REUSABLE MODEL TRANSFORMATION TECHNIQUES FOR AUTOMATING MIDDLEWARE QOS CONFIGURATION IN DISTRIBUTED REAL-TIME & EMBEDDED SYSTEMS

By

Amogh Kavimandan

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, 2008

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Douglas C. Schmidt

Dr. Janos Sztipanovits

Dr. Gabor Karsai

Dr. Jeff Gray

In loving memory of my mother

and

To my wife, Kanchan, for her patience and support

ACKNOWLEDGMENTS

I am indebted to the following individuals for their guidance, support, encouragement, and friendship during my tenure as a graduate student at Vanderbilt University. The experiences I have had during the past five years while working in the DOC group will be, I think, invaluable to me in the future.

I would like to thank my Adviser, Prof. Aniruddha Gokhale, for giving me the unique opportunity of working with him at Vanderbilt University, which has been a greatly rewarding, enriching, and truly enjoyable experience for me. I am grateful to him for his guidance through every phase in my graduate research and education. The in-depth discussions I've had with him, and the constructive feedback I have received from him over the years, were crucial in concretization and development of my dissertation work and research artifacts resulting from it. Over the years, I have learned a great deal from him, on academic, professional, and personal front and I hope to continue my learning experience with him in the future as well. Next, I would like to thank Prof. Douglas C. Schmidt for numerous research collaborations over the last few years. It has been a pleasure working with Prof. Schmidt, and I was honored to have the opportunity to learn and colloborate with him.

I would like to thank Prof. Aniruddha Gokhale, Prof. Jeff Gray, Prof. Gabor Karsai, Prof. Douglas C. Schmidt, and Prof. Janos Sztipanovits for agreeing to serve on my PhD dissertation committee, for providing me with constructive feedback on my initial dissertation proposal, and for their time and effort spent on reviewing and suggesting improvements to this dissertation. I am particularly grateful to Prof. Gabor Karsai for collaborations over the past two years on the application of structural correspondence technique for verification of transformation algorithms in QUICKER.

I am especially thankful to Krishnakumar (Kitty) Balasubramanian for the initial ideas and continuous guidance on QUICKER. His technical comments, practical observations,

and insightful discussions on my dissertation subject were crucial for its further development and improvement. I am indebted to him for that.

Thanks are also due to my long-time mentor at Avaya Laboratories, Dr. Reinhard Klemm. He provided me with some very useful comments on MTS, and the initial dissertation proposal, and also provided with the use-case for applying it in the context of enterprise communications application.

My dissertation work was sponsored by a variety of sponsoring agencies. I would like to thank them for their support, and for providing real-world problems that motivated some of the work resulting from this dissertation. Specifically, I would like to thank the following agencies/researchers: The initial QUICKER funding came from the DARPA ARMS program. I am thankful to Richard Buskens at Lockheed Martin Advanced Technology Laboratories, Cherry Hill, for providing the funding that resulted in the development of various model transformation algorithms in QUICKER, and the evaluation studies conducted on QUICKER. I am also thankful to Avaya Laboratories for the research travel support during summer 2005; to Lockheed Martin Advanced Technology Center, Palo Alto, for providing us with the NASA Magnetospheric Multiscale space Mission scenario used as a case study in this dissertation; to Cisco Systems, San Jose, for their academic research gift; to the Vanderbilt EECS department for my adviser's faculty startup grant, which supported parts of this dissertation.

I was fortunate to have been associated with several knowledgeable people as my colleagues in the DOC group and in ISIS at Vanderbilt, and have learned a lot from each one of them. I would like to thank Krishnakumar (Kitty) Balasubramanian for bringing me up to speed on the nuances of CORBA and the CoSMIC toolchain and answering numerous questions over the past three years. I would also like to thank Nishanth Shankaran for the discussions on the dynamic reconfiguration aspects in the QUICKER toolchain. Thanks

are also due to Sumant Tambe for discussions on the initial draft of CQML; to Anantha Narayanan for many fruitful discussions on verification of correctness of QUICKER's model transformation algorithms.

I am thankful to the following people for making my stay in Nashville memorable, enjoyable, and fun: Ramya Balachandran, Jaiganesh Balasubramanian, Abhishek Dubey, Sebastian Eluvathingal, Nithin Gomez, Ashish Gupta, Vishal Koparde, Prajakta Koparde, Arvind Krishna, Manish Kushwaha, Anantha Narayanan, Srivatsan Pallavaram, Vishwa Ramachandran, Nilabja Roy, Indranil Roychoudhury, Nishanth Shankaran, and Di Yao.

Finally, I would like to thank my father, my brother, Nikhil, and his wife Reena, for all the support over the years; my wife, Kanchan, for her unwavering patience, love, and encouragement over the years in both good and bad times. Without her I could not have come this far.

Amogh Kavimandan

Vanderbilt University

14^{th}November 2008

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES ix

LIST OF FIGURES x

Chapter

I. Introduction 1

I.1. Overview of Component Middleware 2
I.2. Open Issues in QoS Configuration for Component-based DRE Systems 6
I.3. Research Approach 9
I.3.1. Model-driven QoS Mapping Toolchain & Algorithms . 9
I.3.2. Model Transformation Templatization & Specialization 11
I.4. Dissertation Organization 12

II. Model-driven QoS Mapping Toolchain and Algorithms 14

II.1. Taxonomy of Middleware QoS Configuration Approaches 18
II.1.1. Classification of Configuration Approaches 18
II.1.2. Comparing QoS Configuration Approaches 19
II.2. Challenges in Automated Middleware QoS Configuration 22
II.2.1. DRE system Case Studies 22
II.2.2. Design Challenges 27
II.3. Design of QUICKER 33
II.3.1. Specifying QoS Requirements using GT-QMAP Modeling Capabilities 35
II.3.2. Automating QoS requirements mapping using QUICKER 38
II.3.3. Applying QUICKER for Middleware QoS Configuration 43
II.4. Evaluating GT-QMAP Toolchain for Middleware QoS Configuration 44

III. On the Correctness of QUICKER Transformations 48

III.1. Overview of middleware QoS configuration process 50
III.2. Evaluation of QoS configuration process 52
III.2.1. DRE System Case Study 53

III.2.2. Verifying the correctness of our QoS configuration process . . . 54
III.2.3. Empirically evaluating BasicSP QoS configurations . . 60

IV. Optimization of QUICKER-generated QoS Configurations . . . 65

IV.1. Challenges in Optimizing QoS Configurations . . . 68
IV.2. Optimizing QoS Configuration for Component-based Systems . . 70
IV.2.1. Step I: Modeling Language used in the Transformation Algorithm . . . 71
IV.2.2. Step II: QoS Policy Optimization Algorithm . . . 72
IV.2.3. Resolving the Challenges in Optimizing QoS Configurations . . . 73
IV.3. Evaluating the generated QoS Configuration Optimizations . . . 74
IV.3.1. Representative Case Study . . . 74
IV.3.2. Experimental Setup & Empirical Results . . . 75
IV.3.3. Discussion . . . 78

V. Model Transformation Templatization and Specialization . . . 80

V.1. Representative Motivational Case Studies . . . 83
V.1.1. Communication Dialog Creation for an Insurance Enterprise . . . 83
V.1.2. Middleware QoS Configuration for Component-based Applications . . . 85
V.2. Templatized Model Transformations . . . 86
V.2.1. Step I: Defining the Templatized Transformation Rules . 87
V.2.2. Step II: Generating Variability Metamodels from Constraint Specifications . . . 92
V.2.3. Step III: Synthesizing a Specialization Repository . . . 95
V.2.4. Step IV: Specializing the Application Instances . . . 96
V.3. Evaluating the Merits of MTS . . . 98
V.3.1. Reduction in Development Effort using MTS . . . 99
V.3.2. Performance Overhead of using MTS . . . 101

VI. Applying MTS to Context-sensitive Enterprise Communication Dialog Synthesis . . . 103

VI.1. A Case Study Motivating Context-Sensitive Dialogs . . . 105
VI.2. Design Challenges in Context-Sensitive Dialog Synthesis . . . 108
VI.3. Templatized Model Transformation for Dialog Customization . . 111
VI.3.1. Applying MTS for Context-Sensitive Dialog Synthesis . 114
VI.3.2. Discussion . . . 122

VII. Related Work . . . 124

VII.1.Research on Middleware QoS Configuration . . . 124
VII.2.Research on Model Transformation Templatization . . . 138

VIII. Concluding Remarks 143

Appendix

A. List of Publications 148

A.1. Refereed Conference Publications 148
A.2. Refereed Workshop Publications 150

REFERENCES 152

LIST OF TABLES

Table Page

II.1. Characteristics of Science Application 25

II.2. Complexity of application scenarios 27

II.3. Comparing Requirements DSML against configuration space 46

II.4. Reduction in modeling effort using GT-QMAP 47

III.1. Generated QoS Configuration for BasicSP 62

V.1. SCV Analysis Results for QoS Configuration Case Study. 90

V.2. Details of the representative case studies. 99

VI.1. Dialog profiles for representative communication endpoints 116

VI.2. Using dynamic endpoint characteristics in dialog formatting & rendering 123

LIST OF FIGURES

Figure		Page
I.1.	Real-time QoS Mechanisms in CORBA Component Model	1
I.2.	Key Elements in the CORBA Component Model	3
I.3.	Research Landscape for QoS Assurance in DRE Systems	9
I.4.	QUICKER Model Transformation Toolchain for Automated Middleware QoS Configuration	10
I.5.	MTS Approach to Reusable Model Transformations.	12
II.1.	Configuration DSML snippet for RT request/response Configuration in LwCCM	21
II.2.	Shipboard Computing Environment Operational String	23
II.3.	MMS Mission System Components.	24
II.4.	Challenge 1: Mechanism-level Specification is the Wrong Abstraction for Component-based Application Development.	28
II.5.	Challenge 2: Identifying QoS Policy Set for Realizing Application QoS.	30
II.6.	Challenge 3: Ensuring Semantic Compatibility of & Resolving Dependencies between Application QoS Configurations.	31
II.7.	QUICKER toolchain for mapping QoS requirements to platform-specific QoS Options	34
II.8.	Simplified UML notation of QoS Requirements Associations in QUICKER	36
II.9.	Simplified UML notation of real-time QoS configurations DSML	37
III.1.	Model-driven QoS configuration process	51
III.2.	Structural correspondence using cross-links	55
III.3.	Dependency structure of BasicSP. L_c denotes threadpool lane and B_c denotes priority bands at component c. SD and CP indicate the SERVER_DECLARED and CLIENT_PROPAGATED priority models, respectively.	60

III.4. Evaluating BasicSP QoS configurations against increasing workload at a constant 20Hz invocation rate. 61

III.5. Evaluating BasicSP QoS configurations against increasing invocation rate: All the plots use logarithmic X axis and linear Y axis. 63

IV.1. Basic Single Processor 74

IV.2. Average end-to-end Latency 76

IV.3. Standard Deviation in Latency 77

V.1. Context-sensitive Communication Dialog Synthesis 81

V.2. Middleware QoS Configuration across a Heterogeneous Application . . . 81

V.3. A UML Representation of a Generic Communication Dialog. 84

V.4. A UML Representation of Middleware QoS Configuration Metamodels. . 85

V.5. Steps involved in developing model transformation using GReAT. 86

V.6. MTS Approach to Reusable Model Transformations. 87

V.7. Syntax of Constraint Specification Notation. 91

V.8. Templatized Transformation Rule in QoS Configuration Case Study. . . . 92

V.9. The Generation of VMM using MTS Higher-order Transformation. . . . 94

V.10. Generated VMM for the Representative Case Study. 95

V.11. A Sample VMM model for a Variant of QoS Configuration Case Study. . 95

V.12. Translations of Variabilities into VMM Model Objects. 96

V.13. Specializing the Application Instances using MTS Higher-order Transformation. 97

V.14. Specialization of a QoS configuration rule using MTS. 98

V.15. Overhead in using MTS for the development of templatized transformations. The Y axis denotes the time taken by Algorithms 4 and 5. 100

VI.1. MTS: Model Transformation Templatization and Specialization 113

VI.2. Generic dialog structure for supporting enterprise communication 114

VI.3. Auto-generated variability metamodel using SCV analysis results from Phase I 121

CHAPTER I

INTRODUCTION

A common requirement of a large number of systems found in domains such as avionics mission computing, shipboard computing, and intelligent transportation systems is their need for different real-time quality of service (QoS) properties, such as bounded request execution times, service prioritization, support for real-time asynchronous event-based communication, and low overhead event scheduling, filtering and dispatching. Systems with these characteristics are generally referred to as distributed, real-time and embedded (DRE) systems.

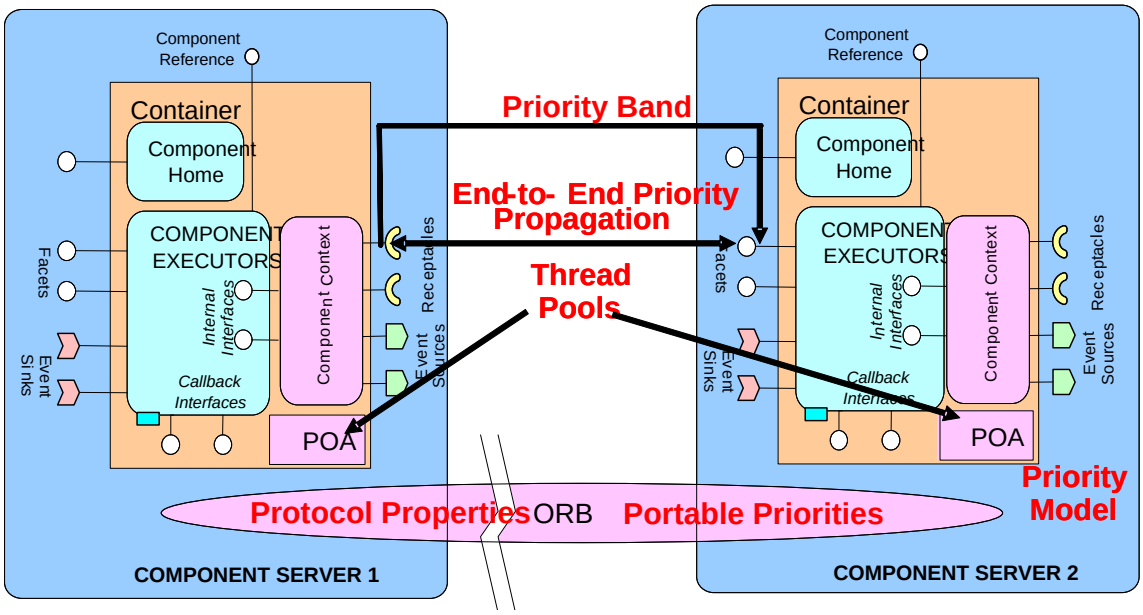


Figure I.1: Real-time QoS Mechanisms in CORBA Component Model

The increasing scale and complexity of modern DRE systems [79, 110] has prompted

their developers to move away from traditional stovepiped architectures to more open architectures that leverage newer software development paradigms [110] including component-oriented middleware platforms. The capabilities of component-based DRE systems are realized by deploying system functionality encapsulated within components on the resources of target environment, and configuring middleware, operating system (OS) and networking platforms on which these system components execute.

Contemporary component middleware platforms, such as Lightweight CORBA Component Model (LwCCM) [92], Enterprise Java Beans (EJB) [119] and .NET Web Services [84], are designed to be highly flexible to support a large class of DRE systems from multiple domains. The success of such component middleware technologies has raised the level of abstraction used to develop software for DRE systems. As a result, commercial-off-the-shelf (COTS) middleware, such as application servers and object request brokers (ORBs), now provides out-of-the-box support for traditional concerns affecting QoS in DRE system development, including multi-threading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms. This support helps decouple application logic from QoS mechanisms. For example, as shown in Figure I.1, the supported QoS options include portable priority mapping, end-to-end priority propagation, thread pools, distributable threads and schedulers, request buffering, and managing event subscriptions and event delivery. This in turn, shields the developers from low-level OS specific details, and promotes more effective reuse of such mechanisms.

I.1 Overview of Component Middleware

Component middleware technologies like EJB [119], Microsoft .NET [85], and LwCCM [91] 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*.

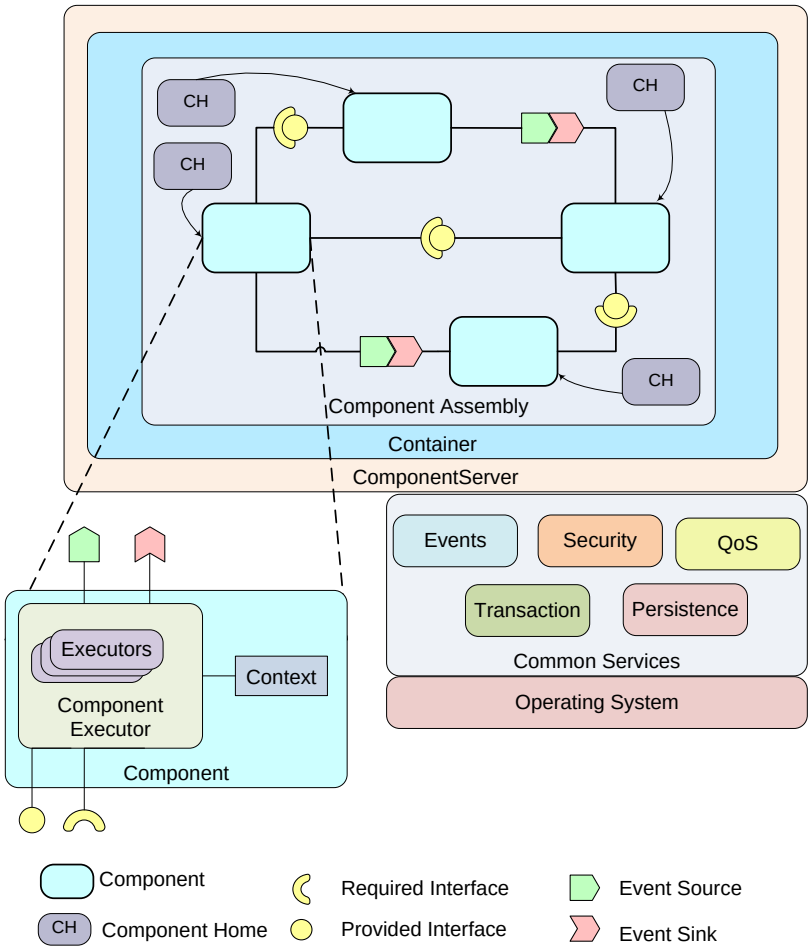


Figure I.2: Key Elements in the CORBA Component Model

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

- **Component**, which is the basic building block used to encapsulate an element of cohesive functionality. Components separate application logic from the underlying middleware infrastructure.
- **Component Ports**, which allow a component to expose multiple views to clients. Component ports provide the primary means for connecting components together to form assemblies.
- **Component Assembly**, which is an abstraction for composing components into

larger reusable entities. A component assembly typically includes a number of components connected together in an application-specific fashion. Unlike the other entities described here, there is no runtime 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 [106], 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 [14]. 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.

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.2, 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.2 Open Issues in QoS Configuration for Component-based DRE Systems

Although component middleware has helped move the configuration complexity away from the application logic, the middleware itself has become more complex to develop and configure properly. Assuring DRE system QoS properties involves multiple different

factors. Apart from making the right decisions on deployment and functional composition, it is critical to perform the *middleware QoS configuration* activity. Such an activity requires insights about different middleware configuration options, their impact on resulting QoS, and their inter-dependencies. These QoS options exist at various levels (*i.e.*, component-, component server-, and port-level) in Figure I.2 and the middleware QoS configuration must be performaned at each of these levels.

Specifically, the configuration process involves the binding of application level *QoS policies*—which are dictated by domain requirements—onto the solution space comprising the *QoS mechanisms* for tuning the underlying middleware. Examples of domain-level QoS policies include (1) the number of threads necessary to provide a service, (2) the priorities at which the different components should run, (3) the alternate protocols that can be used to request a service, (4) the granularity of sharing among the application components of the underlying resources such as transport level connections, (5) the number and size of outstanding requests that are permissible at any instant in time, and (6) the maximum and minimum amount of time to wait for completion of requests. All these must be mapped to middleware-specific configurations.

QoS configuration bindings can be performed at several time scales, including *statically*, *e.g.*, directly hard coded into the application or middleware, *semi-statically*, *e.g.*, configured at deployment time using metadata descriptors, or *dynamically*, *e.g.*, by modifying QoS configurations at runtime. Regardless of the binding time, however, the following challenges must be addressed:

1. The need to translate the domain-specific QoS policies of the application into QoS configuration options of the underlying middleware.
2. The need to choose valid values for the selected set of QoS configuration options.
3. The need to understand the dependency relationships and impact between the different

QoS configuration options, both at individual component level (local) as well as at aggregate intermediate levels, such as component assemblies, through the entire application (global).

4. The need to validate the local and global QoS configurations, which include the values, the dependency relationships, and the semantics of QoS configuration options at all times throughout the DRE system lifecycle.
5. The need to optimize QoS configurations for an application such that they can exploit platform-specific optimizations (for example, collocating components together has been shown to reduce latencies [7]).

Further, the above challenges must be addressed for each of the middleware platforms as well as every sub-application hosted on it, in the context of a DRE system. DRE system developers understand application-specific design and implementation issues but seldom have the necessary expertise to perform middleware QoS configuration. Failure to carefully map domain-level QoS requirements onto low-level middleware-specific configuration options can lead to a suboptimal middleware configuration degrading the overall system performance, and in worst cases cause runtime errors that are costly and difficult to debug. As a result, failures will stem from a new class of configuration errors rather than (just) traditional design/implementation errors or resource failures.

There is a significant need to bridge the gap that exists between domain-level requirements and platform-specific mechanisms that actuate the system QoS. Specifically, for DRE developers, it is desirable that system QoS requirements can be expressed in terms of higher level abstractions and automated techniques that map these requirements to low-level QoS configurations. In conclusion, the challenge of automated middleware QoS configuration (as shown in Figure I.3) has largely been unaddressed to date and needs to be resolved.

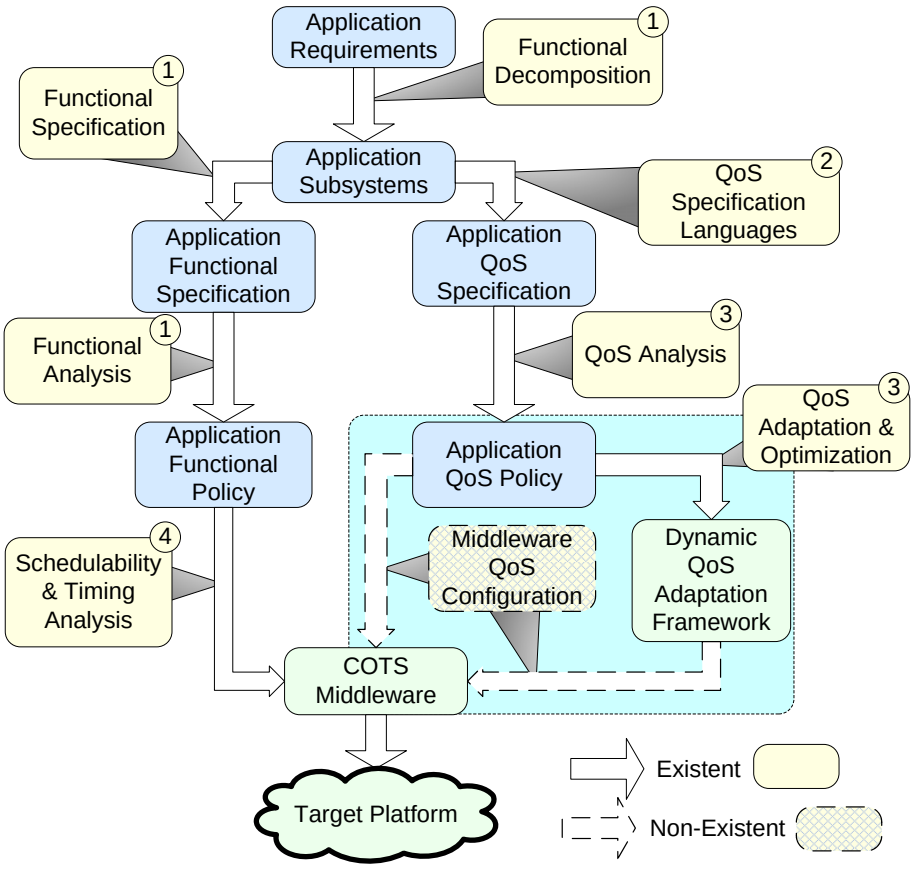


Figure I.3: Research Landscape for QoS Assurance in DRE Systems

I.3 Research Approach

This dissertation explores the use of MDE to solve the challenges outlined earlier in Section I.2.

I.3.1 Model-driven QoS Mapping Toolchain & Algorithms

To address QoS configuration challenges, we developed the *QUality of service pICKER* (QUICKER) model-driven engineering (MDE) toolchain. As shown in Figure I.4, QUICKER extends the *Platform-Independent Component Modeling Language* (PICML) [8], which is a domain-specific modeling language (DSML) built using the *Generic Modeling Environment* (GME) [2].

