid at design-time. Although this approach works well for static DRE systems, a new class of dynamic DRE systems [62, 120] alter the composition of the systems at run-time in an adaptive fashion. In order to ensure that such systems generated dynamically satisfy the domain constraints, our current DSML-based approach needs to be extended such that we can create models that are accessible at run-time. There are a number of challenges when it comes to synchronizing the design-time state of a system with its current run-time state, including granularity of synchronization, frequency of synchronization, whether such synchronization can be done fast enough to be tolerable and both useful as well as complications due to the distributed nature of DRE systems. Further research on closing the loop between the design-time state of the system and the current dynamic state of the system is crucial to ensuring that the benefits of the DSMLs are not restricted to design-time, minimizing the utility of DSMLs for a whole class of dynamic DRE systems.

- **"Just-in-time" physical assembly generation.** Our current approach of optimizing component assemblies using "deployment-time" techniques as implemented in PAM works well for systems with static/semi-static composition. By "semi-static," we refer to systems which exhibit certain adaptations at run-time, but the set of such adaptations are defined apriori. For a certain class of systems, however, the PAM approach is not feasible since the mapping of components to nodes as well as the connectivity between components keeps changing dynamically. Unlike our current implementation in PAM, our "deployment-time" techniques need to be applied at run-time. Applying the fusion algorithms at run-time on systems that rely on a traditional source-code compilation model imposes its own challenges including availability of an environment to create an implementation through compilation. For certain other systems which are based on a virtual machine like Java and C#, however, it is an attractive option since composition of physical assemblies can be done dynamically. Further research is necessary to ensure that the creation of physical assemblies at run-time in a dynamic fashion is faster than the time between adaptations to the system.

- **Integration-time QoS decomposition.** Our approach in SIML to functional integration in an automated fashion is a first step towards integration, especially in DRE systems. As described in Section VII.1.3, ensuring QoS from the integrated system is crucial to ensuring overall success of the integration. There are two challenges with ensuring QoS in the integrated system: (1) Mapping QoS between entities implemented using heterogeneous COTS middleware but are connected together, (2) Mapping the system-level QoS requirements into a sub-system specific to QoS, where each sub-system could be implemented using different COTS middleware technologies. Mapping QoS between entities requires a semantic mapping of the elements of QoS configuration between the different COTS middleware technologies integrated. This is a non-trivial process because there could be incompatibilities between the

same type of QoS as implemented by the two technologies; or worse one technology could just be incapable of providing that type of QoS. Similarly, decomposing QoS into sub-system specific QoS is non-trivial as it could have implications on the modularity of the system. Further research is needed to ensure that the desired QoS is implemented by the integrated system as a whole.

APPENDIX A

LIST OF PUBLICATIONS

Research on PICML, PAM and SIML has lead to the following journal, conference and workshop publications.

A.1 Book Chapters

1. Douglas C. Schmidt, Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, and Aniruddha Gokhale. Model-driven Development of Component-based Distributed Real-time and Embedded Systems. *Model Driven Development for Distributed Real-time and Embedded Systems, edited by Sebastien Gerard, Joel Champea, and Jean-Philippe Babau*, Hermes, 2005

2. Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Akos Ledeczi. System Integration via Model Composition. *Designing Software-Intensive Systems: Methods and Principles*, Edited by Dr. Pierre F. Tiako, Published by Idea Group, 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. *Science of Computer Programming: Special Issue on Model Driven Architecture*, Edited by Mehmet Aksit, 2007

2. 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. *Elsevier Journal of Computer and System Sciences*, Volume 73, Number 2, pp. 171–185, March 2007

3. Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, Sandeep Neema. Developing Applications Using Model-Driven Design Environments. *IEEE Computer*, vol. 39, no. 2, pp. 33–40, Feb., 2006

4. Krishnakumar Balasubramanian, Aniruddha Gokhale, Yuehua Lin, Jing Zhang, and Jeff Gray. Weaving Deployment Aspects into Domain-Specific Models. *International Journal on Software Engineering and Knowledge Engineering*, vol. 16., no. 3, pp. 403–424, June 2006

5. 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 Design and Verification of Real-Time Embedded Software*, vol. 2, no. 3/4, pp.142–155, 2006

A.3 Refereed Conference Publications

1. Amogh Kavimandan, Krishnakumar Balasubramanian, Nishanth Shankaran, Aniruddha Gokhale, and Douglas C. Schmidt. QUICKER: A Model-driven QoS Mapping Tool. *Proceedings of the 10th IEEE International Symposium on Object/Component/-Service-oriented Real-time Distributed Computing*, pp. 62–70, 2007, Santorini Island, Greece.

2. Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Akos Ledeczi. Component-based System Integration via (Meta)Model Composition. *Proceedings of the 14th Annual IEEE International Conference and Workshop on the*

Engineering of Computer Based Systems (ECBS), pp. 93–102, 2007, Tucson, Arizona

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. *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 190–199, 2005, San Francisco, CA

A.4 Refereed Workshop Publications

1. Krishnakumar Balasubramanian, Douglas C. Schmidt. Ultra-Large Scale System Integration via Model Composition. *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Portland, OR USA October 2006

2. Krishnakumar Balasubramanian, Douglas C. Schmidt, Nanbor Wang, Christopher D. Gill. Towards Composable Distributed Real-time and Embedded Software. *Proceedings of the 8th IEEE Work- shop on Object-oriented Real-time Dependable Systems (WORDS)*, pp. 226–233, 2003, Guadalajara, Mexico,

REFERENCES

[1] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In Kim Guldstrand Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003. ISBN 3-540-21671-5.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, third edition, 2000.

[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. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.4.

[4] Krishnakumar Balasubramanian, Aniruddha S. Gokhale, Yuehua Lin, Jing Zhang, and Jeff Gray. Weaving deployment aspects into domain-specific models. *International Journal of Software Engineering and Knowledge Engineering*, 16(3):403–424, 2006.

[5] Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Akos Ledeczi. Component-based system integration via (meta)model composition. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2772-8. doi: http://dx.doi.org/10.1109/ECBS.2007.24.

[6] Dusan Bálek and Frantisek Plasil. Software connectors and their role in component deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V. ISBN 0-7923-7481-9.

[7] Keith Ballinger, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham, and Prasad Yendluri. WS-I Basic Profile. www.ws-i.org/Profiles/BasicProfile-1.1.html, April 2006.

[8] Don Batory, Roberto Lopez-Herrejon, and Jean-Phillipe Martin. Generating Product-Lines of Product-Families. In *Proceedings of the Automated Software Engineering Conference*, 2002.

[9] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2004.23.

[10] 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. ISBN 1-59593-265-8. doi: http://doi.acm.org/10.1145/1101560.1101567.

[11] Paul V. Biron and Ashok Malhotra et al. XML Schema Part 2: Datatypes. W3C Recommendation, 2001. URL www.w3.org/TR/xmlschema-2/.

[12] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4, pages 1–74. Kluwer Academic Publishers Group, 1999.

[13] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). Extensible Markup Language (XML) 1.0 (2nd Edition). W3C Recommendation, 2000. URL citeseer.nj.nec.com/bray00extensible.html.

[14] Chris Britton and Peter Bye. *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems*. Addison-Wesley Professional, Boston, MA, USA, May 2004.

[15] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of ACM*, 16(9):575–577, 1973. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362342.362367.

[16] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*, 4, April 1994.

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

[18] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference*, pages 15–28, June 2004.

[19] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. www.w3.org/TR/wsdl, March 2001.

[20] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 160–178. Springer-Verlag, 2001. ISBN 3-540-42800-3.

[21] Lionel Cons and Piotr Poznanski. Pan: A high-level configuration language. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 83–98, Berkeley, CA, USA, 2002. USENIX Association.

[22] World Wide Web Consortium. Web Ontology Language. `www.w3.org/2004/OWL/`, Feb 2004.

[23] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

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

[25] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 266–276, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-472-X. doi: http://doi.acm.org/10.1145/581339.581374.

[26] Dionisio de Niz and Raj Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems*, 2005.

[27] Linda DeMichiel and Michael Keith. Enterprise Java Beans 3.0 Specification: Simplified API. `jcp.org/aboutJava/communityprocess/final/jsr220/index.html`, May 2006.

[28] 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*, Anaheim, CA, October 2003. ACM, ACM Press.

[29] John DeTreville. Making system configuration more declarative. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

[30] Ada Diaconescu and John Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 44–53, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101918.

[31] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 214–221, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2114-2.

[32] Ulrich Drepper. How to write shared libraries. http://people.redhat.com/drepper/dsohowto.pdf, Nov 2002.

[33] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: a flexible, optimizing idl compiler. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 44–56, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258921.