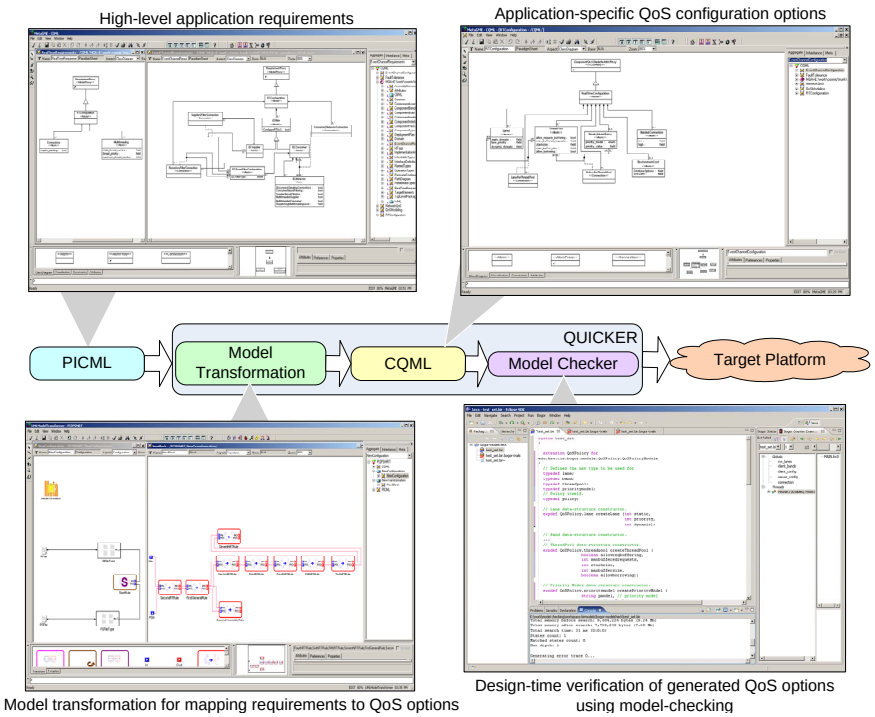


Figure I.4: QUICKER Model Transformation Toolchain for Automated Middleware QoS Configuration

QUICKER enables developers of component-based DRE systems to annotate applications with QoS policies. These policies are specified at a higher level of abstraction using *platform-independent* models, rather than using low-level platform-specific configuration options typically found in middleware configuration files. QUICKER thus allows flexibility in binding the same QoS policy to other middleware technologies. Before the components in a DRE system can be deployed, however, their platform-independent QoS policies must be transformed into platform-specific configuration options. QUICKER therefore uses model-transformation techniques [23] to translate the platform-independent specifications of QoS policies into a platform-specific model defined using the *Component QoS Modeling Language* (CQML), which models the QoS configuration options required to implement the QoS policies of the application specified in PICML. Unlike PICML (whose models are platform-independent), CQML models are specific to the underlying middleware infrastructure (which in our case is Real-time CCM [26]).

QUICKER subsequently uses generative techniques on the CQML model to synthesize the descriptors in a middleware-specific format (such as XML) required to configure the functional and QoS properties of the application in preparation for deployment in a target environment.

I.3.2 Model Transformation Templatization & Specialization

In this dissertation we present *MTS (Model-transformation Templatization and Specialization)* we have developed to address these questions in the context of visual model transformation tools. MTS provides transformation developers with a simple specification language to define variabilities in their application family such that the variabilities are factored out and are decoupled from the transformation rules. MTS provides a higher order transformation [1] [11] algorithm that automates the synthesis of a family-specific *variability metamodel*, which is used by transformation developers to capture the variability across the variants of an application family. Another higher order transformation algorithm defined in MTS generates the specialized instances of the application family variants. MTS requires minimal to no changes to the underlying model transformation engine.

MTS uses domain-specific modeling languages (DSMLs) [40] as its source and target languages. The MTS approach is shown in Figure I.5 and consists of the following steps:

1. Identifying the variabilities: In this step transformation developers analyze their application family to identify variabilities across the variants. Step 1 of Figure I.5 shows how these variabilities are input to the model transformation in terms of a simple *constraint notation specification*. This step decouples the transformation algorithm from its variabilities that can change in an instance-specific manner. This is akin to template functions [34] in C++ that outline the pattern of the function code.

2. Generating variability metamodel: In this step, developers use a higher order transformation (*i.e.*, those model transformations that work on meta-metamodels to translate

[1] Since the transformation(s) themselves become the input and(or) output, we refer to the transformation process in MTS as higher order transformations.

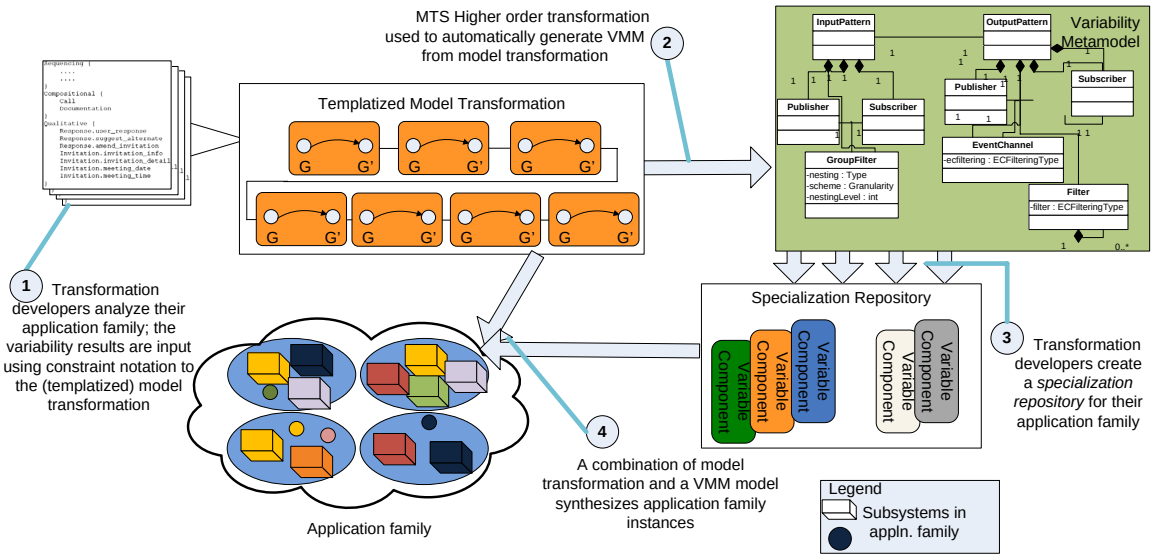


Figure I.5: MTS Approach to Reusable Model Transformations.

source metamodel(s) to target metamodel(s)) defined in MTS to automatically generate the variability metamodel (VMM) for their application family.

3. Synthesizing specialization repository: Next, developers create VMM models, where each VMM model corresponds to a family member. Thereafter, the variabilities identified in Step 1 are instantiated for every family member. A collection of all the VMM models is termed as a specialization repository of that family.

4. Specializing the application instances: Finally, as shown in Step 4, developers use another higher order transformation defined in MTS to create application variants. This step is similar to instantiating a C++ template where the compiler generates type-specific code based on the type of the argument passed.

I.4 Dissertation Organization

Our research on MDE-based middleware QoS configuration and model transformation templatization techniques have resulted in improved support for DRE system component-based software development. This dissertation is organized as follows: Chaper II introduces the MDE-based QUICKER toolchain, and discusses its input and output DSMLs. It

also discusses the overall approach and design of the toolchain, the model transformation algorithms for QoS mapping. It also discusses in detail our evaluations of its modeling capabilities and reduction in development effort using QUICKER. Chapter III discusses the empirical validation of the generated QoS configurations by applying it to a representative DRE system case study. Chapter IV explains how we have optimized the generated QoS configurations further by applying known techniques as model transformation algorithms. Chaper V discusses our templatization approach, its design and implementation, and also the various higher-order transformations used in MTS, and evaluates MTS. Chaper VI demonstrates the broader applicability of QUICKER's MTS and it in the context of an enterprise communications dialog synthesis case study, and shows how it handles the variabilities in MTS. Chaper VII discusses the existing research in middleware QOS configuration, templatization of model transformation and points out the differences between QUICKER and MTS research, respectively.

CHAPTER II

MODEL-DRIVEN QOS MAPPING TOOLCHAIN AND ALGORITHMS

Component-based software engineering (CBSE) [46] is finding wide acceptance in the development of modern distributed real-time and embedded (DRE) systems. Consequently component middleware platforms, such as Lightweight CORBA Component Model (LwCCM), are designed to be highly flexible to support a large class of DRE systems from multiple domains. These middleware platforms provide a number of configuration mechanisms for (1) allocating CPU, network and OS resources *a priori*, (2) (re)configuring and (re)deploying distributed system components, and (3) (de)marshaling communication requests, component activation/deactivation and persistence services, all of which are decoupled from the functional composition aspects of DRE systems.

Assuring DRE system QoS properties involves multiple different factors. Apart from making the right decisions on deployment and functional composition, it is critical to perform the *middleware QoS configuration* activity *i.e.*, correctly mapping system QoS properties onto the underlying middleware configuration options. Such an activity requires insights about different middleware configuration options, their impact on resulting QoS, and their inter-dependencies. DRE system developers understand application-specific design and implementation issues but seldom have the necessary expertise to perform middleware QoS configuration. Failure to carefully map domain-level QoS requirements onto low-level middleware-specific configuration options can lead to a suboptimal middleware configuration degrading the overall system performance, and in worst cases cause runtime errors that are costly and difficult to debug.

As discussed earlier in Chapter I, existing works in QoS assurance for DRE systems have focused on: (1) application functional specification, decomposition and analysis [45] to capture and study application structure and behavior, (2) QoS analysis, optimization and

adaptation [76] to allocate resources to applications, provide for application QoS optimization and adaptation in multiple QoS dimensions, and (3) schedulability and timing analysis [42, 116] to determine exact priorities and time periods for applications. Some work has also been done in QoS specification languages [8, 100, 128] for capturing application QoS properties by elevating middleware artifacts (such as its configuration options) to first class modeling entities. We argue that this level of abstraction does not resolve the challenges involved in middleware QoS configuration, which is the focus of this dissertation.

Solution Approach → Model-driven Middleware QoS Configuration. Model driven engineering (MDE) has shown significant promise and success in enabling the reasoning of system properties using domain-specific notations, and automating platform-specific artifacts using generative capabilities [37, 49, 73]. MDE has been successfully used in verification of system correctness properties [45], and functional and QoS modeling [8].

This dissertation outlines the challenges and conceptual ideas in middleware QoS configuration and describes our QUICKER MDE toolchain. It also delves into the details of the automated transformation capabilities, which are the cornerstone of tools like QUICKER. In particular, we describe QUICKER, which uses graph transformations [102, 108] on system models to automate the middleware QoS configuration. QUICKER uses a process of mapping domain-specific QoS requirements onto the right middleware-specific configuration options. Our model transformation-based approach begins with domain-specific, platform-independent models (PIMs) of DRE system QoS requirements that are automatically transformed to more refined and detailed middleware platform-specific models (PSMs). In this dissertation we focus only on the automated QoS configurations for real-time (RT) request-response and publish-subscribe communication dimensions shown in Figure I.3.

To address QoS configuration challenges, we developed the QUICKER MDE toolchain. Figure I.4 shows the toolchain. QUICKER extends the *Platform-Independent Component Modeling Language* (PICML) [8], which is a domain-specific modeling language (DSML)

built using the *Generic Modeling Environment* (GME) [2]. 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 DSMLs is also used to build models, which are instances of the metamodels. *Model interpreters* can be developed using the generative capabilities in GME. The interpreters are used to traverse the models for generating artifacts for analysis tools such as model-checking, emulation tools, etc.

QUICKER enables developers of component-based DRE systems to annotate applications with QoS policies. These policies are specified at a higher-level of abstraction using *platform-independent* models, rather than using low-level platform-specific configuration options typically found in middleware configuration files. QUICKER thus allows flexibility in binding the same QoS policy to other middleware technologies.

To describe and evaluate the algorithms developed for QUICKER, we use the following domain specific modeling languages (DSMLs) as input and output typed graphs for the automated QoS mapping: (1) Platform Independent Component Modeling Language (PICML) [8] used for modeling component assemblies, inter-and intra-assembly interactions and interfaces, and simplifying various activities of component-based system development such as packaging, and deployment, and (2) LwCCM QoS Modeling Language (CQML)that allows system developers to express QoS configurations at different levels of granularity using intuitive, visual representations.

The *Requirements metamodel* in QUICKER can be used to augment any system composition modeling language (SCML), such as PICML, that models functional composition of a DRE system hosted on a component middleware platform. The Requirements metamodel enables system models to be annotated with domain-specific QoS requirements. The *QoS Configuration metamodel* in CQML on the other hand models low-level, LwCCM-specific configuration QoS options. The transformation rules defined in QUICKER in terms of input and output typed graphs (*i.e.*, input and output metamodels) to automate the entire

middleware QoS configuration processs, thereby significantly reducing the system software development lifecycle costs and time-to-market.

QUICKER is designed to bridge the gap shown in Figure I.3 between:

Functional specification and analysis tools, such as PICML [9, 10] and Cadena [45], that allow specification and analysis of application structure and behavior,

Schedulability analysis tools, such as TIMES [4], AIRES [66], VEST [116], that perform schedulability and timing analysis to determine the exact priorities and time periods for application components, and

Dynamic QoS adaptation frameworks, such as the Resource Adaptation and Control Engine (RACE) [109] and QuO [135], that allocate resources to application components, monitor the QoS of the system continuously, and apply corrective control to modify the QoS configuration of the middleware at runtime.

By combining model transformation and generative techniques with advanced model-checking technologies, QUICKER automates the mapping of QoS policies of applications to QoS configuration options for a specific middleware technology. In particular, QUICKER's separation of platform-independent and platform-dependent concerns enables the use of PICML models to specify QoS policies that can be mapped to other types of middleware, such as Web Services and Enterprise Java Beans (EJB). As a result, developers can concentrate on inherent complexities in the application domain rather than wrestle with low-level middleware-specific configuration options. QUICKER also helps ensure the validity of the values for the QoS configuration options, both at the individual component (local) level and at the aggregate application (global) level.

Chapter Organization. The remainder of this chapter is organized as follows: Section II.2 describes motivating DRE systems we use to describe the challenges in QoS mapping; Section II.3 describes the QUICKER toolchain and how it addresses the challenges outlined in Section II.2; Section II.4 evaluates QUICKER QoS configuration capabilities in the context of the DRE system case studies.

II.1 Taxonomy of Middleware QoS Configuration Approaches

Performing middleware QoS configuration is critical to achieving the desired application QoS on a particular middleware platform. In this section we discuss different approaches to middleware configuration that can be adopted by DRE system developers elaborating on their pros and cons.

II.1.1 Classification of Configuration Approaches

We outline three approaches for middleware QoS configuration in this subsection.

A. Platform-specific Descriptors. Middleware platforms define standard schemata that allow specification of functinal and QoS properties of the application. For example, platforms such as J2EE, Microsoft .NET and Lightweight CORBA Component Model (LwCCM) use XML descriptor metadata for describing component assemblies, their interfaces and interactions and various non-functional properties. In order to configure their DRE system, the system developers need to learn the XML schema itself before manually populating the platform-specific descriptor document. Additionally, the DRE system developers must also ensure the validity of (a) descriptor document of their application, and (b) configurations of all application components and inter-connections. The above steps involved in this manual QoS configuration approach are crucial to avoiding failures and/or errors during the DRE system deployment; or worse, its execution phase.

B. Platform-specific Configuration Specification. In this approach, platform-specific specifications, for example domain specific modeling languages (DSMLs), are used to create configuration models of DRE system that capture its various configuration options. Further, these configuration DSMLs define type checking constraints that are enforced at design-time to ensure the validity of system configurations. Model interpreters are finally used by developers to synthesize syntactically correct descriptors necessary for DRE system configuration in preparation of its deployment. Use of modeling abstractions together with the generative capabilities of this approach (that can be used repeatedly) shields the

developers from low-level XML schema details and has been shown [8] to be faster than the first approach.

C. Platform-independent Requirements Specification. Although configuration DSMLs provide substantial benefits over a manual approach, developers still must carefully study various QoS options, their dependencies, and their impact on the resulting QoS in order to perform QoS configuration for their DRE systems. Recent studies [59] have shown that these platform-specific configuration complexities are at an incorrect level of abstraction for DRE developers and in fact, may negate the benefits of component middleware technologies.

OMG's Model Driven Architecture (MDA) [93, 94] development process centers around defining platform-independent application models and applying typed, and attribute augmented transformations to these models to generate detailed, platform-specific application models. Recently, this idea has been applied to specification of an application's non-functional (*i.e.*, QoS requirements) properties (in terms of platform-independent, domain-level models) and its subsequent transformation into middleware configurations (in terms of platform-specific models). Thus, using this approach, developers only need to specify the QoS requirements model of their DRE system. Model transformations automatically translate these requirements onto QoS configuration models, to which generative techniques can be applied (as discussed in approach B above) to synthesize system descriptors.

II.1.2 Comparing QoS Configuration Approaches

We now compare and evaluate the applicability of the three approaches across the following dimensions: (a) specification size, (b) scalability, (c) ease of use, and (d) flexibility. In each of the comparisons, we assume that the QoS specification does not reuse existing models (or descriptors, where applicable); *i.e.*, the specification is done from scratch.

A. Specification Size. QoS specification of an application consists of the following two parts: (a) the actual data values of QoS options themselves, and (b) syntactic rules necessary

to construct valid QoS specification. Listing 1 shows a snipplet of an XML descriptor for specifying RT request/response options for LwCCM-based applications.

```
1     <orbConfigs>
2     <resources>
3       <threadpoolWithLanes id="threadpool-2">
4         <threadpoolLane>
5           <static_threads>5</static_threads>
6           <dynamic_threads>0</dynamic_threads>
7           <priority>2</priority>
8         </threadpoolLane>
9         <threadpoolLane>
10          <static_threads>5</static_threads>
11          <dynamic_threads>0</dynamic_threads>
12          <priority>1</priority>
13        </threadpoolLane>
14        <stacksize>0</stacksize>
15        <allow_borrowing>false</allow_borrowing>
16        <allow_request_buffering>false</allow_request_buffering>
17        <max_buffered_requests>0</max_buffered_requests>
18        <max_request_buffered_size>0</max_request_buffered_size>
19      </threadpoolWithLanes>
```

Listing 1: XML Descriptor Snippet for RT request/response Configuration in LwCCM

As shown in lines 3-19, `ThreadPoolWithLanes` consists of the following two options: (a) `Lane`, specifying the number of thread resources and their type, and (b) `ThreadPool`, governing various characteristics of a pool of `Lanes`. For each option in this listing, the value of that option must be enclosed in appropriate XML tags such that the QoS specification is valid and complete.

Figure II.1 shows a snippet of a configuration DSML for the same RT request/response options specification. In order to completely capture QoS configuration for a DRE system, similar to a manual approach, all data values of various QoS options must be specified. However, as can be seen in the figure, since all the XML tags are modeled as reusable elements (*e.g.*, `ThreadPool` element in Figure II.1), the size of QoS specification using this configuration DSML is considerably smaller. A carefully designed Requirements DSML, which is our third approach, would be able to reduce configuration specification size even further as demonstrated in Section II.4.

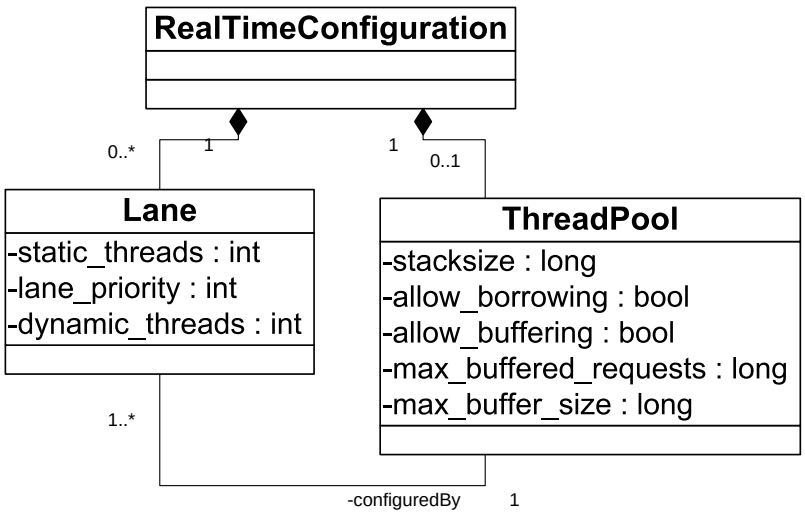


Figure II.1: Configuration DSML snippet for RT request/response Configuration in LwCCM

Recently introduced requirements DSMLs [61, 121] use system structure and platform-specific heuristics to deduce many of the QoS options from DRE system requirements specification. A carefully designed requirements DSML would be able to reduce configuration specification size even further. This hypothesis is corroborated by our work [56] which shows that the configuration effort using requirements a DSML is reduced by over 75%.

B. Scalability and Ease-of-use. Manually modifying descriptors for QoS configuration is the least scalable of the three approaches and has been previously shown to be either extremely tedious or in some cases infeasible [8] because of sheer size and complexity of descriptor files.

Ease-of-use would be highest with Requirements DSMLs since they operate on domain-level abstractions that are well-understood by DRE developers. Configuration DSMLs provide reusable modeling abstractions, and thus are easier to use than the manual approach.

C. Flexibility in QoS configuration. Assuming that they closely model descriptor schemata, both configuration DSMLs and manual approaches provide the same high degree of flexibility in QoS configuration. Requirements DSMLs on the other hand, ratiocinate many of the options. In other words, these DSMLs do not explicitely capture all QoS options and

therefore are not as flexible. However, as shown in Section II.3, these can be used by generative algorithms to synthesize the configuration options by way of mapping them from the application QoS requirements or deducing them from certain application characteristics.

In summary, even though they are not as flexible, requirements DSMLs shield DRE developers from the complexities of low-level middleware QoS mechanisms and are thus better suited for rapid QoS configuration. In the subsequent sections, we discuss the details of our QUICKER toolchain and show how we leverage its Requirements DSML and model transformations to automate the QoS configuration activity.

II.2 Challenges in Automated Middleware QoS Configuration

Section II outlined the need for automating the tedious and error-prone process of middleware QoS configuration. Developing a scientific approach to automate this activity poses a certain set of challenges. We discuss these challenges in the context of three case studies, which we also use in the chapter for evaluating our approach.

II.2.1 DRE system Case Studies

We chose the following DRE systems as the application scenarios for our experiments:

BasicSP. The Basic Single Processor (BasicSP) [110] is a scenario from the Boeing Bold Stroke component avionics computing product line [110, 111]. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed and configured using a QoS-enabled component middleware platform. The application is deployed using a single deployment plan on two physical nodes.

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are fixed at 20 Hz. The scenario shown in Figure IV.1 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component

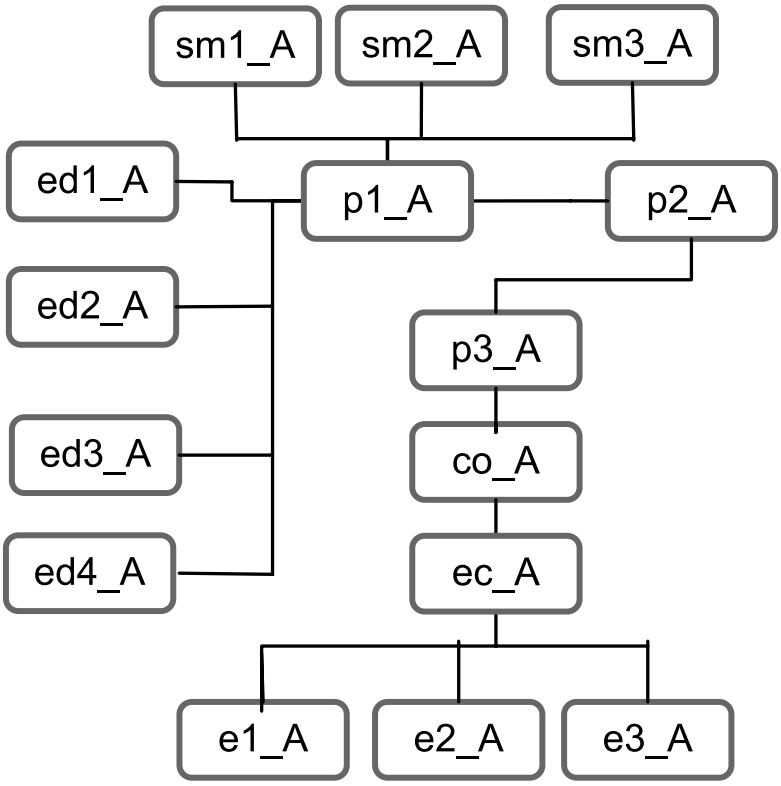


Figure II.2: Shipboard Computing Environment Operational String

generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the Airframe and updates its state and displays it to the pilot.

SCE. The Shipboard Computing Environment (SCE) consists of a sequence of several components connected together to form multiple operational strings[1]. Each operational string has different importance levels and these levels are used to resolve any resource contention between them.

As shown in Figure II.2, each operational string contains a number of sensor components (*e.g.*, *ed1_A*,*ed2_A*) and system monitor components (*e.g.*, *sm1_A*, *sm2_A*) that publish data from the physical devices to a series of planner components (*e.g.*, *p1_A*, *p2_A*).

[1] A single operational string is represented as a component assembly inside the application model

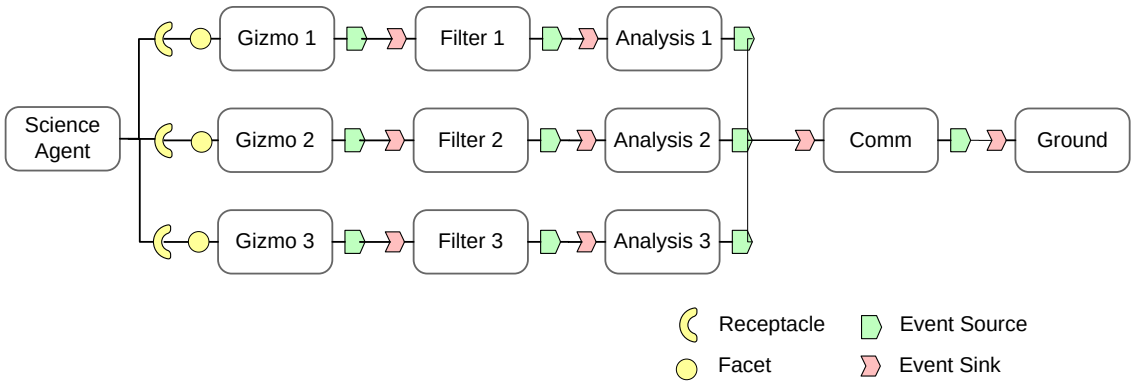


Figure II.3: MMS Mission System Components.

Once the inputs from sensors and system monitors have been analyzed, the planners perform control decisions using the effector components (*e.g.*, *e1_A*, *e2_A*). Each operational string contains ten components altogether. SCE has ten operational strings that are deployed using ten deployment plans on five physical nodes.

MMS. We use NASA's Magnetospheric Multi-scale (MMS) space mission (stp.gsfc.nasa.gov/missions/mms/mms.htm) as an example to motivate the need for automated tools for mapping the domain-specific QoS requirements to middleware-specific QoS configurations. NASA's MMS mission is a representative DRE system consisting of several interacting subsystems with a number of complex QoS requirements. It consists of four identical spacecrafts that orbit around a region of interest in a specific formation. These spacecrafts sense and collect data specific for the region of interest and at appropriate time intervals send it to the ground stations for further analysis.

Application developers of the MMS mission must account for mission-specific QoS requirements along two separate dimensions: (1) each spacecraft needs to operate in multiple modes, and (2) each spacecraft collects data using sensors whose importance varies according to the data being collected. The MMS mission involves three modes of operation: *slow*, *fast*, and *burst* survey modes. The *slow* survey mode is entered outside the regions of scientific interests and enables only a minimal set of data acquisition (primarily for health monitoring). The *fast* survey mode is entered when the spacecrafts are within

one or more regions of interest, which enables data acquisition for all payload sensors at a moderate rate. If plasma activity is detected while in fast survey mode, the spacecraft enters *burst* mode, which results in data collection at the highest data rates. Resource utilization by, and importance of, a science application is determined by its mode of operation, which is summarized by Table II.1.

Mode	Relative Importance	Resource Consumption
Slow survey	Low	Low
Fast survey	Medium	Medium
Burst	High	High

Table II.1: Characteristics of Science Application

Each spacecraft consists of an on-board intelligent mission planner, such as the *spreading activation partial order planner* (SA-POP) [64] that decomposes overall mission goal(s). SA-POP employs decision-theoretic methods and other AI schemes (such as hierarchical task decomposition) to decompose mission goals into navigation, control, data gathering, and data processing applications. In addition to initial generation of GNC and science applications, SA-POP incrementally generates new applications in response to changing mission goals and/or degraded performance reported by on-board mission monitors.

A prototype of the data processing subsystem of this distributed system has been developed [120] by our collaborators at Vanderbilt University using the *Component-Integrated ACE ORB* (CIAO) [26] QoS-enabling component middleware framework, the RACE [109] dynamic QoS adaptation framework and the PICML [8]. In this case study section we focus on the physical activity sensor, data collection, and transmission challenges in the MMS mission, which NASA is developing to study the microphysics of plasma processes.

Figure II.3 shows the components and their interactions within a single spacecraft. Each spacecraft consists of a *science* agent that decomposes mission goals into navigation, control, data gathering, and data processing applications. Each science agent communicates with multiple *Gizmo* components, which are connected to different payload sensors. Each

Gizmo component collects data from the sensors, which have varying data rate, data size, and compression requirements.

The data collected from the different sensors have varying importance, depending on the mode and on the mission. The collected data is passed through *Filter* components, which remove noise from the data. The *Filter* components pass the data onto *Analysis* components, which compute a quality value indicating the likelihood of a transient plasma event. This quality value is then communicated to other spacecraft and used to determine entry into burst mode while in fast mode. Finally, the analyzed data from each *Analysis* component is passed to a *Comm* (communication) component, which transmits the data to the *Ground* component at an appropriate time.

The use of QoS-enabled component middleware and MDE tools provided several advantages during development of software components for our MMS mission prototype. For example, we modeled all components of the prototype using PICML, which (1) supported a high-level abstraction for describing the structure of the MMS scenario and (2) automated the generation of deployment metadata used to deploy the MMS components. Likewise, implementing the components with CIAO enhanced flexibility by supporting runtime component swapping [5] that allowed runtime reconfiguration of the algorithms used by the *Filter* and *Analysis* components. Finally, using RACE to control the resource usage of the CIAO components allowed dynamic management of resources used by the *Gizmo*, *Filter*, and *Analysis* components. Dynamic resource management helps our MMS prototype adapt to changes in mission goals as determined by the *Science* components in response to changing conditions or as requested by explicit user commands.

Configuration complexity of scenarios. As already mentioned, in this chapter we have focussed on QoS specification for request-response and publish-subscribe communication paradigms. From our past experiences with developing and configuring QoS for DRE systems [59], we chose a 3-tuple $\{C;I;D\}$ to represent configuration complexity of our application scenarios where,

1. *C* defines the number of components of the application.
2. *I* defines distinct number of interactions between components of the application. An interaction exists between two components if the outgoing port of one is connected to incoming port of the other.
3. *D* defines the distinct number of dependencies between components of the application. A dependency exists between two components if a change in the QoS configuration of one necessitates a change in configuration of the other.

Table II.2: Complexity of application scenarios

Application scenarios	# of components	# of component interactions	# of component dependencies
BasicSP	4	5	6
MMS	12	11	43
SCE	150	260	950

The application scenarios described in this subsection illustrate different levels of configuration complexity and can be summarized using our 3-tuple definition as shown in Table II.2.

II.2.2 Design Challenges

Although QoS-enabled component middleware and existing MDE tools provide several advantages in software development, several key requirements need to be satisfied in order to effectively enable QoS configuration of the middleware platforms hosting the different software components of a DRE system, such as the MMS Mission prototype. In the remainder of this section, we discuss the challenges in automating the QoS configurations.

Challenge 1: Specifying domain-specific QoS requirements System developers are domain experts who can understand and reason about various domain-level issues. As shown in Figure II.4, the QoS requirements of a DRE system must be expressible in terms of *domain* concerns rather than in terms of low-level, middleware-specific mechanisms required

to satisfy these concerns. Additionally, the scale and complexity of middleware configuration space (which includes an appropriate and semantically valid subset of middleware configuration mechanisms and their values) makes the specification non trivial.

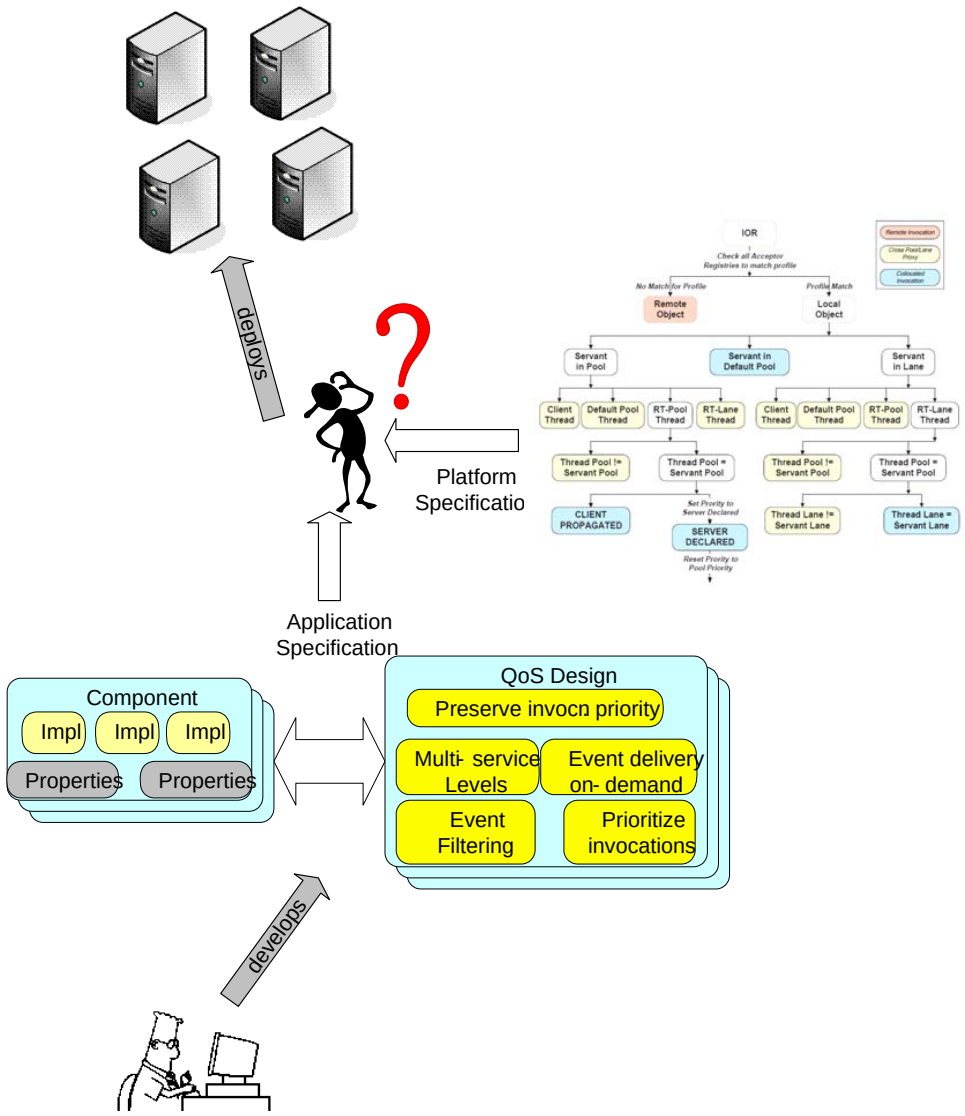


Figure II.4: Challenge 1: Mechanism-level Specification is the Wrong Abstraction for Component-based Application Development.

For example, a requirement for the asynchronous connection between *Comm* and *Analysis* components in the MMS mission is that its access be thread-safe such that only one

Comm component thread can access the asynchronous connection (for retrieving its events, for example) at any given time. Real-time publish/subscribe service provides advanced synchronization mechanisms in order to address such application requirements. It is highly desirable, however, for system developers to be able to specify these requirements at the domain-level instead of the middleware.

Addressing this challenge requires tool support for intuitive modeling capabilities that capture QoS concerns of a system using semantics and notations that are closer to the domain. Further, since DRE systems exhibit multidimensional QoS requirements, the tool should provide clearcut separation of concerns during system QoS specification. Section II.3.1 illustrates how our QUICKER toolchain addresses this challenge.

Challenge 2: Identifying the middleware-specific QoS configuration options for satisfying QoS requirements Although a tool may provide modeling capabilities to specify system-level QoS requirements, there remains the need to identify the right middleware-specific QoS configuration options that will satisfy the system QoS requirements. As shown in Figure II.5, this identification process can be a challenging task because of the following factors: (1) systems evolve either as part of the software development lifecycle, or modified domain requirements/end-goals. Naturally, the new middleware configurations would have to be identified again, which is a tedious and error-prone process, and (2) for large-scale systems this process becomes too time consuming, and in some cases infeasible.

For example, in the SCE application the planner component *p1_A* has the following requirements: (1) asynchronous connections with its client components (*i.e.*, here the system monitors) must support bursty service invocations from each of these components, and (2) service invocations from each of its client components must be prioritized. A way to satisfy the second requirement is by configuring the planner to have a SERVER_DECLARED real-time CCM (RT-CCM) [90] policy that handles invocations at pre-determined priorities. In addition, sufficient thread resources should be available to handle all client priority levels. This can be achieved by configuring the *ThreadPool with Lanes* feature, where a single

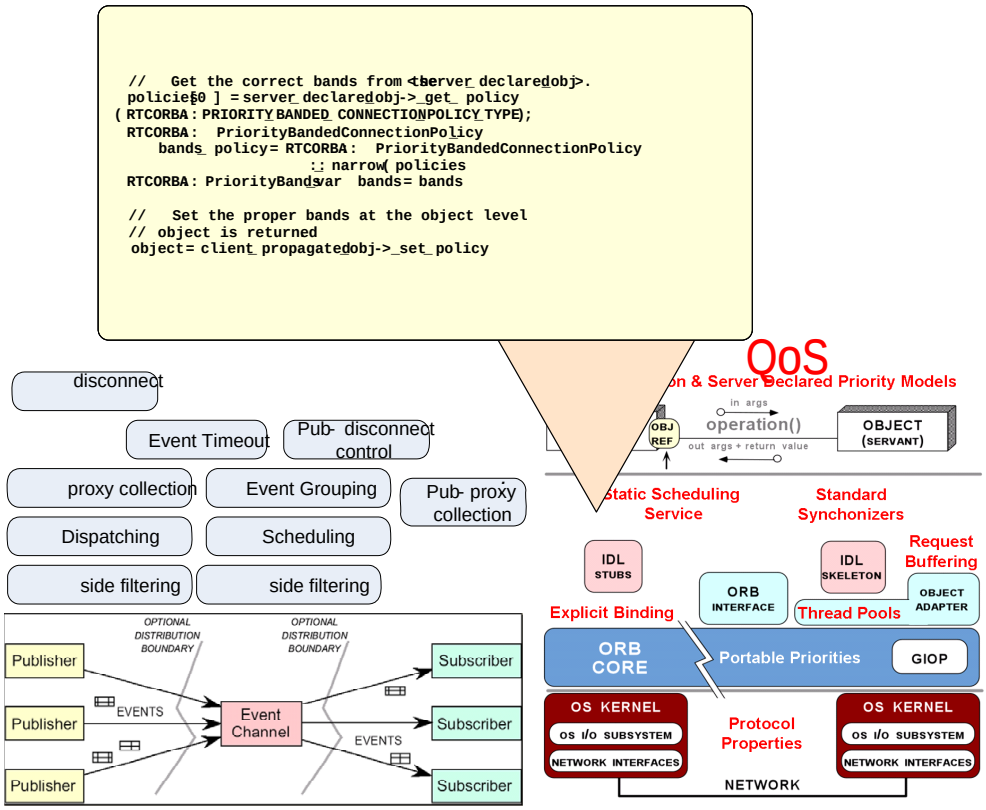


Figure II.5: Challenge 2: Identifying QoS Policy Set for Realizing Application QoS.

lane corresponds to an individual priority level. Such a QoS design scheme also ensures a predictable application execution and does not exhibit unbounded priority inversions [98]. Finally, in order to satisfy the first requirement, it is prudent and economical to assign dynamic thread resources for bursty clients [105] than reserving them in a static manner for the entire application life-cycle.

An automated QoS configuration tool should be able to codify these proven patterns and correctly identify the QoS options necessary to achieve desired system QoS from a given (semantically-correct) input model. If QoS requirements have been specified across more than one RT QoS dimensions, the tool should identify corresponding options pertaining to each of these dimensions. Section II.3.2 illustrates how QUICKER addresses this requirement.

Challenge 3: Mapping the QoS requirements onto QoS configuration options Even if

the QoS configuration options that satisfy the system QoS requirements may be identified, appropriate values for each of the configuration options must be chosen in order to correctly configure the middleware and realize system level QoS properties as shown in Figure II.6. Such a step would have to potentially be performed several times during the development cycle of a system and thus should be easily (and relatively quickly) repeatable.

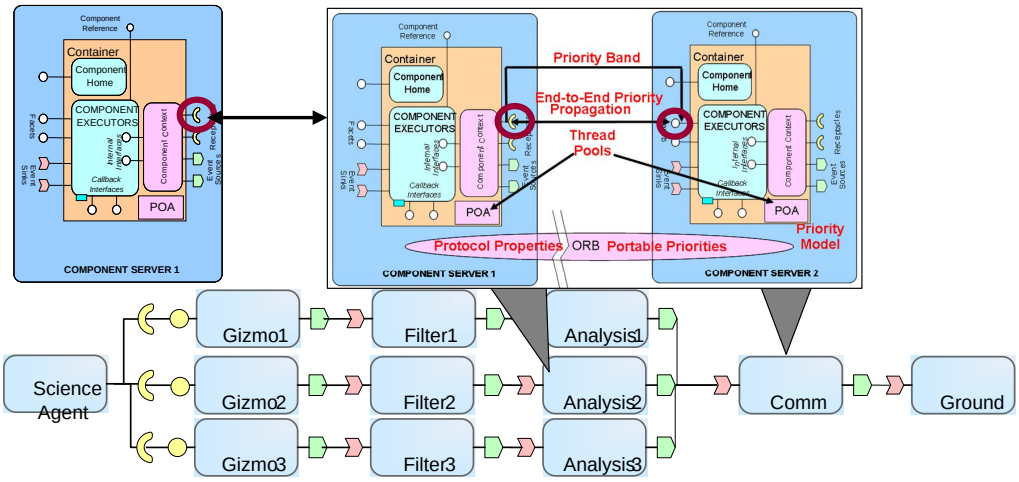


Figure II.6: Challenge 3: Ensuring Semantic Compatibility of & Resolving Dependencies between Application QoS Configurations.

For our MMS mission, the `Comm` component is best realized using the RT-CCM `Threadpool_with_lanes` feature so that it can provide varying levels of service to its clients. Similarly, the `Analysis` components require the use of banded connections to prevent priority of inversions on the communication links. In both of these cases, it is necessary to identify how many lanes are needed in a thread pool, what priority values should be set per lane, how many bands of communication are needed, and what priority ranges are handled by each band.

Depending on the individual QoS requirements, one or more alternative QoS options may be identified in the previous step. A QoS configuration tool should choose suitable values for each of these QoS options. Additionally, it should ensure that QoS options are

valid, both for the *association entity*[2], as well as for the entire component-based application. Section II.3.2 illustrates how QUICKER addresses this requirement.

Challenge 4. Ensuring validity of QoS configuration options with changes in QoS policies. QoS configuration options affect the non-functional behavior of a system, and thus are affected by changes in the system environment. For a DRE system to operate effectively in hostile environments, such as space missions, component middleware and their associated QoS configuration options may need to adapt to their current conditions. Middleware that can only be configured statically (*i.e.*, at design- or installation-time)—but does not allow dynamic reconfiguration—may be of limited use in these scenarios.

While it is useful to change QoS configuration options at runtime to affect changes in behavior (such as re-prioritizing or increasing/decreasing resource usage), such dynamic reconfigurations may incur another set of challenges. In particular, not only must we handle static QoS configuration problems (such as checking validity of values and keeping track of dependencies), there is typically little leeway to accommodate misconfiguration at runtime. It is non-trivial to change a running system because the system might crash during reconfiguration due to misconfiguration of QoS options. Moreover, the reconfiguration process itself must be predictable for the reconfiguration to have the desired effect on system behavior. In a DRE system, for instance, a reconfiguration done too late may be worse than not performing a reconfiguration at all.

In our MMS mission prototype, for example, the *science* agent(s) on all spacecraft have mission goals that represent requests from users or other *science* agents for the times and types of data to acquire. Such changes in mission goals require dynamic reconfiguration, which in turn can trigger changes in QoS configuration, such as modifying the relative importance assigned to the *gizmo* components. Depending on the nature and extent of the changes, dependent components of the *gizmo* component may be reconfigured using the available options, along with re-validating the values chosen for the options. For instance,

[2]In the context of component middleware, an association entity would be, for example, a component, a connection between components, or an assembly to which a QoS configuration is associated.

the size of the buffers in the *comm* agent corresponding to the data collected from the different *gizmo* components, may need reconfiguration to accommodate changes in the relative importance of the *gizmo* components.

Such exhaustive evaluation of possible choices of QoS configuration options and validation of the reconfigured state is too time consuming to perform at runtime and can delay the reconfiguration process itself, rendering it useless. Once again, tools are needed to help validate and automate this reconfiguration process. Section III.2.2.2 therefore describes how the QUICKER MDE tool helps evaluate possible choices of QoS configuration design-time evaluation of possible choices can be used to select runtime QoS configurations by the RACE dynamic QoS adaptation framework.

Thus, without an appropriate tool-support each of these challenges would have to be addressed by manually configuring the middleware for individual applications. Such ad-hoc solutions lead to sub-optimal QoS middleware configuration, degrading the overall application performance. In the worst case it might lead to runtime errors that are costly and difficult to debug. An automated tool support is therefore crucial to solve the middleware QoS configuration challenges effectively. The rest of the chapter discusses how QUICKER addresses these challenges in the context of CIAO component middleware.

II.3 Design of QUICKER

This section describes the QUICKER QoS mapping toolchain for QoS-enabled component middleware. QUICKER is a MDE framework, which relies on DSMLs for the description of high-level, domain-specific QoS requirements that enable capturing the (platform-independent) system requirements across various QoS dimensions. Additionally, QUICKER uses model-driven graph transformations [54] for the translation of these QoS requirements into platform-specific QoS configuration options necessary to realize these QoS requirements on the underlying platform.

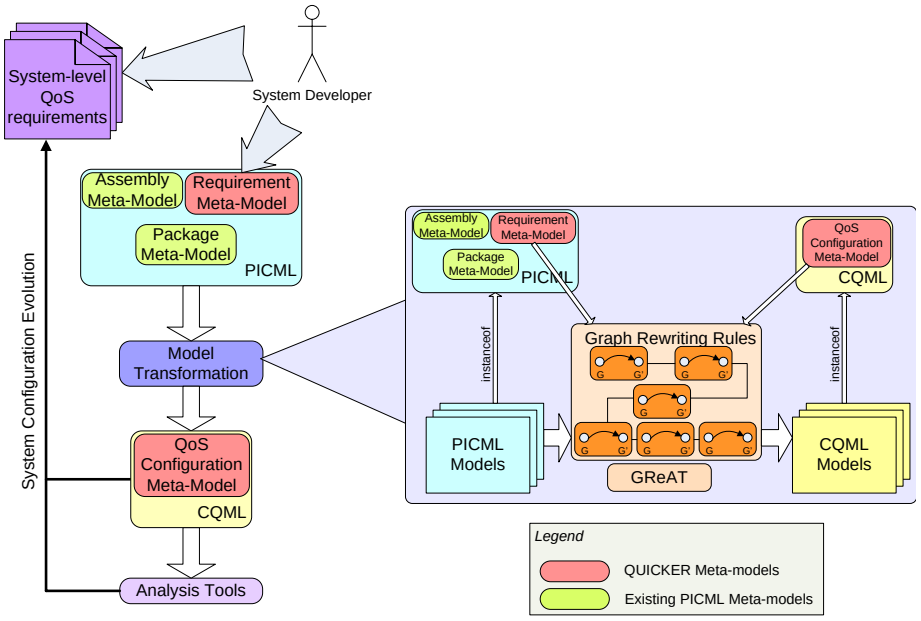


Figure II.7: QUICKER toolchain for mapping QoS requirements to platform-specific QoS Options

Figure II.7 shows the overall QUICKER toolchain. DRE system developers use the *Requirements* DSML in QUICKER to specify the system QoS requirements. A specification of system QoS requirements acts as the source model of the QUICKER transformation. Similarly, middleware-specific QoS configuration options are captured as models using the *QoS Configurations* DSML which serves as the target model in the transformation process.

QUICKER uses the Generic Modeling Environment (GME) [2] toolkit for developing the modeling languages used to describe the above, which provides a graphical user interface that can be used to define both DSML semantics and system models that conform to the DSMLs defined in it. *Model interpreters* can be developed using the generative capabilities in GME. The interpreters are used to traverse the models for generating artifacts for analysis tools such as model-checking, emulation tools, etc.

We have used the Graph Rewriting And Transformation (GReAT) [54] language for

defining model-to-model translations of QoS requirements.GReAT, which is developed using GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine). The graph rewriting rules are defined in GReAT in terms of source and target typed graph (*i.e.*, metamodels). QUICKER transformation rules are used by the GR-Engine in order to create the QoS options model of a DRE system from its QoS requirements model.

For evaluating QUICKER modeling capabilities and demonstrating them through a prototypical implementation, our Requirements DSML has been superimposed on PICML. The requirements modeling abstractions however, are not tied to PICML alone and thus can be generally associated with any other structural modeling language that provides capabilities for modeling functional entities (for example, a component, an assembly, or connections thereof) of a component-based system.

In order to be able to associate the QoS policies with structural units (for example, a component, an assembly, or connections thereof) of a component-based DRE system, the Requirements metamodel is superimposed on the Platform Independent Modeling Language (PICML) [8]. PICML can be used to capture the structure of a DRE system in terms of its components, assemblies, their interfaces and interactions. Unless stated otherwise, our use of PICML throughout the remainder of the chapter refers to its QoS Requirements modeling capabilities.

II.3.1 Specifying QoS Requirements using GT-QMAP Modeling Capabilities

In Challenge 1 of Section VI.2 we motivated the need for domain-specific QoS specification for component-based DRE systems. We define modeling constructs in GT-QMAP that can be used by the DRE system developers to define models that capture QoS requirements. This section describes this capability and how it resolves Challenge 1.