[34] Matthew Emerson and Janos Sztipanovits. Techniques for Metamodel Composition. In *The 6th OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2006*, pages 123–139, Portland, OR, Oct 2006. ACM, ACM Press.

[35] Areski Flissi and Philippe Merle. A generic deployment framework for grid computing and distributed applications. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences. Proceedings, Part II*, volume 4276 of *Lecture Notes in Computer Science*, pages 1402–1411. Springer, 2006. ISBN 3-540-48274-1.

[36] Bryan Ford, Mike Hibler, and Jay Lepreau. Using annotated interface definitions to optimize rpc. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, page 232, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.225833.

[37] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6):37–46, 2002. ISSN 0018-9162. doi: dx.doi.org/10.1109/MC.2002.1009167.

[38] Apache Software Foundation. Apache virtual host documentation. http://httpd.apache.org/docs/2.2/vhosts/, 2006.

[39] G. Muller and R. Marlet and E.-N. Volanschi and C. Consel and C. Pu and A. Goel. Fast, Optimized Sun RPC Using Automatic Program Specialization. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 240, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8292-2.

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

[41] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 449–460, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-680-3. doi: http://doi.acm.org/10.1145/775152.775217.

[42] Christopher D. Gill, Ron Cytron, and Douglas C. Schmidt. Middleware Scheduling Optimization Techniques for Distributed Real-time and Embedded Systems. In *Proceedings of the* 7th *Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[43] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, New York, 2004.

[44] 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. ACM.

[45] 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.

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

[47] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series)*. Addison-Wesley Professional, 2006. ISBN 0321334434.

[48] James H. Hill, John Slaby, Steve Baker, and Douglas C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[49] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, Boston, MA, USA, October 2003.

[50] Frank Hunleth and Ron K. Cytron. Footprint and Feature Management Using Aspect-oriented Programming Techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 02)*, pages 38–45. ACM Press, 2002. ISBN 1-58113-527-0. doi: doi.acm.org/10.1145/513829.513838.

[51] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, Raleigh, NC, USA, 2004. ISBN 0974514020.

[52] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the singularity project. Technical report, Microsoft Research, 2005.

[53] IBM. MQSeries Family. www-4.ibm.com/software/ts/mqseries/, 1999.

[54] IBM. WebSphere. www.ibm.com/software/info1/websphere/index.jsp, 2001.

[55] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[56] R.M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY, 1972.

[57] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, 2004.

[58] Gabor Karsai, Sandeep Neema, Ben Abbott, and David Sharp. A Modeling Language and Its Supporting Tools for Avionics Systems. In *Proceedings of 21st Digital Avionics Systems Conf.*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[59] 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.

[60] Amogh Kavimandan, Krishnakumar Balasubramanian, Nishanth Shankaran, Aniruddha Gokhale, and Douglas C. Schmidt. Quicker: A model-driven qos mapping tool for qos-enabled component middleware. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 62–70, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2765-5. doi: http://dx.doi.org/10.1109/ISORC.2007.50.

[61] 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.

[62] John S. Kinnebrew, Ankit Gupta, Nishanth Shankaran, Gautam Biswas, and Douglas C. Schmidt. Decision-Theoretic Planner with Dynamic Component Reconfiguration for Distributed Real-time Applications. In *The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, Sedona, Arizona, March 2007.

[63] Ina Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 2001. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(00)00286-3.

[64] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G. Shin. Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers. In *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS 2003)*, Washington, DC, May 2003. IEEE.

[65] 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. ACM.

[66] 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.

[67] 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.

[68] Ákos Lédeczi, Greg Nordstrom, Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On Metamodel Composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, pages 756–760, Mexico City, Mexico, 2001. IEEE.

[69] SangJeong Lee, Kang-Won Lee, Kyung Dong Ryu, Jong-Deok Choi, and Dinesh Verma. Ise01-4: Deployment time performance optimization of internet services. *Global Telecommunications Conference, 2006. GLOBECOM'06. IEEE*, pages 1–6, Nov 2006.

[70] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23 (1-2):85–126, 2002. ISSN 0922-6443.

[71] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web Service Level Agreement Language Specification. `researchweb.`

watson.ibm.com/wsla/documents.html, January 2003.

[72] Chris Lüer. Evaluating the eclipse platform as a composition environment. In *3rd International Workshop on Adoption-Centric Software Engineering ACSE 2003, ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 789–790. IEEE Computer Society, 2003.