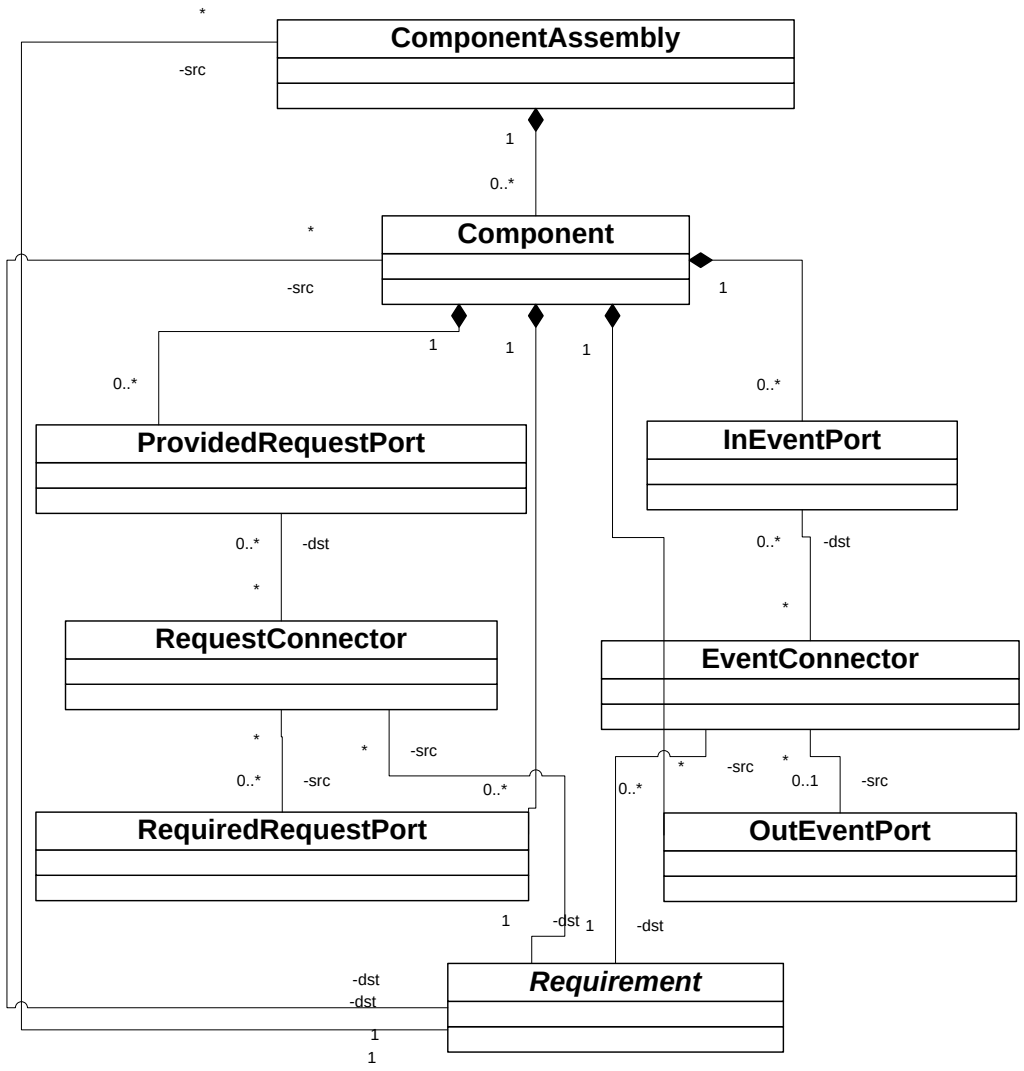


Figure II.8: Simplified UML notation of QoS Requirements Associations in QUICKER

II.3.1.1 Modeling QoS Requirements using QUICKER

QUICKER defines the `Requirement` element as a generalization of QoS requirements. As shown in Figure II.8, source elements `Component`, `ComponentAssembly` or `Port` connections can be associated with a `Requirement` element. Modeling abstractions in QUICKER allow association of multiple source elements with the same `Requirement` as long as those source elements are of the same *type*. Moreover a `ComponentAssembly`'s `Requirement` is also associated with all the components contained in that `ComponentAssembly`. Such associations provide significant benefits in terms of

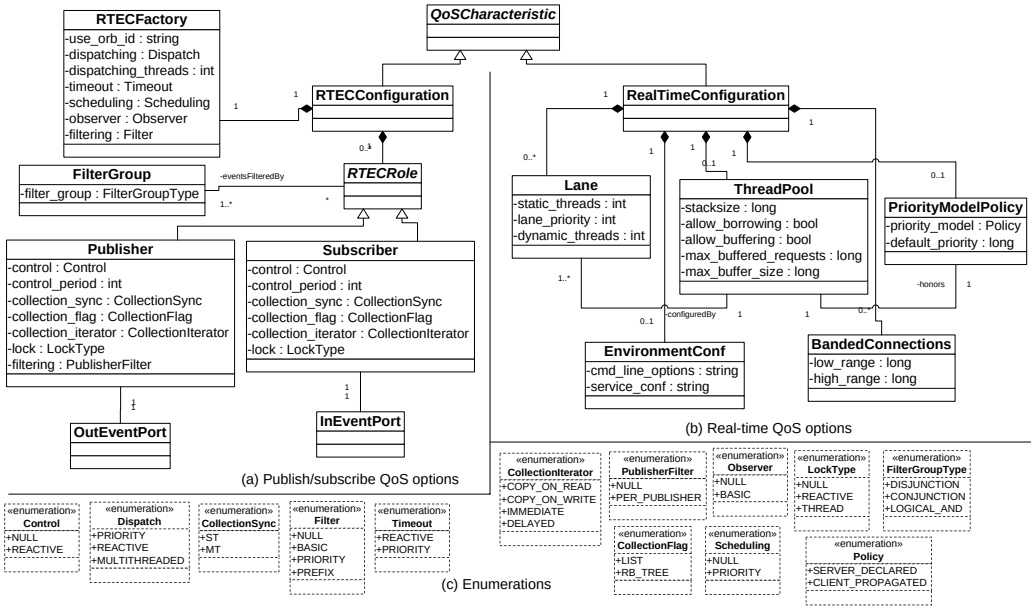


Figure II.9: Simplified UML notation of real-time QoS configurations DSML

flexibility in the creation of QoS requirements models and scalability of the models. The metamodels we describe below have been integrated with PICML using these associations, thus a single model of DRE system captures its entire QoS requirements specification.

As mentioned earlier for the scope of this chapter, QUICKER has been used in conjunction with PICML and therefore the discussion above pertains to associations of QUICKER QoS requirements with various CCM-specific functional entities. However its modeling abstractions are flexible enough to be extended for other component-oriented technologies.

Next we discuss the requirements specification across the following two RT QoS dimensions: (1) RT-CCM that is used to specify requirements for components and synchronous connections between components, and (2) RT publish/subscribe service that is used to specify requirements for asynchronous connections between components.

Real-time QoS requirements. Real-time requirements have component-level granularity. A `RTRequirement` element which is derived from `Requirement`, captures real-time requirements of a component and may have the following two attributes:(1) `fixed_priority_service_execution`, a server component Boolean property for specifying

whether or not it modifies client service invocation priorities, and (2) `bursty_client_requests`, a server component Boolean property for specifying the profile of service invocations made by its client components.

Publish/subscribe QoS requirements. We have modeled requirements for real-time publish/subscribe event service to enable specification of QoS across asynchronous and anonymous interactions in component-based DRE systems. In the context of a publish/subscribe service, a **Subscriber** component subscribes to receive events from a **Publisher** component that generates events. Publisher (subscriber) component connects to a mediator entity, an **Event Channel**, to publish (subscribe to) events.

The `ECRequirement` element is derived from `Requirement`. It models the properties of the event channel and can be used to specify the following QoS requirements: (1) `network_quality`, a connection-level property that captures the quality value of network used for running the application. (2) `connection_frequency`, a component-level property specifying the frequency at which the component (dis)connects with the publish/subscribe connection. (3) `event_distribution_ratio`, a connection-level property that specifies the ratio: $\frac{E^a{}_c}{E^s{}_c}$, where $E^a{}_c$ denotes number of events available for subscription at connection c and $E^s{}_c$ denotes average number of events subscribed to at connection c by each subscriber component. These modeling capabilities are at a sufficiently high level of abstraction and are intuitive to be applied to a variety of publish/subscribe mechanisms. All the requirements have an enumerated data type with values `LO` and `HI`.

II.3.2 Automating QoS requirements mapping using QUICKER

Challenge 2 and 3 in Section VI.2 motivated the need for an automated toolchain for performing QoS configuration of the underlying middleware platform. In this section, we first describe CQML QoS Configuration DSML that defines middleware-specific QoS options and outline our transformation algorithm that transforms system QoS requirements.

II.3.2.1 Modeling Middleware QoS options in CQML

Rather than directly transforming source models of DRE system into configuration descriptors required for deploying it on the middleware, we chose to generate models of middleware-specific QoS options from these source models such that they can be used for further analysis such as model-checking QoS properties of the DRE system. We have developed interpreters for parsing CQML system models and generating deployment descriptors in preparation of deploying the DRE system on target environment.

Real-time QoS options. As shown in Figure II.9, CQML defines the following elements corresponding to several RT-CCM configuration mechanisms: (1) `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure *static* thread (*i.e.*, those that remain active till the system is running and *dynamic* thread (*i.e.*, those threads that are created and destroyed as required) numbers using `Lane` element. (2) `ThreadPool`, which controls various settings of `Lane` elements, or a group thereof. These setting include `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and maximum resources assigned to buffer requests that cannot be immediately serviced. (3) `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is preserved, or SERVER_DECLARED if the server component changes the priority of invocation. (4) `BandedConnections`, which defines separate connections for individual (client) service invocations.

Publish/subscribe QoS options. For QoS configuration of asynchronous event communications, CQML defines the following elements: (1) `Publisher` and `Subscriber` modeling elements contain all the event source and sink settings, respectively. These include, for example, thread locks management mechanisms for publishers (subscribers) that are accessed by multi-threaded systems, and types of event filtering used, (2) `RTECFactory` element contains configurations specific to the event channel itself. These include, for example, event dispatching method that controls how events from publishers are forwarded

to the respective subscribers, scheduling of events for delivery and other scheduler-related coordination, and handling of timeout events in order to forward them to respective subscribers, and (3) `FilterGroup` element that specifies strategies to group more than one filters together for publishers (subscribers).

II.3.2.2 QUICKER Transformations for QoS Mapping

The QUICKER model transformation rules have been defined in GReAT and are based on our past experiences in configuring QoS for component-based DRE systems. They are applicable to any system model that conforms to the Requirements DSML, and thus can be used by the system developers repetitively during the development and/or maintenance phase(s) of the DRE system. QUICKER model transformations preserve the granularity specified in the source models.

Mapping real-time QoS requirements. Let ${R_p}^o$ and ${R_p}^i$ denote, respectively, the set of outgoing (required/event source) and incoming (provided/event sink) ports of component $p \in P$. Let S and C be the sets of server and client components respectively and are given by:

$p \in S \; if \; {R_p}^i \neq \emptyset$ and $p \in C \; if \; {R_p}^o \neq \emptyset$

Algorithm 1 describes (non-exhaustive) RT-CCM QoS mappings in QUICKER. Lines 5-13 show the thread resource allocation scheme for server components. For every incoming port of a server component, the number of interface operations and client components are counted (lines 9 and 10). These counts are used by the auxillary function $ThreadResources$ to calculate the total threads required for handling all client service invocations at that server.

For handling bursts of client requests, server components should configure their thread pool to grow dynamically such that threads are created only when required.$assignThreadResources$ function is used to adjust the ratio of static and dynamic threads for a server, depending on

Algorithm 1: Real-time QoS Requirements Mapping

Input: set of client components C, set of server components S, set of bursty client components B, set of threadPool lanes $TPLanes$

```
1  begin
2      InterfaceOperationsCount ioc; ClientsCount cc;
3      IncomingPort ip; OutgoingPort op;ThreadCount tc;
4      Component c;set of Components CPS; Buffering bf;
5      foreach p ∈ S do
6          ioc ← 0; cc ← 0; tc ← 0; bf ← false;
7          CPS ← ClientComponents(p);
8          foreach ip ∈ R_p^i do
9              ioc ← ioc + countOperations(p, ip);
10             cc ← cc + countClientComponents(p, ip);
11         end
12         tc ← ThreadResources(ioc, cc);
13         createTPLanes(p, tc);
14         foreach c ∈ CPS do
15             if c ∈ B then
16                 bf ← true;
17                 assignThreadResources(
18                 TPLanes_p, c, tc);
19             assignTPoolAttributes(TPLanes_p, bf);
20             ioc ← 0;
21             foreach op ∈ R_c^o do
22                 ioc ← ioc + countOperations(c, op);
23             end
24             createBands(c, ioc); matchPriorities(p, c);
25         end
26     end
27 end
```

whether its `bursty_client_requests` property is set to TRUE. In addition, lane borrowing feature at the server is set to TRUE such that the thread pool lanes across various priority levels can be borrowed. Finally, *PriorityBands* are configured and the their priority values are matched with server-side lane values in line 24.

Mapping publish/subscribe QoS requirements. Let PC_c^s denote the synchronization mechanism, PC_c^t denote the type, PC_c^i denote the iterator in proxy collection PC for component c, respectively. Let L_c denote the locking policy, CP_c denote control policy, SF_c

denote supplier-based filtering at component c, respectively. Algorithm 2 gives the (non-exhaustive) publish/subscribe QoS mappings.

Algorithm 2: Publish/Subscribe service QoS Requirements Mapping

Input: set of components CPS

```
1  begin
2      Component c; ThreadPoolLaneCount lc;
3      NetworkQuality nq;
4      foreach c ∈ CPS do
5          lc = countThreadResources(c); cf = connectionFrequency(c);
           nq = networkQuality(c); dr = eventDistributionRatio(c);
6          if lc ≠ 1 then
7              PC_c^s = MT; L_c = THREAD;
8          else
9              PC_c^s = ST; L_c = NULL;
10         end
11         if cf ≠ LO then
12             PC_c^t = LIST; PC_c^i = COPY_ON_READ;
13         else
14             PC_c^t = RB_TREE; PC_c^s = COPY_ON_WRITE;
15         end
16         if nq ≠ LO then
17             CP_c = NULL;
18         else
19             CP_c = REACTIVE;
20         end
21         if c ∈ S then
22             if dr ≠ LO then
23                 SF_c = PER_SUPPLIER;
24             else
25                 SF_c = NULL;
26             end
27     end
28 end
```

A publish/subscribe service has several settings for configuring the way collections of publisher and subscriber object references are created and accessed, which must be chosen appropriately for individual applications. Lines 6-9 in Algorithm 2 show how the choice

of serialization mechanism is affected by the number of thread resources configured at component c.

The choice of the *type* of collection is based on the following: (1) RB_TREE data structure exhibits faster ($O(log(n))$) insertion and removal operations. Therefore, it is more suited for connections whose components have a high (dis)connection rate; (2) LIST data structure on the other hand, should be chosen in cases where iteration is frequent (and therefore, more crucial for efficient application execution) than modifications to it.

Lines 11-14 give the steps in algorithm that configure the collection type. Finally, REACTIVE policy is chosen for applications that use low-quality value network on Lines 16-19, which ensures that (publisher/subscriber) components are periodically polled for determining their states (*i.e.*, whether or not they are connected to the event channel).

II.3.3 Applying QUICKER for Middleware QoS Configuration

The challenges described in Section VI.2 are resolved using QUICKER modeling and automated QoS configuration capabilities as follows:

Resolving Challenges 1 & 2: Target typed graph elements (*i.e.*, QoS options), are well-understood by implementation middleware experts. QUICKER transformation algorithms 1 and 2 are designed in terms of source and target typed graphs by these experts. System developers can describe their system QoS requirements using the modeling capabilities discussed in Section II.3.1.1. By providing platform-independent modeling elements in QUICKER and defining representational semantics that closely follow those of the system requirements, QUICKER allows system developers to describe system QoS using simple, intuitive notations. Further, model transformations defined in QUICKER automatically identify and deduce QoS configurations that are best suited to achieve the desired QoS for DRE systems being configured.

For example, in the MMS mission, QUICKER automatically identifies thread safety

mechanisms applicable for asynchronous connection between *Comm* and *Analysis* components as can be seen from lines 6-9 in Algorithm 2. In the SCE application, the requirement of prioritization of service invocations at *p1_A* component can be easily specified by setting `fixed_priority_service_execution` to TRUE.

Resolving Challenge 3: QUICKER transformation rules contain information about the semantics of the QoS options, their inter-dependencies, and how they affect the high-level QoS requirements of a DRE system and therefore are used to asssign values to the subset of options chosen earlier. Further, QoS options semantics are known precisely during transformations, and thus QUICKER ensures preservation of target typed graph semantics. Component interactions defined in input typed graph instance (*i.e.*, source model), along with the user-specified QoS requirements captured in that instance are used to completely generate an instance of the output graph.

For example, in SCE application, in addition to setting `fixed_priority_service_execution` to TRUE, recall from discussion in Challenge 2 in Section VI.2 that sufficient thread resources should also be configured to handle all client priority levels at *p1_A*. *ThreadResources* on line 12 in Algorithm 1 calculates appropriate number of thread resources as a function of client components of *p1_A* and their interface operations.

II.4 Evaluating GT-QMAP Toolchain for Middleware QoS Configuration

In this section we evaluate GT-QMAP modeling (*i.e.*, using its Requirements DSML) and transformation capabilitites in the context of DRE system case studies discussed in Section V.1. Class count metrics were used for evaluating modeling effort in using GT-QMAP. All the measurements use GME 6.11.9, GReAT 1.6.0 software packages on a Windows XP SP2 workstation. Our prototype implementation of GT-QMAP uses PICML and CQML from CoSMIC toolchain version 0.5.7.

CQML models represent detailed, middleware-specific DRE system QoS configurations that are used for generating configuration descriptors necessary for its deployment. In

order to find the reduction in modeling effort using GT-QMAP, we compare its (Requirements) modeling capabilities with those of CQML.

Class counts is an important metric for model-based quantitative software measurementsand has been applied and adopted in industrial contexts [16]. For our measurements, we use the following counts from the (meta)models: (1) modeling elements, which includes all the concrete modeling objects, (2) connections between modeling elements, (3) constraints that provide design-time type and/or dependency checks for enforcing language semantics, and (4) attributes of modeling elements. The counts were measured for both real-time and publish/subscribe QoS dimensions.

A comparison of CQML and GT-QMAP metamodels in terms of class counts given above is tabulated in Table II.3. The configuration space in this table simply refers to all of CQML's modeling elements each of which corresponds to RT-CCM and publish/subscribe options as explained in Section II.3.2.1. Using GT-QMAP, the number of modeling elements are reduced by an average of $\sim$58% while the number of attributes are reduced by an average of $\sim$81%.

The results from class count measurments for BasicSP, MMS and SCE application scenarios are shown in Table II.4. From these results it is observed that the modeling elements and number of attributes required for QoS specification for the publish/subscribe QoS dimension reduced by an average 54.55% and 76.4%, respectively. Reductions for RT-CCM were considerably higher *i.e.*, modeling elements reduction was 86.53% while number of attributes were reduced by 88.47%.

Connections defined in GT-QMAP are simple associations between modeling elements. For instance, recall from Section II.3.1.1 that real-time and publish/subscribe QoS requirement elements have component- or connection-level granularity. In contrast, modeling elements in CQML exhibit more complex dependency relationships. For example, *e.g.*, a REACTIVE event dispatching method at an event channel necessitates that `ProxyCollection` at corresponding publisher and subscriber components be either MT or ST.

Table II.3: Comparing Requirements DSML against configuration space

Effort measured on	# of modeling elements	# of connections	# of constraints	# of Boolean	# of int/long	# of string	# of enum
CQML							
publish/subscribe	9	3	7	0	3	1	18
RT-CCM	6	2	3	2	9	2	1
GT-QMAP							
publish/subscribe	4	1	0	1	0	0	4
RT-CCM	1	1	0	2	0	0	0

It is easier to evolve DRE system QoS using GT-QMAP owing to its automated requirements mapping capabilities. For example, an additional requirement in the SCE scenario during its development cycle necessitates that similar to *p1_A*, component *ec_A* must prioritize its service invocations. In GT-QMAP this is achieved simply by setting `fixed_priority_service_execution` property at *ec_A* (to TRUE). For the entire SCE application since it contains 10 such application strings (and therefore, 10 *ec_A* components), this additional requirement requires modification of 10 attributes in its GT-QMAP model. In CQML, on the other hand, such an additional requirement would require the following modifications: (1) Configuring the *PriorityModelPolicy* to SERVER_DECLARED, and assigning sufficient *Lanes* at *ec_A* for handling all of its client service invocations. (2) Assigning *PriorityBands* at client components (*e1_A* etc.) such that a separate connection is used for each request priority level. This configuration further requires that these band priority values match with some lane values at *ec_A* component. Even if smallest possible number of *Bands* and *Lanes* are chosen at respective components, this requires specifying $\sim$10 modeling elements, $\sim$4 connections, and $\sim$16 attributes for each of the 10 application strings in SCE.

Qualitative Modeling Effort using GT-QMAP. Extensive user studies would be required for measuring qualitative value of GT-QMAP's modeling capabilities, however, we argue

Table II.4: Reduction in modeling effort using GT-QMAP

Effort measured on	# of modeling elements	# of conns.	# of constr.	# of Bool.	# of int/long	# of string	# of enum
BasicSP in CQML							
publish/subscribe	27	9	21	0	3	1	54
RT-CCM	18	9	9	4	28	6	2
BasicSP in GT-QMAP							
publish/subscribe	12	3	0	3	0	0	12
RT-CCM	3	3	0	4	0	0	0
MMS in CQML							
publish/subscribe	101	35	77	0	35	11	210
RT-CCM	87	44	21	14	163	20	7
MMS in GT-QMAP							
publish/subscribe	46	11	0	11	0	0	46
RT-CCM	10	10	0	18	0	0	0
SCE in CQML							
publish/subscribe	1100	390	840	0	390	120	2270
RT-CCM	980	390	360	240	1000	160	120
SCE in GT-QMAP							
publish/subscribe	510	120	0	120	0	0	510
RT-CCM	120	120	0	240	0	0	0

that in general, the platform-independent QoS specification in GT-QMAP is qualitatively better than CQML owing to the following observations about the two languages:

1. Apart from the use of domain-level abstractions that naturally lead to simpler system modeling, all the attributes defined in GT-QMAP are either Boolean or enumerated data types with two values. Modeling in GT-QMAP is thus similar to answering a set of questions about QoS requirements of the application.
2. No explicit type checking constraints are defined in GT-QMAP, as opposed to CQML which defines a total of 10 constraints as shown in Table II.3. This feature allows easier QoS specification since the language semantics are simpler, and is extremely useful as DRE systems QoS configurations change during its development/maintenance cycle.

CHAPTER III

ON THE CORRECTNESS OF QUICKER TRANSFORMATIONS

The success of component middleware technologies like Enterprise Java Beans (EJB) and CORBA Component Model (CCM) has raised the level of abstraction used to develop software for distributed real-time and embedded (DRE) systems, such as avionics mission-computing and shipboard computing systems. As a result, commercial-off-the-shelf (COTS) middleware, such as application servers and object request brokers (ORBs), now provides out-of-the-box support for traditional concerns affecting QoS in DRE system development, including multi-threading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms. This support helps decouple application logic from QoS mechanisms (such as portable priority mapping, end-to-end priority propagation, thread pools, distributable threads and schedulers, request buffering, and managing event subscriptions and event delivery necessary to support the traditional concerns listed above), shields the developers from low-level OS specific details, and promotes more effective reuse of such mechanisms.

Contemporary component middleware technologies, such as Enterprise Java Beans (EJB) and CORBA Component Model (CCM), have helped to decouple application logic from the quality of service (QoS) configuration of distributed real-time and embedded (DRE) systems by moving the QoS configuration activity to the middleware platforms that host these systems. Middleware provides out-of-the-box support for traditional concerns affecting QoS in DRE systems including multi-threading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms.

Although this component middleware simplifies application logic, the DRE system developers are now faced with the complexities of choosing the right set of middleware configuration parameters that meet the QoS demands of their systems. This problem is particularly pronounced in general-purpose middleware platforms, such as CCM and EJB, which are designed to be highly flexible and configurable to meet the needs of a large class of distributed systems.

Prior research on software processes and artifacts for QoS management in DRE systems have focused on different dimensions of the problem space. For example, configuration, analysis, optimization and adaptation techniques [76, 127] allow allocation and dynamic QoS adaptation such that end-to-end application goals are met. Another online approach [131] applies feedback control theory in conjunction with application monitors for affecting resource allocation. Several works for schedulability and timing analysis [66, 116], and behavioral analysis and verification [45] exist in the literature for calculating the exact priority schemes, time periods, and resolving functional dependencies. These related approaches often provide either runtime solutions for QoS management or their outcomes are independent of any middleware platforms and hence must be mapped onto middleware configuration options via another technique. It is only recently that design-time techniques are starting to address [56, 72, 133] some of the challenges of middleware configurations, which includes support for configuring low-level QoS properties of the middleware platform and generating test suites for benchmarking, among others.

Despite these recent research efforts in addressing the middleware configuration problem for DRE systems, techniques to evaluate the correctness of these software processes and validating their effectiveness in meeting system QoS objectives remains largely unaddressed to date. This chapter focuses on this unresolved dimension of the problem space. We demonstrate our ideas on our earlier work on a design-time process for middleware QoS configuration, which includes the QUICKER [59] model-driven engineering (MDE) toolchain and its QoS mapping algorithms [56] that use graph transformations.

In this chapter we verify the correctness of our QoS configuration process and validate its effectiveness in meeting the QoS requirements of DRE systems. To this end, we use *structural correspondence* between source and target modeling languages in QUICKER to verify the correctness of their mapping. Further, we show how the Bogor [101] model-checking tool can be seamlessly extended to employ real-time CCM (RT-CCM)-specific language constructs to ascertain that the generated configurations are appropriate. We subsequently apply our configuration in the context of a representative DRE system case study and empirically evaluate the generated QoS configurations to show how the QoS requirements are met.

Chapter Organization. The remainder of this chapter is organized as follows: Section III.1 gives a brief overview of our configuration process and the input and output languages used in its QoS mapping algorithms; Section V.3 verifies the correctness and empirically evaluates our configuration process in the context of a DRE system case study.

III.1 Overview of middleware QoS configuration process

Figure III.1 shows our overall approach (for details please see [56]). DRE system developers use the *Requirements* domain-specific modeling language (DSML)/metamodel to specify the system QoS requirements. Using our configuration process, a specification of system QoS requirements acts as the source model of QoS mapping algorithms. As can be seen in Figure III.1, our process uses model transformations for achieving QoS mapping. Middleware-specific QoS configuration options are captured as models using the *QoS Configurations* DSML which serves as the target model in the transformation process.

We have used the Generic Modeling Environment (GME) [2] toolkit for developing these source and target languages, which provides a graphical user interface that can be used to define both language semantics and system models that conform to the languages defined in it. The model-to-model transformations, on the other hand, have been developed

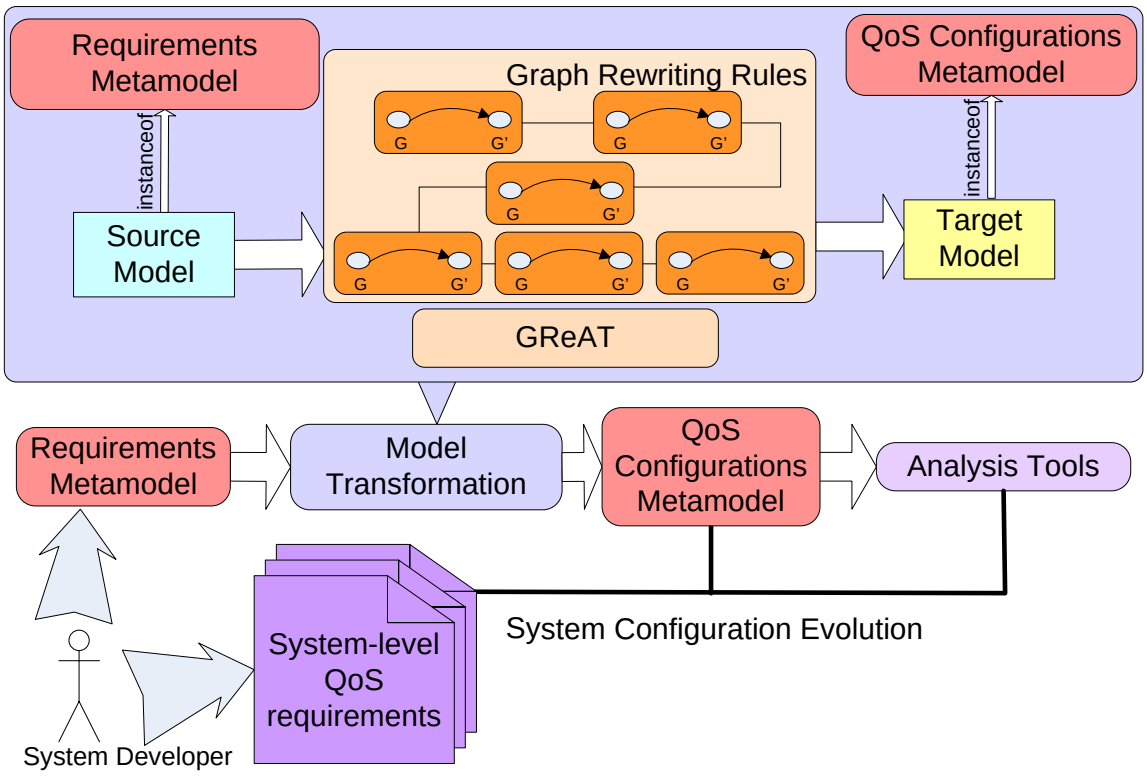


Figure III.1: Model-driven QoS configuration process

using the Graph Rewriting And Transformation (GReAT) [54]. GReAT, which is implemented within the framework of GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine).

In our configuration process developers specify QoS using the requirements language. Our QoS mapping algorithms are codified as GReAT transformation rules which use the modeled system structure and platform-specific heuristics for automatically translating the system requirements to detailed QoS configuration models. This translation is an example of a *vertical exogenous* transformation [83] that starts with an abstract type graph as the input and refines the graph to generate a more detailed type graph as the output. Finally, the generated configuration models are used for synthesizing (1) descriptors necessary for configuring functional and QoS properties of DRE system in preparation for its deployment on target platform, and (2) input to external model-checking tool for further analysis.

In our configuration process, modeling real-time requirements is simple and involves specifying the following two component-level Boolean attributes: (1) `fixed_priority_service_execution` that indicates whether or not the component modifies client service invocation priorities, and (2) `bursty_client_requests` that allows specification of the profile of service invocations made by its client components *i.e.*, whether the invocations are bursty or periodic in nature.

Figure II.9 shows the QoS configurations metamodel which defines the following elements corresponding to several RT-CCM configuration mechanisms: (1) `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. Threads can be *static* or *dynamic* depending on their state with respect to the application execution lifecycle; (2) `ThreadPool`, which controls various settings of `Lane` elements including `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and resource limits for buffering requests that cannot be immediately serviced; (3) `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is to be preserved end-to-end, or SERVER_DECLARED if the server component changes the priority of invocation; and (4) `BandedConnections`, which defines separate connections for individual service invocations to avoid head-of-line blocking of high priority packets by low priority packets.

For a detailed discussion of our QoS mapping transformation algorithms, we refer the reader to [56]. In the next section, we evaluate our configuration process by applying it in the context of a representative DRE system case study.

III.2 Evaluation of QoS configuration process

This section evaluates our configuration process to verify the correctness of its transformation algorithms and validate its effectiveness in meeting the QoS requirements of DRE

systems. First we present a representative DRE system case study we used for the evaluation. Next we discuss our structural correspondence technique for proving that the input and output models of transformations used in our process are correctly mapped. We then describe how we have applied Bogor model-checking tool for verification of the generated configurations. Finally, through empirical evaluation, we validate the generated QoS options.

III.2.1 DRE System Case Study

The Basic Single Processor (BasicSP) is a scenario from the Boeing Bold Stroke component avionics computing product line. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed using a QoS-enabled component middleware platform. The application is deployed using a single deployment plan on two physical nodes. s A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario shown in Figure IV.1 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot. In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Airframe* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of

the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space. For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is RT-CCM) ensure that these application level QoS requirements are satisfied. In the remainder of the chapter, we focus on verification and validation of the QoS options generated using our approach.

III.2.2 Verifying the correctness of our QoS configuration process

Verifying our model-based configuration process entails verification of correctness properties across the following two dimensions: (1) Correctness of QoS mapping algorithms *i.e.*, QoS options generated are equivalent to the QoS requirements from which these options are mapped. In our case, this translates to verification of the QoS mapping/transformations used. (2) Correctness of the generated QoS options themselves *i.e.*, whether individual values of these options are appropriate locally (*e.g.*, for a component) as well as globally (*e.g.*, for all dependent components). This section discusses verification of our process across the above two dimensions.

III.2.2.1 Assuring the correctness of QoS mapping algorithms

To provide an assurance that the QoS requirements specifications were correctly mapped into the QoS configuration model, we have used the transformation verification technique described in [88]. The source and target portions of the transformation are treated as typed, attributed graphs, and the correctness of the transformation is specified as a relation between these graphs. Such a relation, called a structural correspondence, is specified by

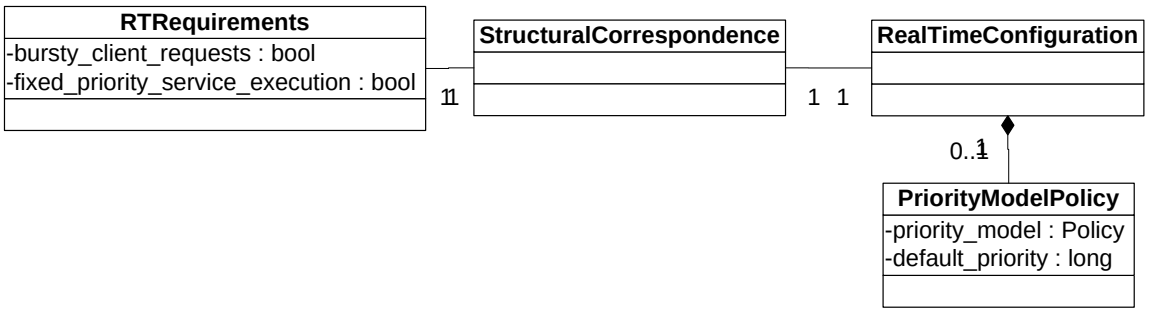


Figure III.2: Structural correspondence using cross-links

identifying pivot nodes in the metamodel and specifying what constitutes a correct transformation for these nodes.

Using structural correspondence, the verification consists of two phases: the specification of the correctness conditions, and the evaluation of the correctness. In the first phase, we identify important points in the transformation, and specify the structural correspondence rules for these points. From these rules, a model traverser is automatically generated, which will traverse and evaluate the correspondence rules on the instance models. This step needs to be performed only once. The second phase involves invoking the model traverser after each execution of the model transformation. In this phase, the model instance being transformed is traversed, and the structural correspondence rules are evaluated at each relevant node. If any of the rules are not satisfied, it indicates that the model has not been transformed satisfactorily.

Structural correspondence rules are described using (1) specification of the correspondence condition itself, and (2) the rule path expressions, which are similar to XPath queries. Figure III.2 shows how we have used cross-links in GReAT as means of specifying the correspondence condition between input and output language objects such that their equivalence can later be established. `RTRequirements` is an input language object that denotes real-time requirement specification for a component. It has a correspondence relation with `RealTimeConfiguration` output language object, indicated by presence of a cross-link between them in Figure III.2.

Additionally, one of the transformation rules in our QoS mapping algorithms states that if the Boolean attribute `fixed_priority_service_execution` of `RTRequirements` is set to TRUE in the input model, then `priority_model` attribute of `PriorityModelPolicy` object be set to SERVER_DECLARED in the output model. Otherwise `priority_model` should be set to CLIENT_PROPAGATED. Additionally, if `priority_model` is set to SERVER_DECLARED for a component, `Lane` values at that component and `BandedConnection` values at its clients must match. In order to complete the correspondence rule specification, the above is encoded as a rule path expression as follows:

```
(RTRequirement.
fixed_priority_service_execution =true ∧
(∀b ∈ RTConfiguration. BandedConnection
∃l ∈ RTConfiguration. Lanes :
    (b.low_range ≤ l.priority ≤ b.high_range)) ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "SERVER_DECLARED") ∨
(RTRequirement.
fixed_priority_service_execution = false ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "CLIENT_PROPOGATED")
```

If this expression evaluates to TRUE on an instance model, then it implies that the QoS configuration for this particular property has been mapped correctly. This applies to the `RTRequirements` and `RealTimeConfiguration` classes, and correspondence condition is added as a link between these classes in the metamodel. Similar to correspondence condition between `RTRequirements` and `RealTimeConfiguration` we described, other conditions for each of the QoS mapping rules have been specified ensuring that the transformation is verified correct if all these conditions are satisfied.

III.2.2.2 Verifying the generated QoS configurations using model-checking

This section illustrates how the correctness of QoS configuration mappings is verified using the Bogor model-checking framework, which is a customizable explicit-state model checker implemented as an Eclipse plugin. Verifying a system using Bogor involves defining (1) a model of the system using the *Bogor Input Representation* (BIR) language and (2) the *property* (*i.e.*, specification) that the model is expected to satisfy. Bogor then traverses the system model and checks whether or not the property holds. To validate QoS configuration options of an application using Bogor, we need to specify the application model and its QoS configurations. We use Bogor's extension features to customize the model-checker for resolving the QoS configuration challenges for component-based applications.

It is cumbersome to describe middleware QoS configuration options using the default input specification capabilities of BIR. This is because such a representation is at a much lower level of abstraction compared to domain-level concepts, such as components and QoS options, which we want to model-check. Additionally, specifying middleware QoS configuration options using BIR's low-level constructs can yield an unmanageably large state space since representing domain-level concepts with a low-level BIR specification requires additional auxiliary states that may be irrelevant to the properties being model-checked [101]. Therefore we have defined composite language constructs that represent functional sub-systems (such as components) and QoS options (such as thread pools) as though they were native BIR constructs.

Listing 1 shows an example of our QoS extensions in Bogor to represent QoS configuration options in middleware, which define two new data types: `Component`, which corresponds to a CCM component, and `QoSOptions`, which captures QoS configuration options, such as `lane`, `band`, and `threadpool`.

In addition to defining constructs that represent domain concepts, such as components and QoS options, we also need to specify the *property* that the application should satisfy. In our case, property simply means whether or not the QoS configurations are verified correct.

```
extension QoSOptions for
edu.ksu.cis.bogor.module.QoSOptions.QoSOptionsModule
{
  // Defines the new type to be used for
  typedef lane;
  typedef band;
  typedef threadpool;
  typedef prioritymodel;
  typedef policy;
  // Lane constructor.
  expdef QoSOptions.lane createLane (
   int static, int priority, int dynamic);
  // ThreadPool constructor.
  expdef QoSOptions.threadpool
  createThreadPool (boolean allowreqbuffering,
   int maxbufferedrequests, int stacksize, int
   maxbuffersize, boolean allowborrowing);
  // Set the band(s) for QoS policy.
  actiondef registerBands (QoSOptions.policy
   policy, QoSOptions.band ...);
  // Set the lane(s) for QoS policy.
  actiondef registerLanes (QoSOptions.policy
   policy, QoSOptions.lane ...);
  ...
}
extension Quicker for
edu.ksu.cis.bogor.module.Quicker
{
  // Defines the new type.
  typedef Component;
  // Component Constructor.
  expdef Quicker.Component
  createComponent (string component);
  // Set the QoS policy for the component.
  actiondef registerQoSOptions (Quicker.Component
    component,QoSOptions.policy policy);
  // Make connections between components.
  actiondef connectComponents (Quicker.Component
    server,Quicker.Component client);
  ...
}
```

Listing 1: QUICKER BIR Extension

Thus, since we need to verify values of various QoS options as means to check whether application property is satisfied, we define *rules* that capture values of these QoS options. BIR primitives are used to express these rules in the input specification of DRE system. Primitives are also used to capture component interconnections in BIR format which are required for populating the dependency structure for the specified input application. They are also used later during verification of options for connected components.

QoS extensions are also helpful in maintaining and resolving dependencies between application components. For example, consider a real-time configuration of BasicSP scenario

in which each of the *GPS*, *AirFrame*, and *NavDisplay* components are configured to have `priority bands` for separate service invocation priorities and the *Timer* component is configured to support multiple priority levels during generation of pulse events. Given such a configuration, we have that `priority band` values at *GPS* (client) component must match `ThreadPoolLanes` at *Timer* (server) component *i.e.*, a direct configuration dependency exists between these two components.

Further, since the pulse events are subsequently reported to *AirFrame* and *NavDisplay* components there is a similar indirect dependency between `band` values at these components and `lanes` at *Timer* component. The dependency structure of BasicSP scenario is maintained in QoS extensions to track such dependencies between QoS options. Figure III.3 represents the dependency structure generated using QoS extensions with the given configurations for our BasicSP scenario. Occurrences of change in configurations of either of the dependent components are followed by detection of potential mismatches such that all dependencies are exposed and resolved during application QoS design iterations.

Applications that need to be model-checked by Bogor must be represented in BIR format. Writing and maintaining BIR manually can be tedious and error-prone for domain experts (*e.g.*, avionics engineers) since configuring application QoS policies is typically done iteratively. Depending on the number of components and available configuration options, manual processes do not scale well.

To automate the process of creating BIR specification of applications, we therefore used the generative capabilities in GME to automatically generate BIR specification of an application from its QoS configurations model. This generative process is done in GME using a model interpreter that traverses the QoS configurations model and generates a BIR file that captures the application structure and its QoS properties. Our toolchain therefore automates the entire process of mapping application QoS policies to middleware QoS options, as well as converting these QoS options into BIR. A second model interpreter is used to generate the Real-time CCM-specific descriptors required to configure functional and QoS

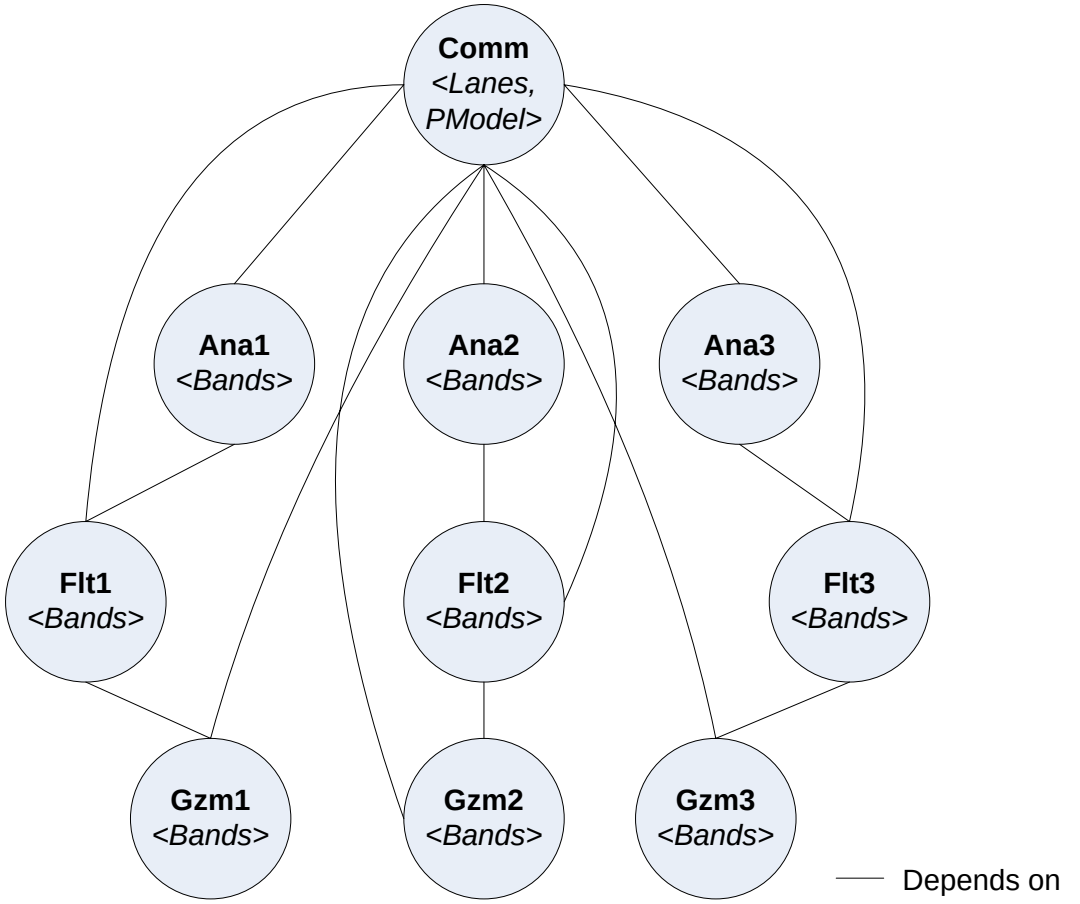


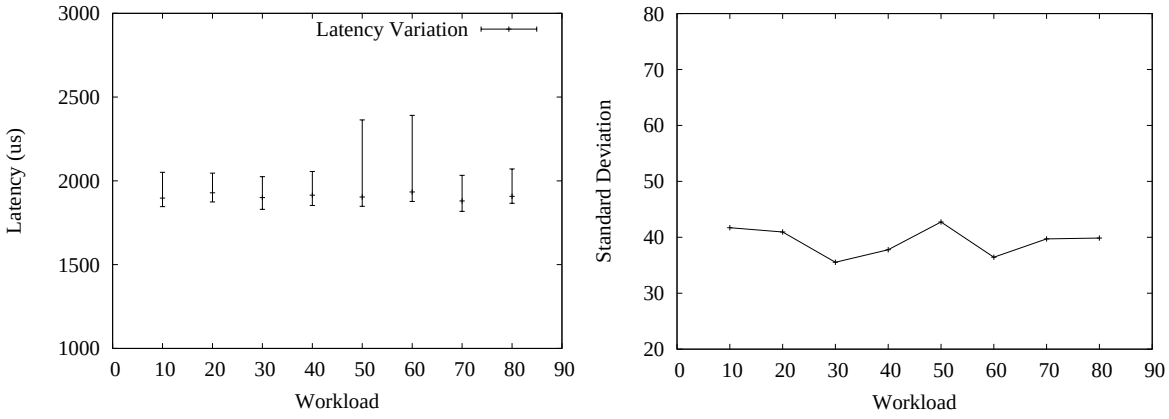
Figure III.3: Dependency structure of BasicSP. L_c denotes threadpool lane and B_c denotes priority bands at component c. SD and CP indicate the SERVER_DECLARED and CLIENT_PROPAGATED priority models, respectively.

properties of an application and deploy it in its target environment. In the next section we empirically validate these generated QoS configurations.

III.2.3 Empirically evaluating BasicSP QoS configurations

In this section we empirically validate the effectiveness of the generated QoS configurations for the BasicSP case study.

Experiment Configuration. We have used ISISLab (www.dre.vanderbilt.edu/ISISlab) for evaluating observed QoS properties of DRE systems based on middleware QoS configurations generated using our configuration process. Each of the physical nodes



(a) End-to-end latency measurements (b) Statistical dispersion of latency

Figure III.4: Evaluating BasicSP QoS configurations against increasing workload at a constant 20Hz invocation rate.

used in our experiments was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz network interface, and 40GB hard disks. Version 0.6 of our RT-CCM middleware CIAO was used running on Redhat Fedora Core release 4 with real-time preemption patches. The processes that hosted BasicSP components were run in the POSIX scheduling class SCHED_FIFO, enabling first-in-first-out scheduling semantics based on the priority of the process.

As the first step, we modeled BasicSP QoS requirements using the requirements DSML described in Section III.1. `bursty_client_requests` was set to FALSE for all components and `fixed_priority_service_execution` attribute was set to FALSE for every component except *Timer*. Secondly, we applied our model transformation algorithm to the requirements model above for generating detailed application configurations. Table III.1 captures some of the important QoS configurations generated in our process. These configurations are represented as an application model. In the final step, we apply model interpreters for synthesizing descriptors required to configure the functional and QoS properties of the application during deployment.

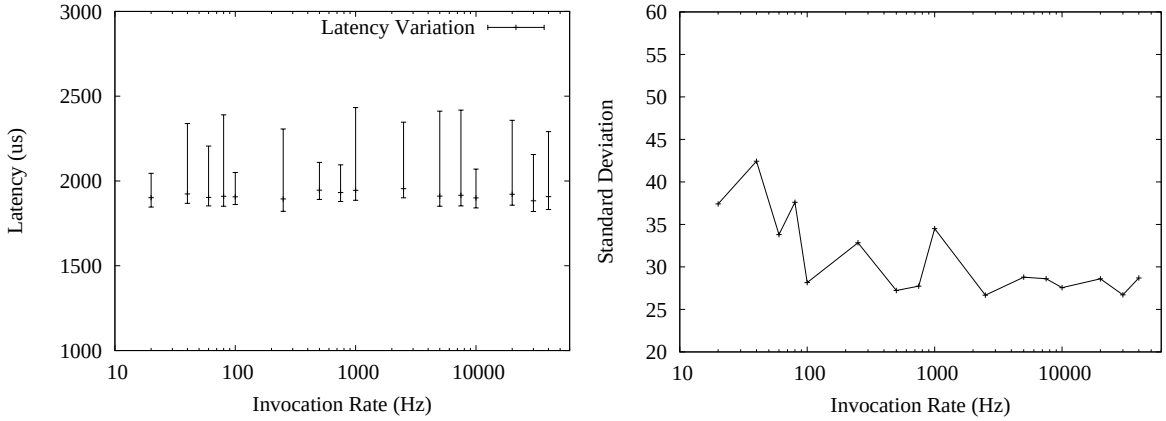
In evaluating effectiveness of our configuration process, we collected end-to-end latency measurements between *Timer* and *NavDisplay* components. Earlier in Section III.2.2.2

Table III.1: Generated QoS Configuration for BasicSP

QoS configuration	Timer	GPS	Airframe
PriorityModel	SD	CP	CP
ThreadPool			
stacksize	1024	1024	1024
max_buff_reqs.	–	20	20
allow_borro.	FALSE	FALSE	FALSE
allow_req_buff	FALSE	TRUE	TRUE
Lane			
static_thrds	4	8	8
dyna_thrds	0	0	0

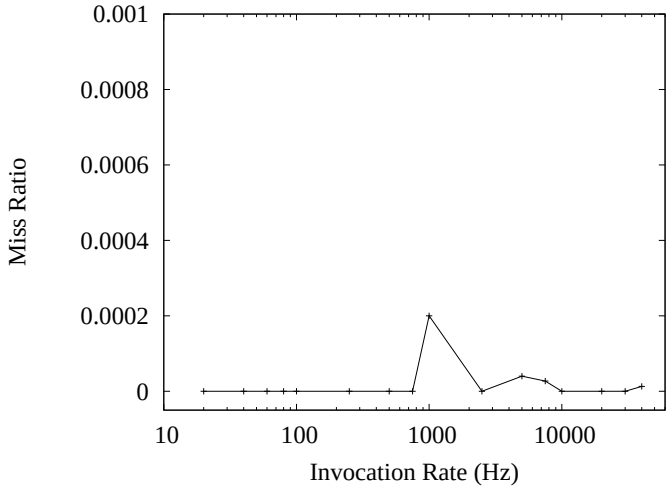
we discussed how correctness of QoS options can be verified using our process, the first experiment discussed below empirically evaluates the effectiveness of these options in meeting 20Hz operational display refresh rate of BasicSP from low to high workload conditions. Further, operational conditions of DRE system might change (unfavorably) during its execution. In order to evaluate the tolerance of our generated configurations under such conditions, in the second experiment we measure the metrics discussed above when invocation rate is steadily increased. Each of these experiments were performed for a constant time period and after executing 10,000 warmup iterations.

Experiment 1: Increasing System Workload. Figure III.4 plots the latency measurements under increasing system workload. The workload is characterized [98] as a function performed with every client invocation. The signature of the function is given as: $void\ \ work(int\ units)$; where *units* argument specifies the amount of processor intensive work performed per call. The experiment was run for workload values of 10 through 80. End-to-end latency was observed to be at an average value of $\sim$1925 as can be seen in Figure III.4a. Further, successive event-driven computations in the scenario exhibit an almost constant time complexity, indicated by relatively small dispersion in latencies as plotted in Figure III.4b.



(a) End-to-end latency measurements

(b) Statistical dispersion of latency



(c) Missed deadlines

Figure III.5: Evaluating BasicSP QoS configurations against increasing invocation rate: All the plots use logarithmic X axis and linear Y axis.

Experiment 2: Increasing System Invocation Rate. Performance of the generated configurations for BasicSP is given in Figure III.5. Throughout this experiment the rate of invocation was increased from a normal operational value of 20Hz to a maximum of 40000Hz. Latency results are shown in Figure III.5a which plots maximum, mean and minimum delay measurements for each invocation rate data point. Even with increasing rate the mean latency did not change significantly and was observed to be consistently just above 1900 μs for the entire range of invocation rates. Note that this is a desirable characteristic since even with an unfavorable change in operational conditions (*i.e.*, change in invocation rate)

the latency was observed to be constant. Jitter in latencies for each invocation rate is plotted in Figure III.5b which shows that the deviation is bound between a high value of 42.44 (at 40Hz) and low value of 26.68 (at 2500Hz). At frequencies 2500Hz and higher the jitter values became quite stable showing a maximum variation of only 2.11. Overall, our results indicate that even under increased rate of invocation, the configurations perform effectively in achieving BasicSP latency requirements.

CHAPTER IV

OPTIMIZATION OF QUICKER-GENERATED QOS CONFIGURATIONS

The component-based programming, design, and configuration paradigm has received much attention over the past few years to develop distributed, real-time, and embedded (DRE) systems. DRE systems have stringent quality of service (QoS) requirements that must be satisfied by their resource constrained runtime environments. Examples of DRE systems include emergency response systems, aircraft navigation and command supervisory systems, and total shipboard computing systems. With a simplified programming model and mechanisms to separate functional and non-functional aspects of the system being designed, lends the component-based paradigm to rapid prototyping, (re-) configuration, and easier maintenance of DRE systems.

Deployment of application functionality (*i.e.*, allocation of CPU and other resources to components based on their QoS requirements and resource availability) and configuration of the infrastructure (*i.e.*, choosing the right configuration options of the middleware) play a key role in realizing the QoS requirements of DRE systems. To that end, middleware that provides the component-based programming paradigm to DRE systems, such as Lightweight CORBA Component Model (LwCCM) [92], have been designed to be flexible and configurable such that they can be manipulated and optimized during system development for individual application domains.

Specifically, component middleware provide a large variety of mechanisms, such as containers to host components that (1) allow choosing the number of thread resources to be configured for each component, their type (*i.e.*, static or dynamic), and properties such as stacksize, etc., (2) control over asynchronous event communication, and the publisher/subscriber event filtering and delivery options, and (3) set the client request invocation priorities on the server component. The *configuration space* – identified by all the

mechanisms for specifying system QoS and their appropriate values – becomes highly complex due in large part to the generality of these middleware platforms.