[73] Chris Lüer and David S. Rosenblum. Wren—an environment for component-based development. In *ESEC / SIGSOFT FSE*, pages 207–217, 2001.

[74] Gabor Madl, Sherif Abdelwahed, and Gabor Karsai. Automatic Verification of Component-Based Real-time CORBA Applications. In *The 25th IEEE Real-time Systems Symposium (RTSS'04)*, Lisbon, Portugal, December 2004.

[75] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003. ISBN 1930110995.

[76] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst.*, 19(2):217–251, 2001. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/377769.377778.

[77] 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. IEEE/ACM.

[78] Bertrand Meyer. Applying Design By Contract. *Computer (IEEE)*, 25(10):40–51, October 1992.

[79] Mira Mezinia and Klaus Ostermann. Variability Management with Feature-oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1041685.1029915.

[80] Microsoft. Virtual address dump. http://support.microsoft.com/kb/927229, December 2006.

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

[82] Sun Microsystems. J2EE Connector Architecture Specification. java.sun.com/j2ee/connector/, November 2003.

[83] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/MIC.2004.58.

[84] Ingo Molnar. Linux with real-time pre-emption patches. http://people.redhat.com/mingo/realtime-preempt/, Sep 2006.

[85] Nirmal K. Mukhi, Ravi Konuru, and Francisco Curbera. Cooperative middleware specialization for service oriented architectures. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 206–215, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-912-8. doi: http://doi.acm.org/10.1145/1013367.1013401.

[86] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In Rajeev Alur and Insup Lee, editors, *Embedded Software, Third International Conference*, volume 2855 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003. ISBN 3-540-20223-4.

[87] Object Management Group. *CORBA Messaging Specification*. Object Management Group, OMG Document orbos/98-05-05 edition, May 1998.

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

[89] *CORBA Components v4.0*. Object Management Group, OMG Document formal/2006-04-01 edition, April 2006.

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

[91] *UML Profile for Enterprise Application Integration (EAI)*. Object Management Group, omg document formal/04-03-26 edition, March 2004.

[92] Object Management Group. *Fault Tolerant CORBA, Chapter 23, CORBA v3.0.3*. Object Management Group, OMG Document formal/04-03-10 edition, March 2004.

[93] *MetaObject Facility (MOF) 2.0 Core Specification*. Object Management Group, OMG Document ptc/03-10-04 edition, October 2003.

[94] *Minimum CORBA - Joint Revised Submission*. Object Management Group, OMG Document orbos/98-08-04 edition, August 1998.

[95] Object Management Group. *Real-time CORBA Specification*. Object Management Group, 1.2 edition, January 2005.

[96] *Unified Modeling Language (UML) v1.4*. Object Management Group, OMG Document formal/2001-09-67 edition, September 2001.

[97] *MOF 2.0/XMI Mapping Specification, v2.1*. Object Management Group, OMG Document formal/05-09-01 edition, September 2005.

[98] Object Technology International, Inc. *Eclipse Platform Technical Overview: White Paper*. Object Technology International, Inc., Updated for 2.1, Original publication July 2001 edition, February 2003.

[99] Martin Odersky and Matthias Zenger. Scalable component abstractions. *SIGPLAN Not.*, 40(10):41–57, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1103845.1094815.

[100] Nicole Oldham, Kunal Verma, Amit Sheth, and Farshad Hakimpour. Semantic ws-agreement partner selection. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 697–706, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-323-9. doi: http://doi.acm.org/10.1145/1135777.1135879.

[101] Ömer Erdem Demir, Prémkumar Dévanbu, Eric Wohlstadter, and Stefan Tai. An aspect-oriented approach to bypassing middleware layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-615-7. doi: http://doi.acm.org/10.1145/1218563.1218567.

[102] OMG. *The Common Object Request Broker: Arch. and Specification*. OMG, 2002.

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

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

[105] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 89–110, London, UK, 2002. Springer-Verlag. ISBN 3-540-43759-2.

[106] Adrian Paschke, Jens Dietrich, and Karsten Kuhla. A logic based sla management framework. In Lalana Kagal, Tim Finin, and James Hendler, editors, *ISWC '05: Proceedings of the Semantic Web and Policy Workshop, 4th International Semantic Web Conference*, pages 68–83, Baltimore, MD, USA, November 2005. UMBC eBiquity Research Group.