Figure I.3 provides an overview of existing research in QoS configuration and deployment for DRE systems, which so far has focused on: (1) application functional specification, decomposition and analysis [45] to capture and study application structure and behavior, which is then realized as components of DRE systems that can be deployed, (2) QoS analysis, optimization and adaptation [76, 135], allocation of resources to applications, and node placement [25], (3) QoS specification languages [31, 100, 128] for capturing application QoS properties, which are helpful in determining the configuration parameters for the middleware, and (4) schedulability and timing analysis [42, 116] to determine exact priorities and time periods for applications, which help in partitioning the system resources and configuring the middleware.

Despite these advances, there still remain unresolved challenges in the effective deployment and configuration of DRE systems. For example, although bin packing algorithms [25] make effective decisions on component deployment, these decisions are at best coarse grained since they determine only the nodes on which the components must be deployed but do not indicate how they are deployed within the containers of the component middleware. These limitations may lead to suboptimal performance since these decisions are now left to application developers.

On the other hand, research efforts, such as the Physical Assembly Mapper (PAM) [7], demonstrate how time and space overheads can be reduced by optimizing (*i.e.*, merging) collocated components at system deployment-time. However, these benefits can be realized only if the identities of the collocated components is known *a priori*. Unfortunately, these decisions too are left to the developer.

Chapter II dealt with automating the process of mapping QoS configuration options from higher level DRE system QoS requirements. In particular, we described the use of

graph transformations [102, 108] to automate the translation of domain-specific, platform-independent models of DRE system QoS requirements to more refined and detailed middleware platform-specific models. However, these research efforts in QoS configuration, including ours, have so far been largely restricted to specification, analysis, and mapping techniques in the configuration space.

To address existing gaps in deployment and configuration for DRE systems, this chapter presents a model transformation-based approach to optimize QoS configuration in component-based DRE systems. Our approach optimizes QoS configuration of component-based applications by combining (a) our analysis of the configuration space design decisions based on higher level application requirements with (b) heuristics based on its allocated computing resources and deployment plans[1]. Our research prototype has been implemented for the LwCCM middleware, and it encapsulates the platform-specific rules that can be used to optimize the DRE system QoS configuration. These rules are encoded as model transformation rules and thus can be reapplied repeatedly during DRE system development. We present the design of our approach and the process in using it to achieve QoS configuration for a representative component-based DRE system. We also discuss our results in applying our technique to a representative application and compare them with the existing state-of-the-art.

This chapter is organized as follows: Section IV.1 discusses the important sources of gratuitous overheads in component-based application configuration, and the challenges developers face in achieving optimal QoS configuration for such systems; Section IV.2 presents the overall approach taken, the enabling technologies used in our technique, and the model transformation algorithm we have developed; Section IV.3 discusses a representative case study, and also empirically evaluates our approach.

[1]Deployment plan identifies the deployment specification capturing how component to node mapping occurs for an application at runtime.

IV.1 Challenges in Optimizing QoS Configurations

We now present the challenges in bridging the gap in current deployment and configuration techniques so that the resulting QoS configurations are optimal for component-based applications. We focus on issues that are both innate to the middleware platforms as well as those that are accidental.

Challenge 1: Inherent challenges in QoS Policy Configuration. In the context of component middleware platforms, such as LwCCM, a container is an execution environment provided for hosting the components such that they can access the capabilities of the hardware, networking and software (OS and middleware) resources. In particular, containers act as a higher-level of abstraction for hosting the components in which all the developer-specified QoS policies can be properly configured. Components with similar QoS configuration specifications are hosted within the same container so that all components in that container obtain the same QoS capabilities.

Service request invocations between the components hosted on different containers, despite being on the same host and process, have to go through the typical request demultiplexing layers and marshaling/demarshaling and mechanisms. Therefore, such invocations are considerably slower than the invocations between components that share the same container [126].

A number of analyses of DRE systems (*e.g.* application schedulability and timing analysis, and component priority analysis) and deployment and resource allocation decisions (*e.g.*, where each component resides in the available computing node farm) affect the QoS configurations chosen for individual components of the application. For example, LwCCM defines QoS policies such as allocation of priorities to every component, and fixed/variable priority request invocation (`PriorityModelPolicy`), mapping component priority with the execution platform priority (`PriorityMapping`), number of thread resources, their type (*i.e.*, static or dynamic), and concurrency characteristics (`ThreadPool`).

For a component-based application, the mapping of the above analyses onto these available policies results in a number of unique QoS configurations, and naturally, as many containers. Thus, in effect, components placed on different containers (which are in turn created from unique QoS policies) are unable to exploit the collocation optimizations performed by the middleware. As such, the sub-optimal QoS configuration of the application leads to increased average end-to-end latencies. As the number of components in the system that are sub-optimally configured increases, the adverse impact on end-to-end latencies can significant.

Challenge 2: Accidental Complexities in QoS Policy Configuration. The developers can keep track of the QoS policies created, and depending on their application can potentially choose to combine these policies. Such a manual approach introduces several non-trivial challenges for the application developers:

1. Large-scale applications typically consist of hundreds of components spanning across multiple assemblies. Manually keeping track of all the policies (and potentially combining them) in such large-scale applications is very difficult and in some cases infeasible.
2. Development of DRE systems is often an iterative process where new requirements are added. Thus, the application configuration needs to evolve accordingly to cater to new requirements, and the optimizations listed above need to be performed at the end of each reconfiguration cycle.
3. The configuration optimization activity forces the developers to have a detailed knowledge of the middleware platform. Further, the activity itself is not central to the development of application logic and may in fact be counter-productive.

In conclusion, it is essential to design and develop automated tools and techniques to perform the QoS configuration optimization process in component-based DRE systems. The rest of the chapter discusses our solution approach that addresses the challenges discussed in this section.

IV.2 Optimizing QoS Configuration for Component-based Systems

We now present our model transformation-based approach to QoS configurations for component-based DRE systems. The model transformation algorithm in our approach takes the following as its input: (1) DRE QoS configuration specification, and (2) DRE system deployment plan indicating coarse-grained component collocation groups (*i.e.*, components that can be placed together on the same process). Its output is a new QoS policy set[2], which is incorporated into the DRE system model. Our approach produces optimized QoS policy set by employing novel ways of reusing and/or combining existing configurations based on deployment heuristics in an application-specific manner.

We have used the Generic Modeling Environment (GME) [2] toolkit for developing the modeling languages used. GME provides a graphical user interface that can be used to define both modeling language semantics and system models that conform to the languages defined in it. *Model interpreters* can be developed using the generative capabilities in GME that parse and can be used to generate deployment, and configuration artifacts for the modeled application.

We have used the Graph Rewriting And Transformation (GReAT) [54] language for defining our algorithms. GReAT, which is developed using GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine). The graph rewriting rules are defined in GReAT in terms of source and target languages (*i.e.*, meta-models).

DRE system developers use GME to model their application, specify the QoS, and deployment files. The GReAT toolchain is later used for applying our model transformation algorithm to the DRE system model such that its QoS configuration is updated (*i.e.*, optimized). We explain each of these steps in more details.

[2]QoS policy set is a group of configuration files that completely capture the application QoS. These files are used by the middleware to ultimately provision infrastructure resources such that QoS can be met.

IV.2.1 Step I: Modeling Language used in the Transformation Algorithm

To demonstrate our technique we require a modeling language for component-based DRE systems. To that end we leverage our Component QoS Modeling Language (CQML) [59] for modeling QoS configurations and deployment plans of a DRE system. A simplified UML QoS configuration metamodel in CQML is shown in Figure II.9.

As shown, CQML defines the following elements corresponding to several LwCCM real-time configuration mechanisms:

1. `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure the number of *static* threads (*i.e.*, those that remain active till the system is running, and *dynamic* threads (*i.e.*, those threads that are created and destroyed as required) using `Lane` element.
2. `ThreadPool`, which controls various settings of `Lane` elements, or a group thereof. These settings include `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and maximum resources assigned to the buffer requests that cannot be immediately serviced.
3. `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is preserved, or SERVER_DECLARED if the server component changes the priority of invocation.
4. `BandedConnections`, which defines separate connections for individual (client) service invocations. Thus, using `BandedConnections`, it is possible to define a separate connection for each (range of) service invocation priorities of a client component. The range can be defined using `low_range` and `high_range` option values of *BandedConnections*.

Thus, using the CQML modeling elements explained above, developers specify QoS configuration policies for their DRE systems. We now explain how these QoS policies specified by the developers are updated using our model transformation algorithm.

IV.2.2 Step II: QoS Policy Optimization Algorithm

Algorithm 3: Transformation Algorithm for Optimization of QoS Policy Configurations

Input: set of deployment plans SP, set of components SC, SCS, SCC, component c, c_p, deployment plan p, set of QoS policies SQ_1, SQ_2, qp_a, qp_b, set of collocation groups SCG, collocation group g

```
1  begin
2      foreach p ∈ SP do
3          SCG ← collocationGroups(p);
4          foreach g ∈ SCG do
5              c_p ← components(p);
6              SC ← SC + c_p | c_p ∈ cgComponents(g);
7              if c ∈ SC | c.priorityModel == SERVER_DECLARED then
8                  SCS ← SCS + c;
9              else if c ∈ SC | c.priorityModel == CLIENT_PROPAGATED then
10                 SCC ← SCC + c;
11             foreach c ∈ SCS do
12                 SQ_1 ← SQ_1 + c.QoSPolocy();
13                 minimize SQ_1
14                 subject to qp_a ⋈ qp_b | qp_a ≅ qp_b;
15             end
16             foreach c ∈ SCC do
17                 SQ_2 ← SQ_2 + c.QoSPolocy();
18                 minimize SQ_2
19                 subject to qp_a ⋈ qp_b | qp_a ≅ qp_b;
20             end
21         end
22         modifyDeploymentPlan(p, SQ_1, SQ_2);
23     end
24 end
```

Algorithm 3 shows our model transformation algorithm, which uses CQML as its source and target language, for optimizing QoS policies. The algorithm is executed for all the deployment plans specified for an application and the policy optimizations are applied for each such plan as shown in Line 2. In Line 6, all the components from a single collocation group are found.[3] Based on whether they have SERVER_DECLARED or

[3]Note that this is a host-based collocation group.

CLIENT_PROPAGATED priority model, they are grouped together in *SCS* and *SCC* as shown in Lines 8 and 10, respectively.

Finally, for each set of components above, the algorithm minimizes the number of QoS policies in Line 13 subject to the condition in Line 14: if QoS policies of two (sets of) components a and b each indicated in Algorithm by qp_a and qp_b, respectively, are similar (binary Boolean function $\cong$ finds whether the policies are similar), then they are combined (indicated by $\bowtie$) leading to reduction in the size of SQ_1. The Algorithm implements symmetric rules for CLIENT_PROPAGATED policy model. In Line 22 the results from applying all the above rules to the DRE system model are used to modify the current deployment plan, and the process is repeated for all the remaining plans of the DRE system.

At the end of step I, the developers created the application model that captured the initial QoS policies. The transformation Algorithm 3 is applied in step II to that DRE system model, which updates it and generates a modified QoS configuration policies using the rules described above.

IV.2.3 Resolving the Challenges in Optimizing QoS Configurations

Our automated, model transformation-based approach resolves the challenges we have discussed in Section IV.1 as follows: The inherent platform-specific complexities in optimizing DRE system QoS configurations are encapsulated in the transformation rules described in Section IV.2.2. The developers can thus focus on application business logic, and use our approach to optimize the QoS configuration. Further, the model transformation rules are reusable and can be applied repeatedly, during application development, and maintenance.

Next, we apply our approach to a representative case study and show how it improves the end-to-end invocation latency.

IV.3 Evaluating the generated QoS Configuration Optimizations

This section evaluates our approach to optimizing QoS configurations for component-based DRE systems. We describe our results in the context of a small but real DRE system use case. We show how the end-to-end latency results after applying our algorithm achieves considerable improvement over the existing state-of-the-art. Moreover, we also demonstrate how our results can be used by existing optimization frameworks like PAM [7].

IV.3.1 Representative Case Study

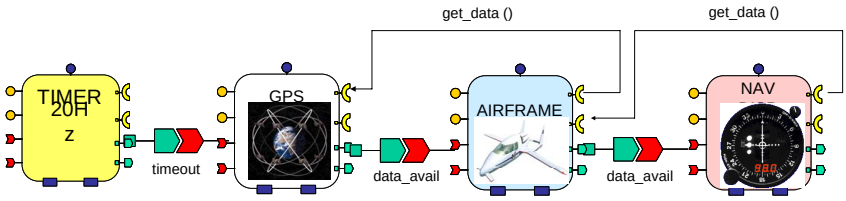


Figure IV.1: Basic Single Processor

The Basic Single Processor (BasicSP) is a scenario from the Boeing Bold Stroke [110] component avionics computing product line. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed using a QoS-enabled component middleware platform. The system is deployed using a single deployment plan on two physical nodes.

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario shown in Figure IV.1 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay*

component retrieves the data from the *Airframe* and updates its state and displays it to the pilot.

In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Timer* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space.

For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is LwCCM) ensure that these application-level QoS requirements are satisfied. For achieving the goal of reducing the latency between *Timer* and *NavDisplay* components, it is crucial to carefully analyze the QoS options chosen for each component in BasicSP, and exploit opportunities to either reuse or combine them such that this goal can be met.

IV.3.2 Experimental Setup & Empirical Results

We have used ISISLab (`www.dre.vanderbilt.edu/ISISlab`) for evaluating our approach in optimizing QoS configurations. Each of the physical nodes used in our experiments was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz network interface, and 40GB hard disks. Version 0.6 of our Real-time LwCCM middleware called CIAO was used running on Redhat Fedora Core release 4 with real-time preemption

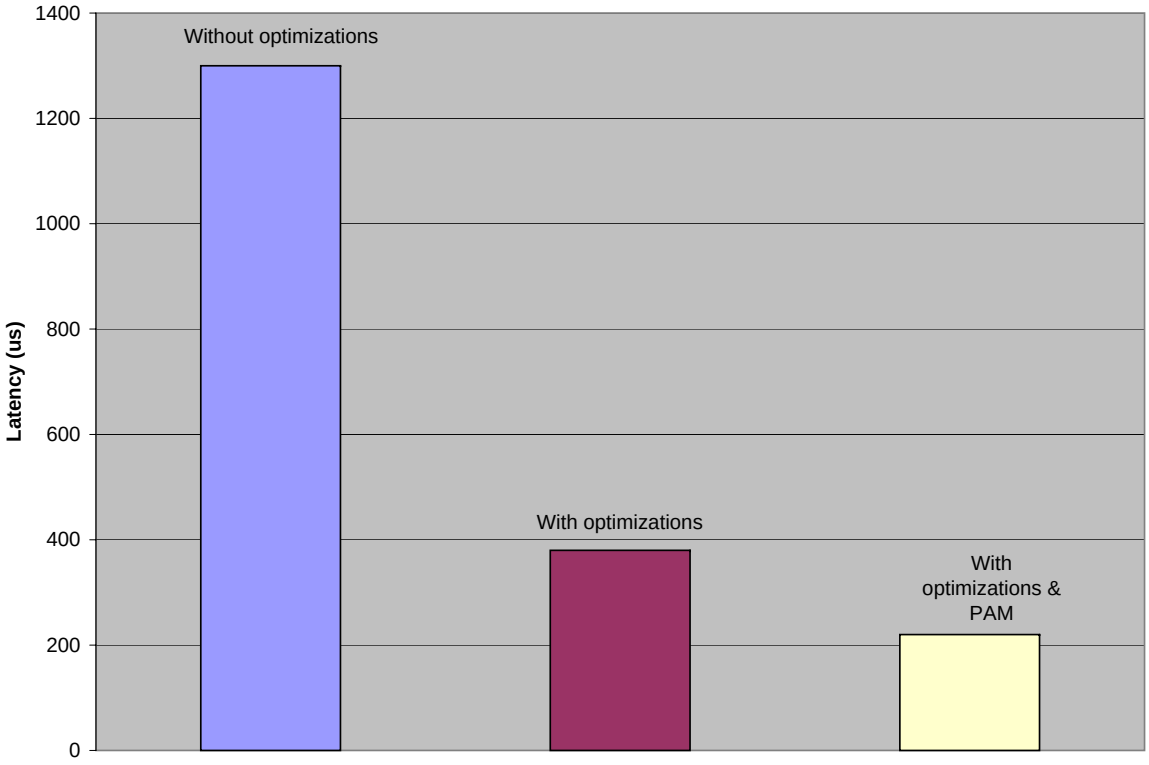


Figure IV.2: Average end-to-end Latency

patches. The processes that hosted BasicSP components were run in the POSIX scheduling class SCHED_FIFO, enabling first-in-first-out scheduling semantics based on the priority of the process.

As the first step we modeled the the BasicSP scenario and generated the standard QoS configuration for each of its components. We then applied the transformation algorithm 3 to our BasicSP application model that updated the application QoS policies. The BasicSP scenario was executed again with the updated QoS policies to get the results after our technique was applied.

Figures IV.2 and IV.3 show the results of applying our approach to BasicSP scenario explained in Section VI.1. The figure plots the average end-to-end latency and its standard deviation for the invocations from *Timer* to *NavDisplay* components in BasicSP with and without our approach. The results were obtained by repeating invocations for 100,000 iterations after 10,000 warmup iterations. As shown in Figure IV.2, the average latency

was improved by ∼70% when our technique was used for optimizing BasicSP QoS configurations. The standard deviation on the other hand, improved by ∼59% as plotted in Figure IV.3.

Without our approach, the initial BasicSP QoS configuration contained separate policies for each of its four components. Out of the four components, only the *Timer* component has SERVER_DECLARED priority model, while the rest of the components have CLIENT_PROPAGATED priority model. Thus, as indicated on Lines 13 and 14, when Algorithm 3 is applied to BasicSP, the QoS policy set is reduced to a size of two, one for each kind of priority model. This reduction in the size of the QoS policy set leads to the ∼70% improvement in end-to-end latency between *Timer* and *NavDisplay* components.

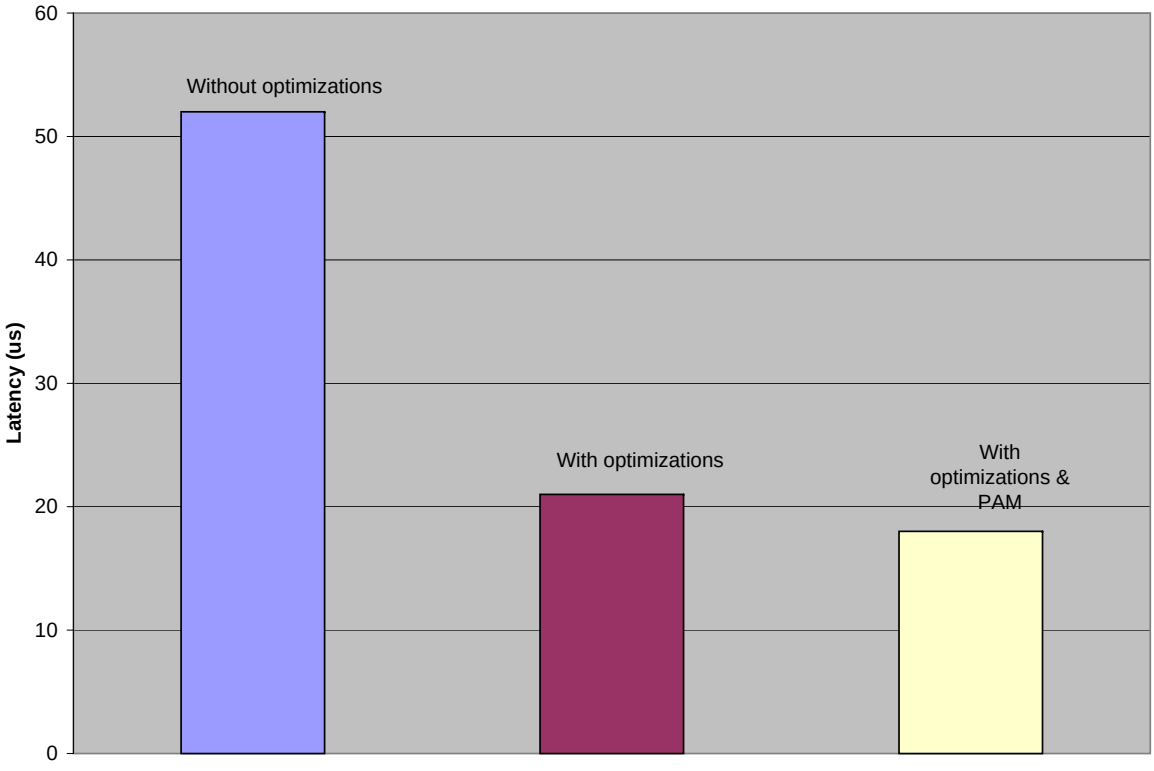


Figure IV.3: Standard Deviation in Latency

IV.3.3 Discussion

Our design-time approach described in Sections IV.2 and IV.3.2 relies on QoS configuration analyses in a platform-specific manner. We specifically showed how it has been realized in the context of a LwCCM middleware implementation. Naturally to extend it to other middleware platforms requires a careful study of the other platform's configuration space.

The results indicated an improvement of $\sim$70% in invocation latency between an execution path consisting of four components (the execution path here refers to the invocations from *Timer*, to *GPS*, to *AirFrame*, and finally to *NavDisplay* components in BasicSP). With large-scale, distributed applications comprising hundreds of components, we expect the improvements would be even higher. This is because, using our approach, the reduction in end-to-end latency is dependent upon how effectively the QoS policy sets SQ_1 and SQ_2 in Algorithm 3 are minimized. Large-scale DRE systems would have a number of QoS policies specified across their component assemblies, and in general, would be expected to have more opportunities to combine and reuse these policies leading to further latency improvements.

Existing deployment- and runtime techniques in standards-based middleware have focused on improving the performance of the infrastructure itself to optimize application-level QoS. For example, NetQoPE [6] deals with improving the deployment framework to improve network QoS provisioning by optimizing QoS policies. PAM [7] describes deployment-time techniques that allow for *fusing* of a set of components to reduce memory footprint and latency between service invocations.

One of the key differences between these techniques and our work is that we raise the level of optimizations to DRE system model-level. Thus, once the transformation algorithm has been applied, the modified DRE system model (*i.e.*, its QoS configuration specification) can be further used by the developers in verification. For example, these models can be used to reason about the correctness of various application properties using model-checking.

Our approach can be used in a complementary fashion with these techniques to further improve the application QoS. Since PAM is essentially a model-driven tool, the modified DRE system QoS configuration model resulting from applying our model transformation algorithm can directly be used to investigate fusion opportunities for the application. As shown in Figures IV.2 and IV.3, when applied in conjunction with PAM, our approach leads to a combined improvement of $\sim$83% in the end-to-end latency and $\sim$65% in the observed standard deviation in latency for BasicSP scenario.

CHAPTER V

MODEL TRANSFORMATION TEMPLATIZATION AND SPECIALIZATION

The industrial software development landscape has gradually transitioned from a strictly low-level, programmatic approach employing third-generation languages for various software development processes to a model-driven approach that relies heavily on the use of distinct application view models to better manage the system complexity. One of the primary requirements in model-driven software development is the support for (1) evolution of models along various application views, each corresponding to a different level of granularity, and (2) representation and translation of application into different models of computation (*e.g.*, finite state automaton, discrete event systems).

Model transformations are key to the success of model-based software development. They are used to define progressive refinements of application models from abstract, high-level views into low-level, detailed views that are used by the execution platform for different purposes, such as application configuration, deployment, and code synthesis. They are also used in transforming models to representations suitable for analysis tools that check various properties, such as correctness or deadlock free behavior. Additionally, they are also used in the transformation of application models to different models of computation used by analysis frameworks to facilitate examination of the application's functional and non-functional properties, such as ensuring property correctness using a model checking framework.

Model transformations have been applied in significantly diverse use cases as in (a) transforming XML documents from XSLT representation into XQuery representation [15], (b) middleware quality of service (QoS) configuration [56], which involves automatically mapping application-specific QoS requirements onto the correct QoS configuration options for the middleware platform, (c) transforming Simulink/Stateflow models into their hybrid

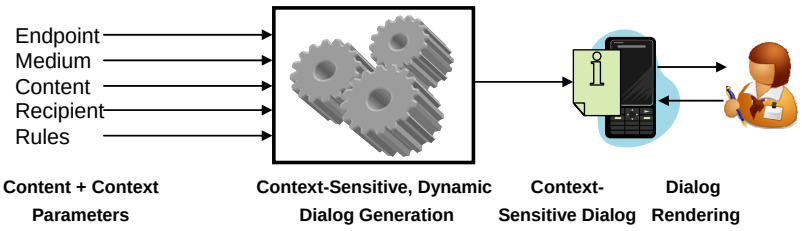


Figure V.1: Context-sensitive Communication Dialog Synthesis

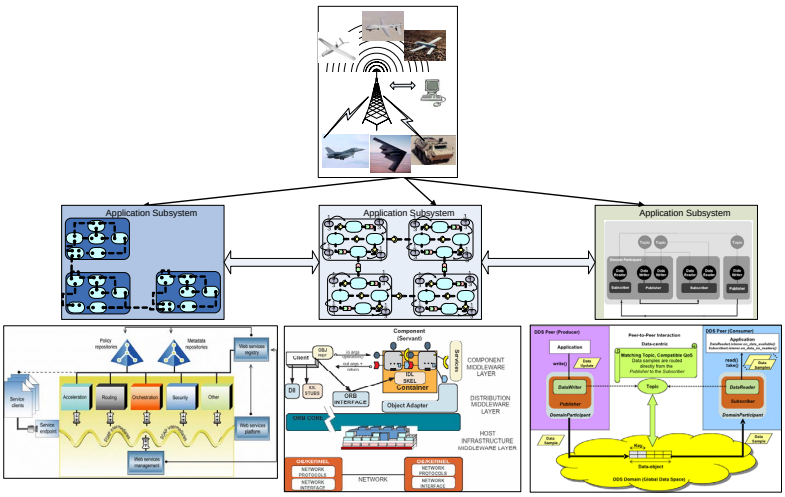


Figure V.2: Middleware QoS Configuration across a Heterogeneous Application

automata representation for formal verification [1], and (d) synthesizing dialogs for communication endpoints (*i.e.*, hardware devices/software applications for communications, such as cellphone, instant messenger (IM), pager) in enterprise workflows for rapid decision making [60].

Despite the diversity in the use cases, a noticeable trait in the model transformations for variants of individual use cases illustrates a significant commonality in the transformations. For example, as shown in Figure V.2, in the QoS configuration use case for a heterogeneous application, many generated middleware configurations are same across a class of applications that are related to each other due to similarities in their QoS requirements and the implementation platforms. The overall *mapping process* thus, moving from QoS requirements to QoS mechanisms, is essentially the same for all the implementation

platforms though the exact mechanisms and their values change. Similarly, in the dialog use case, despite differences in the communication endpoints, a number of dialog properties remain common. Thus, as shown in Figure V.1, the overall process of mapping communication dialog characteristics & content from the enterprise workflows essentially remains the same.

The existence of multiple variants within each use case illustrates a trait similar to product-line architectures (PLAs) [18]. PLAs are identified by the scope, commonality and variability (SCV) [22] engineering process, and rely on reusing services and artifacts for building systems rather than building them from scratch.

Our study of the advances in model transformations illustrates that despite the strong evidence of recurring patterns in the transformations, model transformation tools and techniques [32, 33, 53, 67] lack support for reusability, modularization and extensibility of the model transformation rules and algorithms. These shortcomings force the transformation developers to reinvent the transformation steps and the translation logic leading to significant code duplication in the transformations, and increased efforts in code maintenance and evolution activities.

Recent research efforts [30, 125] have demonstrated the use of model transformations to families of applications or product lines [18]. Yet, the following research questions remain unresolved: (a) How can the commonalities in the transformation process be factored out such that they can be reused by the entire application family? (b) How can the variabilities be decoupled from the model transformation rules while maximizing the flexibility of the transformation process? (c) How can the model transformation process for an application family be extended with new variants, however, with minimally invasive changes to the transformation rules? (d) How can all these capabilities be achieved with minimal to no changes in existing model transformation tools?

In this chapter we present *MTS (Model-transformation Templatization and Specialization)* to address these questions in the context of visual model transformation tools. MTS

provides transformation developers with a simple specification language to define variabilities in their application family such that the variabilities are factored out and are decoupled from the transformation rules. MTS provides a higher order transformation [1] [11] algorithm that automates the synthesis of a family-specific *variability metamodel*, which is used by transformation developers to capture the variability across the variants of an application family. Another higher order transformation algorithm defined in MTS generates the specialized instances of the application family variants. MTS requires minimal to no changes to the underlying model transformation engine.

Chapter Organization. The rest of the chapter is organized as follows: Section V.1 discusses two representative case studies; Section VI.3 describes the overall solution approach and demonstrates how it can be applied in practice; Section V.3 evaluates our approach.

V.1 Representative Motivational Case Studies

In this section we briefly discuss representative model transformation case studies taken from two diverse problem domains. Our goal is to highlight how two entirely different examples pose similar challenges for model transformation processes irrespective of whether the transformation is endogeneous or exogeneous [83]. To that end we illustrate the commonalities and variabilities in these diverse scenarios that their respective transformation processes must account for.

V.1.1 Communication Dialog Creation for an Insurance Enterprise

Our first case study deals with the creation of dialogs for a set of communication endpoints from *workflow* decision points in an insurance company. As part of their operation, modern enterprise workflows set up communications between decision makers (*i.e.*, employees) in an enterprise, and publish (collect) important information to (from) these decision makers. Since the employees in an enterprise may potentially be using several

[1] Since the transformation(s) themselves become the input and(or) output, we refer to the transformation process in MTS as higher order transformations.

endpoints (*i.e.*, devices), an important consideration in delivering information content from the workflows to the employees is the customization of communication dialogs for individual endpoints, which is accomplished using model transformations.

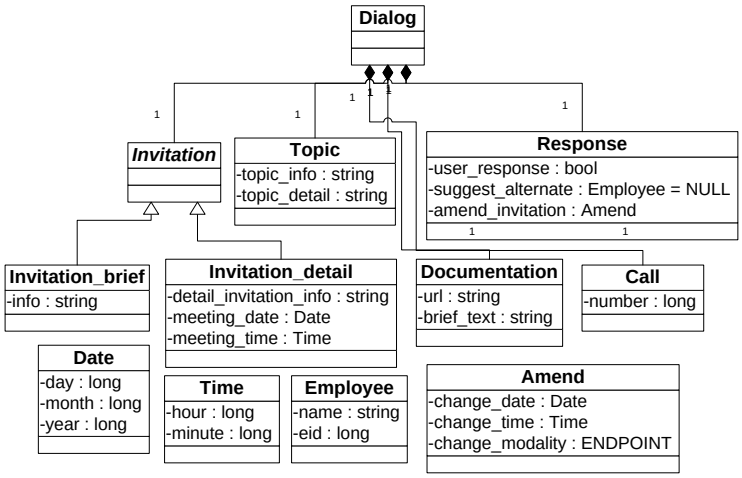


Figure V.3: A UML Representation of a Generic Communication Dialog.

Figure VI.2 shows the communication dialog metamodel used in our case study, which serves as both the source and target metamodel for the transformation process used to customize the dialog thereby making it an endogeneous transformation. The model transformation process is concerned with generating a dialog tailored to the properties of the communication end point. The transformation process must account for the following commonalities and variabilities: (1) the dialogs for all the endpoints contain, at the very least, the `topic_info`, and `info` (in `Invitation_brief` element) attributes; (2) the `Call`, `Documentation`, and `Response` model objects may be present in some but not all endpoints; and (3) the response to a communication dialog may be YES, NO, or may contain advanced options such as `suggest_alternate` or `amend_invitation`.

We will use this specific case study in this chapter to evaluate our MTS approach. Later, in Chapter VI we will discuss this case study in more details, show how we have applied our

MTS tool in the context of handling the variabilities is different communication endpoints in an enterprise.

V.1.2 Middleware QoS Configuration for Component-based Applications

Our second case study requires an exogenous transformation which translates component-based application QoS requirements into the underlying middleware platform-specific QoS configuration options. Figure V.4 shows the UML representation of both the source and the target metamodels used in the QoS configuration case study. As shown, the source metamodel contains the following Booleans for server components: (1) `fixed_priority_service_execution` that indicates whether the component changes the priority of client service invocations, and (2) `multi_service_levels` to indicate whether the component provides multiple service levels to its clients. The output metamodel models the real-time CORBA [95] middleware configurations.

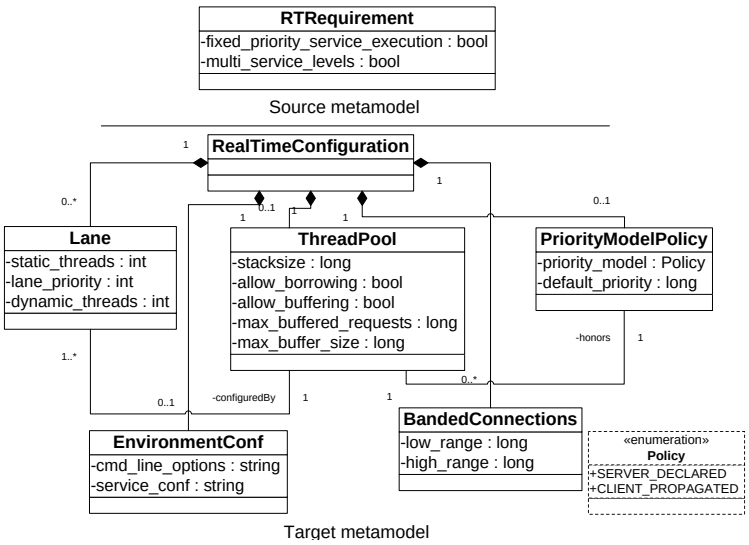


Figure V.4: A UML Representation of Middleware QoS Configuration Metamodels.

Transformations for middleware QoS configuration are applicable across a number of

application domains. The individual configurations generated using the model transformation should be easily customizable for slight variations in these domains. Thus, the case study has the following requirements for the generated middleware QoS configurations: (1) the `PriorityModelPolicy` object along with its attributes are transformed from `fixed_priority_service_execution` source attribute; (2) `Lane` object and its attributes are transformed from `multi_service_levels` source attribute. The `Lane` object cardinality and the exact values of its attributes, however, can change; and (3) the cardinality of `BandedConnections`, the values of all the attributes of `ThreadPool` except `stacksize` are assigned statically, however, they may vary for different application domains.

V.2 Templatized Model Transformations

We now present MTS (Model-transformation Templatization and Specialization). MTS uses the Generic Modeling Environment (GME) [2] as the modeling environment. GME provides a general-purpose editing engine and a separate model-view-controller GUI. GME is metaprogrammable in that the same environment used to define modeling languages is also used to build models, which are instances of the metamodels.

Transformation rules are defined using the Graph Rewriting And Transformation (GReAT) [53]

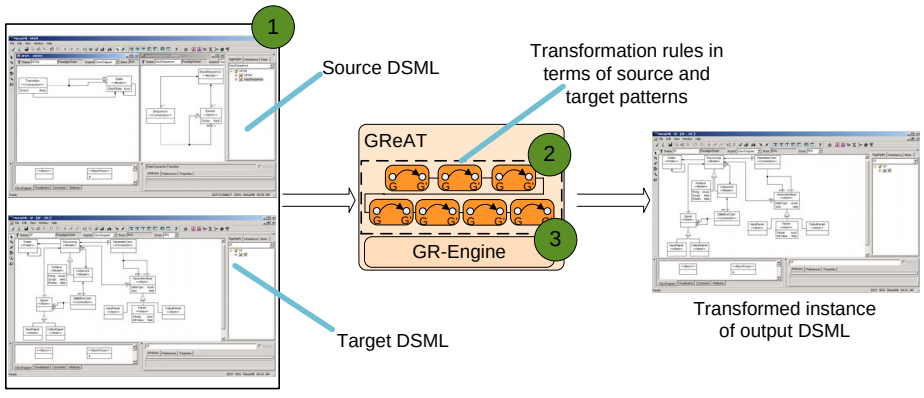


Figure V.5: Steps involved in developing model transformation using GReAT.

language. GReAT is developed using GME and can be used to define model transformation rules using its visual modeling language. It uses domain-specific modeling languages (DSMLs) as its source and target languages. Figure V.5 shows the various high-level steps involved in developing transformation algorithms using the GReAT tool chain. In Step 1, the source and target DSMLs for the transformation tool chain are defined. In Step 2, transformation developers use the GReAT transformation language to define various translation rules in terms of patterns[2] of source and target modeling objects. Finally, in Step 3, developers execute the GR-engine that translates the source model using rules specified in Step 2 into the target model.

The MTS approach shown in Figure V.6 leverages GReAT, however, without modifying it. The remainder of this section delves into the details of each step of MTS.

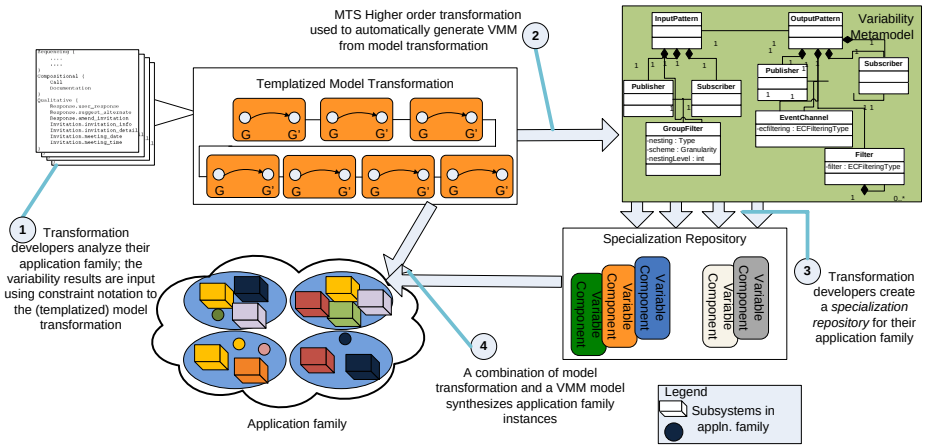


Figure V.6: MTS Approach to Reusable Model Transformations.

V.2.1 Step I: Defining the Templatized Transformation Rules

In programming languages like C++, templates are a mechanism to handle variability among concrete data types. We need a similar capability for model transformations, however, with minimum modifications to the model transformation tools. The basic idea

[2]Here, pattern refers to valid structural composition using model objects in source (target) DSML.

behind templatized transformations in MTS is that all the commonalities of an application family are transformed directly from the input models as family instance-independent transformation rules. The variabilities are dissociated from the transformation rules to allow independent evolution of the transformation and its variabilities.

Algorithm 4: Generating Family-specific VMM from Constraint Specifications.

```
   Input: source modeling language S, target modeling language T, templatized transformation (set of its rules) R
   Output: variability metamodel V
 1 begin
 2     transformation rule r; constraint notation block cnb; set of constraint notation blocks CNB; structural variability cm;set of
       structural variabilities CM; quantitative variability qm;set of quantitative variabilities QM; pattern p; modeling object ob;
       attribute at; modeling object type type;attribute type atttype; integer c;
 3     initializeVMM(V);
 4     foreach r ∈ R do
 5         if r.cnb() ≠ ∅ then
 6             CNB ← r.cnb(); // populate all constraints specifications for that rule
 7         foreach cnb ∈ CNB do
 8             if cnb.structuralVariabilities() ≠ ∅ then
 9                 CM ← cnb.structuralVariabilities();
10                 foreach cm ∈ CM do
11                     p ← cm.SRC();
12                     foreach ob ∈ p do
13                         parseLanguage(S,ob,type); createSRCObject(V,ob,type);
14                     end
15                     p ← cm.TGT();
16                     foreach ob ∈ p do
17                         parseLanguage(T,ob,type); createTGTObject(V,ob,type);
18                     end
                       /* Do similar steps for patterns in target                                                    */
19                     composeVariabilityAssociation(V); /* creates a connection between source and target objects
                         created earlier                                                                          */
20                 end
21             if cnb.quantitativeVariabilities() ≠ ∅ then
22                 QM ← cnb.quantitativeVariabilities();
23                 foreach qm ∈ QM do
24                     p ← qm.SRC();
25                     foreach ob ∈ p do
26                         parseLanguage(S,ob,type); createSRCObject(V,ob,type);
27                          foreach at ∈ ob do
28                             parseObject(ob,at,atttype); createSRCAttribute(V,ob,at,atttype);
29                         end
30                     end
31                     p ← qm.TGT();
32                     foreach ob ∈ p do
33                         parseLanguage(T,ob,type); createTGTObject(V,ob,type);
34                         foreach at ∈ ob do
35                             parseObject(ob,at,atttype); createTGTAttribute(V,ob,at,atttype);
36                         end
37                     end
38                     composeVariabilityAssociation(V); /*creates a connection between source and target objects created
                       earlier*/
39                 end
40             createContainingObject(V); /* name of the containing object is a combination of rule name, and constraint
                 block name, each of which must be unique                                                         */
41         end
42         CNB ← ∅; /* constraint blocks from previous loop are deleted, s.t. those from the next rule can be read  */
43     end
44 end
```

Application families are defined by the commonalities, which constitute the *invariant* characteristics of the family, and variabilities, which constitute family member idiosynchrasies or dissimilarities among the members. MTS enables all the common features of an application family to be transformed directly from the input specification as family instance-independent transformation rules. MTS however decouples the variabilities from the transformation rules to allow independent evolution of the transformation and its variabilities.

To realize this, transformation developers must first carry out scope, commonality and variability (SCV) [22] analysis of their application family. The developers then express the results of SCV analysis as constraint specifications, which is a capability in MTS we discuss below. The following two categories of variabilities can be specified using our constraint notation specification:

(a) Structural variabilities, where the basic building blocks *i.e.*, model elements, or their cardinalities in every family member model are different. Thus, the variation in family member models emanates from dissimilarities in their structural composition.

(b) Quantitative variabilities, where the family member models may share model elements, but the data values of their attributes are different.

Table V.1 shows the results of the SCV analysis for the QoS configuration case study presented in Section V.1.[3] For example, in the QoS configuration case study, we show three configuration variants with their commonalities and variabilities. The variabilities are further classified as structural and quantitative. For example, `Config_1` includes only the `Lane` attribute in its structure while the others include even the `Banded_Connections`. Although the attributes in the quantitative variability dimension are same across the three configurations, they vary in the values of these attributes.

Both these variabilities in Items (a) and (b) above can be denoted as simple implication

[3]Due to space constraints the MTS steps are shown only for the QoS configuration case study.

Table V.1: SCV Analysis Results for QoS Configuration Case Study.

Commonalities	Family Variant	Variabilities in Family Variant	
		Structural	Quantitative
PriorityModelPolicy, EnvironmentConf, stacksize	Config_1	Lane	allow_borrowing, allow_buffering, max_buffd_reqs., max_buff_size,
	Config_2	Lane, Banded-Connections	allow_borrowing, allow_buffering, max_buffd_reqs., max_buff_size
	Config_3	Lane, Banded-Connections	allow_borrowing, allow_buffering, max_buffd_reqs., max_buff_size (Note: values differ from that of Config_1 and Config_2)

relations and are characterized by one of the following types of associations between source ($s \in S$) and target ($t \in T$) objects:

- A one-to-one association between a pair (s,t) of source and target objects respectively, defined as an injective function: $s \in S, t \in T \; \exists \; f(s) \overset{\alpha}{\longmapsto} f(t)$ such that $if \; f(s) = f(t) \; then \; s = t$.

- A one-to-many association between pairs (s,t) of source and target objects respectively, defined as: $s \overset{\phi}{\longmapsto} t$, where $s \in S$, and $t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$.

- A many-to-one association between pairs (s,t) of source and target objects respectively, defined as: $s \overset{\varphi}{\longmapsto} t$, where $t \in T$, and $s = \{P1(s_1, s_2, .., s_m) \mid s_{j=1..m} \in S\}$.

- A many-to-many association between pairs (s,t) of source and target objects respectively, defined as: $s \overset{\varphi}{\longmapsto} t$, where $s = \{P1(s_1, s_2, .., s_m) \mid s_{j=1..m} \in S\}$, and $t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$.

Note that, $P1$ and $P2$ denote patterns of source and target objects.

MTS defines a constraint specification notation whose form is shown in Figure V.7. This notation is used by developers after the SCV analysis to capture the variabilities in their transformations. The `Quantitative` block captures all the attributes while the `Structural` block captures all the model elements that vary between family members. In the `Structural` block shown, there is an association defined between source model object `I1`, and target model objects `O1` and `O2` which implies that the composition of `O1` and `O2` is dependent on `I1`.

```
1   Structural {          Quantitative {
2     I1::O1;O2             I2:A1::O3:A1,O3:A2,O3:A3
3     O7                    I3:A3::O5:A6
4   ...}                    O6:A7
5                         ...}
```

Figure V.7: Syntax of Constraint Specification Notation.

In Figure V.7, the `Quantitative` block captures many-to-many and one-to-one relations between source and target elements. For example, the values of `A1`, `A2`, and `A3` of model object `O3` are dependent on that of the `A1` attribute of object `I2`, as shown in in Line 2 of `Quantitative` block. The specification `O6:A7` states that the attribute `A7` is directly mapped, *i.e.*, it is hard-coded and is assigned statically in the model transformation. This is also true for `O7` in `Structural` block which is created in target model irrespective of presence of any specific source model object(s).

To realize this capability without requiring modifications to the underlying transformation tool like GReAT, MTS requires developers to insert the constraint specifications as special comments inside the `AttributeMapping` model element in GReAT. This element allows assignment of the attributes of matched objects in a transformation rule using C++ code. Note that the constraint specification is opaque to the GReAT interpretation logic (for reading source and target metamodels of the transformation) and hence does not affect its GR-engine execution. Our approach can easily be extended to other model transformation tools as long as they provide the means to develop higher order transformations necessary for implementing MTS.

Figure V.8 shows an excerpt of the templatized transformation rules for the QoS configuration case study. Notice how the structural variability from Table V.1 is captured as a transformation rule from the source element, *i.e.*, `RTRequirement`, to the target element, *i.e.*, `Lane`. The excerpt of the constraint specification block shown captures the Quantitative variability for the attributes of these source and target elements.

As shown, there is an association between `multi_service_levels` attribute of

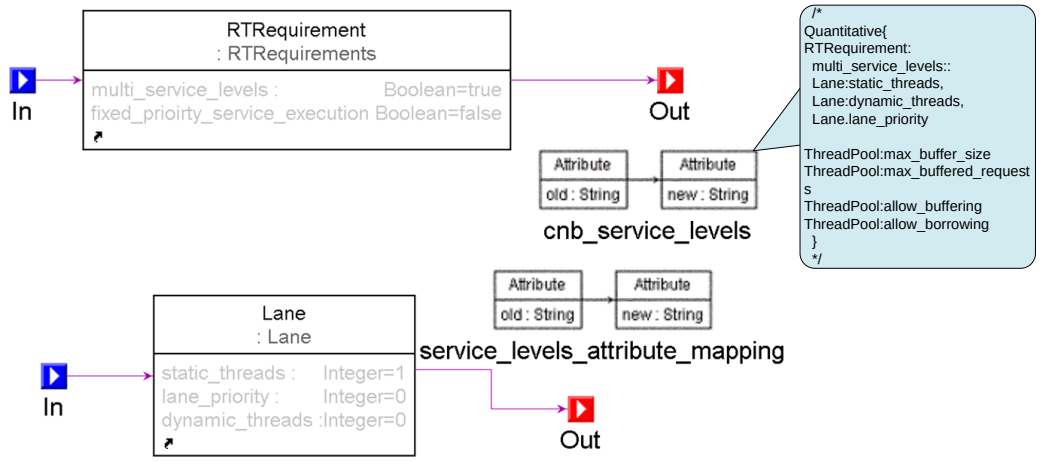


Figure V.8: Templatized Transformation Rule in QoS Configuration Case Study.

`RTRequirement` source object and attributes of `Lane` target object. This indicates that the values of the latter are dependent on that of the former, and that the values themselves can vary for different configurations. The example specification implies the following: for `Lane` attribute set {`static_threads`, `dynamic_threads`, `lane_priority`}, `config_1` in Table V.1 can have data values {10, 20, 50}, while `config_2` can have data values {5, 2, 15} to realize the requirement that the server component should support multiple levels for service invocations of its clients.

V.2.2 Step II: Generating Variability Metamodels from Constraint Specifications

Although the constraint specifications discussed in Step I capture the variability in the transformations, the notation used is not recognized by the model transformation tool. The next step in MTS therefore converts the constraint specifications into a *Variability Meta Model* (VMM) using a higher order transformation. It is used to create source and target model objects in the VMM that correspond to the variabilities in these specifications. A VMM modularizes the variabilities and decouples them from the model transformation rules to promote independent evolution.

Algorithm 4 depicts the higher order transformation for generating the VMM and the

Algorithm 5: Specializing the Model Transformation from a VMM model.

Input: variability metamodel V, templatized transformation R
Output: specialized instance of input templatized transformation R'

```
1  begin
2      transformation rule r; set of model objects OB,IO; pattern p; modeling object ob,io,tmp; attribute at; modeling object type
       type;attribute type atttype;
3      R' ← R; OB ← containingModelObject(V);
4      foreach ob ∈ OB do
5          r ← searchRule(R',objName(ob)); createTempObject(tmp,r); deleteCNB(r);
6          IO ← parseSRCPattern(ob);
7          foreach io ∈ IO do
8              createObjectRefs(io,tmp); assignCardinalities(io,tmp);
9              createAttribs(io,tmp); assignValues(io,tmp);
10         end
           /* do similar steps for target pattern                                                                    */
11         IO ← parseTGTPattern(ob);
12         foreach io ∈ IO do
13             createObjectRefs(io,tmp); assignCardinalities(io,tmp);
14             createAttribs(io,tmp); assignValues(io,tmp);
15         end
16     end
17 end
```

process itself is captured in Figure V.9. The basic idea behind the algorithm is as follows: Recall from Section V.2.1 that the structural variability is concerned only with capturing the (source and target) model objects (or their cardinalities) used in composition of family variants. For every structural variability block, the algorithm creates the corresponding model objects in VMM. The quantitative variability, on the other hand, captures the dissimilarities in values of model object attributes. Therefore, for these variabilities the algorithm creates model objects and their attributes as well.

The function $initializeVMM(V)$ on Line 3 creates a new VMM, V, and initializes its internal variables. This is necessary so that in the following rules the syntax and semantics of V can be defined in GME. Line 11 reads the source patterns that correspond to every structural variability in the templatized transformation R. Next, the types of each modeling object for the pattern read in the previous rule are deduced by parsing the modeling language as shown in Line 13. This type information is used to create appropriate modeling objects corresponding to the specified source patterns. Similar logic is carried out for patterns in the target language.

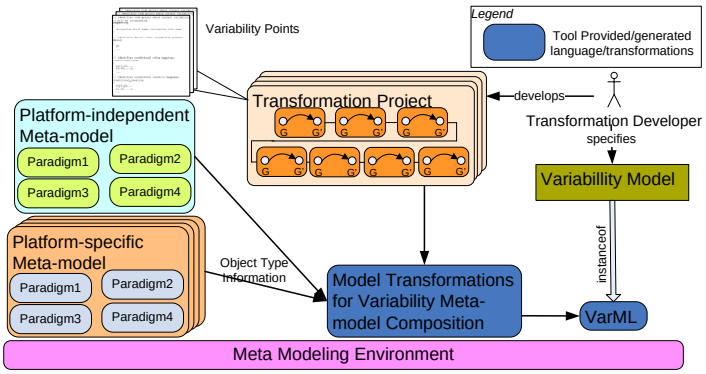


Figure V.9: The Generation of VMM using MTS Higher-order Transformation.

Once the source and target objects are created in the VMM, the function `compose-VariabilityAssociation (V)` creates a simple connection between these objects to denote their association. In a similar fashion, VMM modeling objects are generated for quantitative variabilities in R. Additionally, for quantitative variabilities, attributes of the corresponding modeling objects are also created. The final rule creates a new model object, that contains each of these source and target objects created in earlier rules, as shown on Line 40.

We applied Algorithm 4 to the templatized model transformation of our QoS configuration case study to automatically generate a VMM. Figure V.10 shows a screenshot of the generated VMM in GME. The variabilities are modeled as pairs of `SourcePattern` and `TargetPattern`, and annotated by whether they are `Structural` or `Quantitative` using Boolean attributes. The figure corresponds to the `Quantitative` variability rule of Figure V.8 in that the attributes of a `Lane` are dependent on the `multi_service_levels` attribute of `RTRequirement`. The `ThreadPool` attribute values, on the other hand, can vary among each configuration (*e.g.*, `Config_1`, `Config_2` etc. in Table V.1), and are generated in the VMM.