[107] Shankar R. Ponnekanti and Armando Fox. SWORD: A Developer Toolkit for Web Service Composition. In *WWW '02: Proceedings of the World Wide Web Conference: Alternate Track on Tools and Languages for Web Development*, New York, NY, USA, jan 2002. ACM Press.

[108] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä,

Finland, 9–13 1997. Springer. ISBN ISBN 3-540-63089-9. URL citeseer.nj.nec.com/195556.html.

[109] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron. Achieving End-to-End Predictability of the TAO Real-time CORBA ORB. In 8th *IEEE Real-time Technology and Applications Symposium*, San Jose, September 2002. IEEE.

[110] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. In *Component Deployment, Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004, Proceedings*, volume 3083 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2004. ISBN 3-540-22059-3.

[111] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali. Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In *Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2003)*, Rio de Janeiro, Brazil, 2003.

[112] Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, Redmond, WA, USA, 2002. ISBN 0735614229.

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

[114] Marshal Rose. The Blocks Extensible Exchange Protocol (BEEP) Core. *IETF Network Working Group Request for Comments, RFC 3080*, pages 1–58, March 2001.

[115] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

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

[117] 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.

[118] 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.

[119] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February

2002.

[120] Nishanth Shankaran, Douglas C. Schmidt, Yingming Chen, Xenofon Koutsoukous, and Chenyang Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.

[121] 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 *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, October 2004.

[122] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In Patrick Donohoe, editor, *Software Product Lines: Experience and Research Directions*, volume 576 of *The Springer International Series in Engineering and Computer Science*, New York, NY, USA, Aug 2000. Springer-Verlag.

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

[124] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[125] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

[126] Gurdip Singh and Sanghamitra Das. Customizing event ordering middleware for component-based systems. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 359–362, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2356-0. doi: http://dx.doi.org/10.1109/ISORC.2005.23.

[127] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/505145.505148.

[128] Connie Smith and Lloyd Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable*. Addison-Wesley Professional, Boston, MA, USA, September 2001.

[129] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. Vest: An aspect-based composition tool for real-time systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 58, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1956-3.

[130] Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Boston, 2000.

[131] Venkita Subramonian, Liang-Jui Shen, Christopher Gill, and Nanbor Wang. The design and performance of configurable component middleware for distributed real-time and embedded systems. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2247-5. doi: http://dx.doi.org/10.1109/REAL.2004.53.

[132] SUN. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.

[133] SUN. Java Remote Method Invocation (RMI) Specification. `java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`, 2002.

[134] Sun Microsystems. Enterprise JavaBeans Specification. java.sun.com/products/ejb/docs.html, August 2001.

[135] Clemens Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[136] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn et al. XML Schema Part 1: Structures. W3C Recommendation, 2001. URL `www.w3.org/TR/xmlschema-1/`.

[137] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[138] Bruce Trask, Dominick Paniscotti, Angel Roman, and Vikram Bhanot. Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 846–853, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176733.

[139] David TrowBridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu, and E. G. Nadhan. Integration Patterns. `msdn.microsoft.com/library/default.`

asp?url=/library/en-us/dnpag/html/intpatt.asp, June 2004.

[140] Robert van Engelen and Kyle Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 128–135, Los Alamitos, CA, USA, 2002. IEEE Computer Society. ISBN 0-7695-1582-7. URL csdl.computer.org/comp/proceedings/ccgrid/2002/1582/00/15820128abs.htm.

[141] W3C. Simple Object Access Protocol (SOAP) 1.1. www.w3c.org/TR/SOAP, May 2000.

[142] Nanbor Wang and Christopher Gill. Improving real-time system configuration via a qos-aware corba component model. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90273.2, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2056-1.

[143] Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *24th Computer Software and Applications Conference*, Taipei, Taiwan, October 2000. IEEE.

[144] 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'04)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, October 2004. Springer-Verlag.

[145] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.

[146] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321179366.

[147] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[148] Jules White, Douglas Schmidt, and Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study. In *MODELS 2006: 8th International Conference on Model Driven Engineering*

Languages and Systems, Montego Bay, Jamacia, October 2005. IEEE/ACM, ACM Press.

[149] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004. ISSN 0098-5589. doi: dx.doi.org/10.1109/TSE.2004.11.

[150] Inc ZeroC. The Internet Communications Engine^{TM}. www.zeroc.com/ice.html, 2003.

[151] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards just-in-time middleware architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052904.

[152] 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'02)*, page 301, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1585-1.

[153] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3 (1):1–20, 1997.