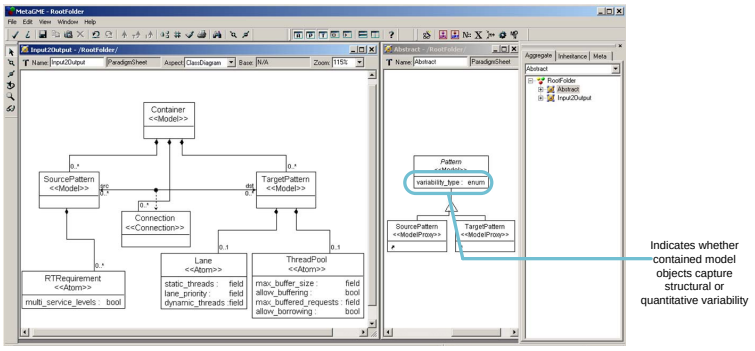


Figure V.10: Generated VMM for the Representative Case Study.

V.2.3 Step III: Synthesizing a Specialization Repository

In the next step transformation developers use the generated VMM to create VMM model instances, where each VMM model corresponds to a family member (or more appropriately, variabilities of a family member). Figure V.12 shows how individual mappings in the initial constraint specification are translated into model objects in VMM. A collection of such VMM models for a family is called a specialization repository. This step is similar to partial specialization in programming languages for different data types.

Figure V.11 shows a sample VMM model that instantiates

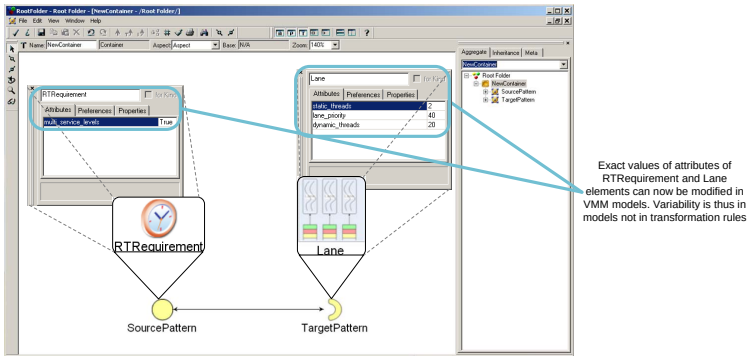


Figure V.11: A Sample VMM model for a Variant of QoS Configuration Case Study.

the quantitative variability in terms of exact values of the `RTRequirement` and `Lane` attributes. Note that since the exact values are now specified as models rather than being

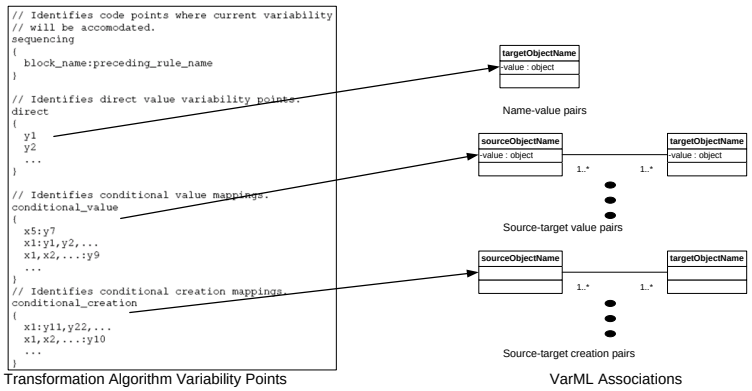


Figure V.12: Translations of Variabilities into VMM Model Objects.

encoded in terms of transformation rules, it is considerably easier to modify these values. Notice that none of the rules are modified to change an existing mapping, and hence the transformation logic need not be re-compiled and linked.

V.2.4 Step IV: Specializing the Application Instances

To realize a complete transformation for a family variant, transformation tools like GReAT must combine the VMM models generated in Step III, which are partial specializations, along with the (original) templatized model transformation rules. To address this need, MTS provides a second higher order transformation that (1) reads the input VMM model for a family member, and (2) adds temporary objects at appropriate points in the templatized transformation rules, which serve as placeholders to insert the instantiated variability of a family member (corresponding to the current VMM model). Figure V.13 shows the overall approach.

Algorithm 5 shows the translation rules in this transformation. Lines 3–5 create a new model transformation instance R' from the input templatized transformation R, read the

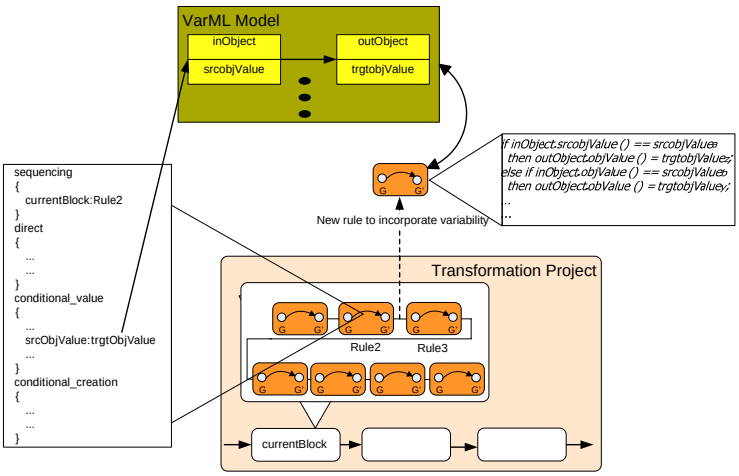


Figure V.13: Specializing the Application Instances using MTS Higher-order Transformation.

containing model objects in VMM V, and for every model object ob search the corresponding rule in the transformation R'.[4] This rule denotes the location where the variabilities contained in ob were specified in Section V.2.1.

Once rule r is known, the constraint block is deleted from this rule in function $deleteCNB(r)$. The function $createTempObject(tmp, r)$ creates a temporary object tmp inside this rule. For every `Structural` variability in the source pattern in ob, object references are read from V, and created in tmp and in addition, their cardinalities are assigned as shown in Line 8. Similarly, attributes in VMM that capture `Quantitative` variabilities are read from V, and created and assigned values in tmp in Line 9. The same rule is also repeated for all target patterns in ob.

We applied Algorithm 5 to our QoS configuration case study. One of the rules in this case study assigns specific data values to the attributes of `Lane` (target) element depending on whether or not the `multi_service_levels` (source) element value is set to `TRUE`. Further, as identified earlier in Section V.2.1, there is a quantitative variability involving

[4]For creating application family instances, it is not necessary to create a new instance R', but is done only for Algorithm 5 to avoid modification of the original templatized transformation R.

these two elements. The same variability is also given in Figure V.14 for reference. The attributes in `tempObject` are assigned values from the values of the corresponding attribute in the VMM model. Similarly, for the structural variability, the model object references are also created by parsing and reading the VMM model.

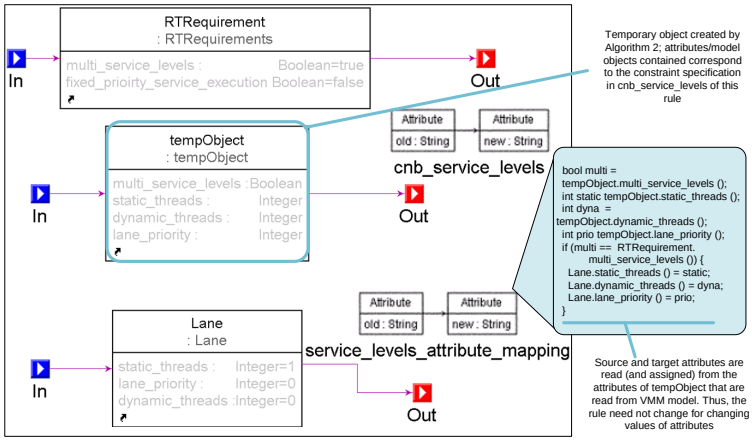


Figure V.14: Specialization of a QoS configuration rule using MTS.

Thus, the rule `service_levels_attribute_mapping` (and in effect, the model transformation itself) need not change, when some of these data values/model object cardinalities have to be altered. This is because the modifications can now be done simply by modifying the appropriate VMM model.

V.3 Evaluating the Merits of MTS

Since MTS is developed to enhance reusability, we first evaluate its merits in terms of the reduction in effort to write the transformation rules. Second, since MTS provides higher order transformations, we also discuss the overhead incurred by the higher order transformations. Our prototype implementation of MTS is part of the CoSMIC[5] tool suite. For all of our experiments below, we used CoSMIC version 0.5.7. We used GME version 6.11.9 and GReAT version 1.6.0 software packages which are necessary for using MTS.

[5]http://www.dre.vanderbilt.edu/cosmic/

All the overhead measurement experiments were run on a Windows XP SP2 workstation with 2.66 GHz Intel Xeon dual processor and 2 GB physical memory.

V.3.1 Reduction in Development Effort using MTS

Recall from Section VI.3 that to create a target model from source model using GReAT, developers need to execute the GR-engine that executes all the translation rules of that model transformation. More specifically, developers must first specify all the rules that transform the elements of the source model to the target model. Thereafter, the GR-engine execution involves the following steps: (1) executing the *master interpreter* that generates the necessary intermediate files containing all the rules in the current transformation, (2) compile these intermediate files, if not done already, and (3) run the generated executable. Steps 1 and 2 must be executed each time the model transformation rules are modified – which is the case with variants of a family.

Table V.2: Details of the representative case studies.

(a) The size of the metamodels.

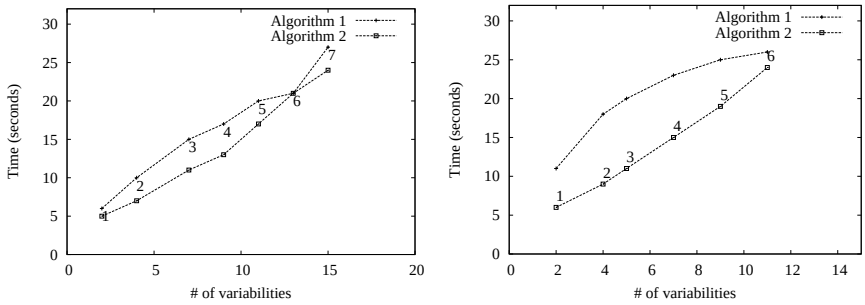
Metamodel	# of modeling elmts.	# of attribs.	# of conns.
Insurance Enterprise			
SRC/TRGT	8	14	0
QoS Configuration			
SRC	3	2	2
TRGT	8	14	4

(b) Distribution of variabilities.

Data Point	Insurance Enterprise		QoS Configuration	
	Quantitative	Structural	Quantitative	Structural
1	2	0	2	0
2	2	2	4	0
3	3	4	5	0
4	3	6	5	2
5	5	6	6	3
6	6	7	8	3
7	6	9	n.a.	n.a.

Without MTS, model transformations for each variant of an application family must expend effort in all of the above steps. Our goal is to evaluate MTS in terms of effort saved. We focus on two specific cases discussed below.

Case 1: Newly added variant is subsumed by existing constraint specifications: The existing constraint specification for the application family may be sufficient to capture all the variabilities of a new family variant. Thus, the developers can create a new variant



(a) Insurance enterprise case study (b) QoS configuration case study

Figure V.15: Overhead in using MTS for the development of templatized transformations. The Y axis denotes the time taken by Algorithms 4 and 5.

simply by re-executing the same model transformation with VMM model of the variant as one of the inputs to the transformation. Note that the first two steps have to be performed only once when the model transformation is being executed for the first time. Since all the instance-specific customizations/changes are done in the corresponding VMM model, developers only need to execute Step 3 after each change to produce output of the transformation (*i.e.*, a new family instance).

In contrast, the traditional approach of one model transformation per single (subset of) family instance(s) will require maintenance of $I * R_n$ rules, where I is the number of family instances, and R_n is the average number of rules per instance. With MTS, assuming that the average number of rules do not change, the total number of rules to be maintained reduces by a fraction of $\frac{I-1}{I}$.

Case 2: New variant requiring additional constraint specifications: If the variabilities of a new family variant are not completely captured using existing constraint specification for the application family, MTS requires enhancements to the constraint specification itself. Such a change necessitates executing the first two steps above once to produce a new VMM which can be used to model variabilities in the new variant. Note that despite this change, the VMM models corresponding to the existing variants will still be valid provided the

changes in constraint specification (on account of a new family variant) are orthogonal to the existing variabilities.

V.3.2 Performance Overhead of using MTS

The rationale behind these experiments is to quantify the overhead placed by the higher order transformations in Algorithms 4 and 5 when the number of structural and quantitative variabilities are increased. The performance overhead was calculated in terms of the time taken by each of these algorithms when used in the context of each of the two case studies. In all we identified (a maximum of) fifteen variabilities for insurance enterprise case study, and eleven variabilities for QoS configuration case study. The performance overhead was measured by increasing the variabilities in each case study from a minimum value of two to the maximum values above. The size of both the metamodels is given in Table V.2a. Table V.2b shows the distribution of variabilities across the quantitative and structural dimensions for these cases.

Figure V.15 shows the overhead involved in using MTS to generate VMM (Step 1), and specialize the transformation (Step 4). In general, the algorithms take slightly more time for QoS configuration than the insurance enterprise, for the same number of variabilities, which is attributed to the larger size of the combined size of the source and target metamodels of the former.

For a variation of {Q=4, S=9} in insurance enterprise case study where Q and S denote the total variation in quantitative and structural variabilities, respectively, the time complexity of Algorithm 4 increased by 350% from an initial value of 6 seconds, while that of Algorithm 5 increased by 380% from an initial value of 5 seconds. For QoS configuration case study, with a total variation of {Q=6, S=3}, the increase was $\sim$136% and $\sim$300%, for Algorithms 4 and 5, respectively.

Thus, if the new family variant is already subsumed by the notation as discussed in case 1 in Section V.3.1, the developers incur an additional overhead in using MTS only for

the first time when each of these algorithms have to be applied (*i.e.*, once for generating VMM, and once for creating temporary objects in the model transformation). Thus, the cost of using MTS is amortized over the total number of transformation runs, during the development cycle of that application family. For the remaining cases, if the variabilities of the new variant are not captured by the existing specification, the two steps listed in Section V.3.1 have to be executed once after modifications in the specification have been made according to the variabilities of the new variant.

CHAPTER VI

APPLYING MTS TO CONTEXT-SENSITIVE ENTERPRISE COMMUNICATION DIALOG SYNTHESIS

As part of their normal or exceptional operation, modern enterprise workflows not only set up communications (calls, conferences, chats, etc.) between decision makers in the enterprise but also need to deliver information to users and, in return, collect important input from users in a timely fashion. Such input is typically based on the information that a workflow delivered to a user and serves as the basis for further decision-making in the workflow. As enterprises strive to increase productivity and efficiency by automating their business processes through workflows, there is a growing need to accelerate this type of interaction between workflows and enterprise users. In this context, we call a mechanism to present information to a user and collect subsequent feedback from the user a *dialog* between workflow and user.

Increasingly, context-aware communications middleware is used to provide communication support to workflows including the synthesis, delivery, and rendering of dialogs. The need for accelerating the interaction between workflows and users results in a requirement to embed sophisticated context-sensitive dialog synthesis, delivery, and rendering mechanisms in the middleware to reach enterprise users in a ubiquitous fashion. Due to ever-increasing user mobility and progress in communications technology, enterprise user communication environments have changed from a limited set of fixed, largely stationary devices and clients to a wide array of personal and shared, stationary and mobile communications endpoints of differing capabilities and supporting different kinds of media. The panoply of endpoints in use in modern enterprises poses a set of complex challenges to the context-sensitive support for dialogs in communications middleware. With a potentially large volume of dialogs between workflows and certain users, the receipt, perusal of, and

response to dialogs has to be as convenient and efficient for the user so as to maintain a high level of user productivity.

The true difficulty with our goal of customizing dialogs for a multitude of endpoints and based on user context, in the interest of accelerating workflow/user interactions and maintaining user productivity, is that most dialogs are created at workflow runtime. Thus, dialog customization has to be done dynamically as well and suggests the development of a large number of customization software modules (akin to device drivers). These modules would need to be constantly adapted to the ever-changing landscape of endpoints.

Despite the need for such a customization, the set of dialogs tend to share strong commonalities with each other and, depending on the endpoint on which they have to be rendered, have certain distinct characteristics. Thus, there is a significant opportunity to synthesize *families of dialogs* by employing customizable and reusable software patterns and artifacts, as opposed to building them from scratch. Product-line architectures (PLAs) and its characteristic scope, commonality, and variability (SCV) [22] engineering process present a promising approach to development of families of dialogs.

This chapter first describes how we have conducted SCV analysis for a family of dialogs. We then show how templatized model transformations can be used to synthesize customized dialogs. In our approach, the commonalities among the dialog variants are captured as a common set of transformation rules. Higher order parametrized rules capture the variability lending themselves well to the notion of templatized transformations.

Chapter Organization. The remainder of the chapter is organized as follows: Section VI.1 discusses an enterprise case study that motivated our work on dialogs; Section VI.2 presents the challenges in context-sensitive dialog synthesis in detail; Section VI.3 discusses design details of our solution and lists various steps involved in the development of reusable model transformations, and how we have applied it to our case study.

VI.1 A Case Study Motivating Context-Sensitive Dialogs

An example of context-aware communications middleware that supports enterprise workflows and ubiquitous, automated dialogs between workflows and users is *Hermes* [51], developed at Avaya Labs Research. An illustrating use case scenario for Hermes, drawn from an extensive case study with several insurance companies, is a business process workflow that deals with claims in a car insurance company. The workflow gets triggered when a policy holder calls in an insurance claim for damages to his/her car. Suppose this claim raises a difficult question and it is unclear how to apply the insurance company's rules to this claim. In such a case, the workflow attempts to set up a conference call between various employees of the insurance company, including a legal expert, an appraiser, and the appraiser's supervisor, to resolve the question.

As part of the process of setting up the conference call, the workflow has to first reach out to potential participants and present (1) a conference call topic (open claim), (2) documentation or a link to documentation pertaining to the case, possibly containing audio, video, and image elements in addition to text, (3) an invitation to a conference call, and (4) a range of user response options to establish the user's ability, availability, and willingness to participate in the conference call.

These four items constitute a simple dialog in Hermes. For example, the dialog may first provide the information *"There is an open claim from policy holder 243779 that cannot proceed due to a mismatch between the appraised damage and a corporate limit on lifetime coverage for a vehicle. For more information, please consult case number 243779-041"*. Next, the dialog may pose the question *"Can you be available to participate in a voice conference about this claim at 2 PM EDT today?"* Eventually, the dialog gives the user a range of response options including "Yes", "1 hour earlier", "1 hour later", "Only if you cannot find somebody else", and "No".

A communications-enabled workflow platform like Hermes could, of course, simply send a notification of a pending dialog to the recipient via an email that contains a link

to an enterprise portal with the actual dialog. The dialog could then be an HTML form or similar. However, this procedure may lead to many scenarios where the recipient may not receive or respond to the dialog in a timely fashion, thus violating our stated goal of accelerating the interaction between workflows and users. The following are some of these scenarios:

1. The recipient is in a location such as a car, an off-site meeting, or a conference room with sporadic, limited, or no email access.
2. Due to a focus on other activities, the recipient is not checking incoming email frequently enough.
3. The recipient can receive email on a mobile device which does not have access to the enterprise portal.
4. The mobile device of the recipient cannot render Web pages or makes it very inconvenient to navigate the enterprise portal and dialogs.
5. The recipient is driving or for other reasons is not in a position to read a dialog or use a manual input device (mouse, keyboard, keypad) to respond to it.

To remedy *some* of the problems that the above approach to conveying dialogs to a user incurs, the Hermes middleware attempts to send (1) a dialog topic (Item 1 above), (2) a URL for the actual dialog in a Web-based portal (i.e., a link to Items 2-4 above) to an endpoint that the user is likely to be present on at this time. Hermes makes a determination of the target endpoint based on the user's context information which includes information about the user's presence and activities on various monitored endpoints such as telephones, instant messaging (IM) and email clients, Web browsers, etc. However, the user must access the URL via a Web browser to find the dialog in question. As with the email approach described in the previous paragraph, forcing the user to log into the Web portal, finding the dialog there, reading it, and responding to it via a mouse or keyboard may not be possible in a timely fashion or, at the least, may negatively impact user convenience and productivity.

Our goal for Hermes is therefore to send the entire dialog to a specific endpoint and

to customize its rendering to this endpoint and a given communication modality (voice, IM, email, Web, SMS, etc.). For example, assuming that the user is present and active on a Web browser on his/her office computer, a dialog may be presented as an HTML popup in that browser [58]. This endpoint is also suitable for presenting the supporting documentation (Item 2 above) about the insurance claim. The response options in the dialog can be rendered as HTML buttons. In addition, a more sophisticated response option may be included, such as a text box that allows a freeform specification of a different time, day, and modality by the user.

On the other hand, suppose the employee is known to be present on his/her mobile phone that has no (known) data connectivity and only a standard phone numeric keypad. Due to its limited hardware capabilities, the content of the dialog to be sent to the mobile phone ought to be significantly different from the HTML popup described earlier. Moreover, the mobile phone user may not be in a position to read the dialog on the mobile device or respond via the keypad because he/she may be driving a car at this point in time. Thus, the dialog may best be rendered as a *call* to the mobile phone with a VoiceXML (VXML) script that first renders Item 1 as voice, skips Item 2 except for a brief summary of the documentation, and renders Item 3. Finally, for collecting the user's response, the script reads the available response choices and asks for either a voice response (*"Yes"* etc.) and/or a key input (*"Press 1 for Yes"* etc.).

Clearly, the set of customized dialogs derived from a defined sequence of Items 1-4 share strong commonalities with each other and, depending on the endpoint on which they have to be rendered, have specific distinct characteristics.

Thus, there is a significant opportunity to synthesize *families of dialogs* by employing customizable and reusable software patterns and artifacts, as opposed to building them from scratch. Scope, commonality, and variability (SCV) [22] analysis is a promising approach to engineering such families of dialogs. This requires the identification of the scope of

the product families, and the determination of the common and variable properties among them.

Next, we use the case study described in this section to present a detailed discussion of the challenges involved in dynamically adapting the content and rendering of dialogs based on user context ("context-sensitive dialog synthesis") in enterprise communications middleware such as Hermes. We demonstrate how we have used model transformations for the automatic synthesis of dialogs from specific decision points in enterprise workflows. We also explain how we have applied SCV analysis to our dialog generation process in order to develop families of dialog variants.

VI.2 Design Challenges in Context-Sensitive Dialog Synthesis

In Section VI.1 we introduced a communications-enabled workflow in an insurance company. We explained the motivation behind adapting the workflow-generated dialogs to suit various communication endpoints of employees of the insurance company. We described how the target communication endpoints for the dialogs were selected based on current user context. Below we discuss some of the design challenges in automatically synthesizing dialogs based on user context.

1. Programmatic, customizable mappings for dialog creation. The dialog in our insurance example asked only one simple question to ascertain the ability, availability, and willingness of an employee to participate in a conference call. However, dialogs in general may be much more complicated and may involve a sequence of sub-interactions. Currently in Hermes, the content of a dialog is a simple text template that a workflow designer manually creates for a specific decision point in a workflow and that can be parameterized at runtime with user names, URLs, case and policy numbers, etc. The entirely manual creation and maintenance, especially of complicated dialogs for a large number of workflow decisions points, requires significant efforts.

Instead, we would like to come up with programmatic mappings from workflow decision points and user context to dialogs on specific endpoint types. User context not only determines which endpoint to send the dialog to and to render on but also ideally results in adjusting the dialog content to the specifics of a user. If, for example, one of the recipients of the insurance dialog in our above example is hearing- or vision-impaired or is most fluent in a language other than the insurance company's official business language, the dialog would ideally accommodate these user-specific parameters. Different sets of employees also have different skill sets. For example, the appraiser in our case study may receive case details as part of the dialog that are meaningful only to somebody with expertise in appraising car damages.

Thus, programmatic mappings must allow customization of the output dialogs based on parameters outside the workflow decision point and user context, such as the enterprise for which the workflow is designed, the vertical market in which the enterprise is operating, or technical constraints of the communications middleware. Section VI.3.1.1 discusses how we have resolved this challenge.

2. Dialog formatting and rendering. Even though content formatting and rendering of a dialog are inextricably intertwined, we list the determination of how to format and render a dialog separately from the content selection because the emphasis is different in both challenges. The challenge in formatting a dialog for and rendering it on a target communication endpoint is the large number of static and dynamic characteristics of endpoints, resulting in vastly different types of formatting and rendering options for basically the same dialog content.

The static characteristics include the modality (voice, IM, email, Web, SMS, plain text, etc.), processing power, screen size and resolution, type of input devices attached to the endpoint, audio/video capabilities, etc. The dynamic characteristics include current data connectivity and battery power. An explanation of how such characteristics are determined in Hermes is provided in [58]. For example, the same abstract dialog may have to be

rendered as a VXML script over a voice connection, via an HTML form on a mobile device, or as a sequence of SMS or IM messages, each one of which would require the user to reply with an SMS/IM message. Section VI.3.2 describes how our approach resolves this challenge by allowing developers to model variabilities in the dialogs so as to render them on their endpoints.

3. Response option definition. To be really useful, many dialogs require fairly differentiated or complex feedback from the user, based on response options given in the dialog. Our insurance example listed response options such as "1 hour earlier", "1 hour later", "Only if you cannot find somebody else" in addition to "Yes" and "No". Clearly, these options unreasonably limit the recipient's expressiveness in terms of the most desirable time and communication modality of and willingness to participate in the conference call. The situation is exacerbated in more complex dialogs.

There is a trade-off between the number of response options in a dialog on the one hand and user convenience and productivity on the other hand. Too few response options may frustrate the user because the response that the user might like to give is not part of the dialog. Too many response options may frustrate the user because it takes too long and too much effort to peruse and understand the given response options, and to select the most appropriate one.

4. Linking to additional documentation. Our stated goal in Section VI.1 was to render the entire dialog in a target communication endpoint, including potentially multimodal supporting documentation (Item 2 in Section VI.1), in the interest of a timely delivery of information to the user, collection of a response, and increased user productivity and convenience. However, even lengthy or rich text documentation, let alone audio/video or other multimodal documentation pertaining to the dialog, is often infeasible or too costly to render on a given endpoint. In such cases, the programmatic mapping to dialogs would have to produce a solution that leaves the supporting documentation out of dialogs but allows dialog recipients to access it as easily and quickly as possible. A fallback solution is

always to email links to the documentation to the recipient but more sophisticated options may be possible as well. For example, the recipient of a dialog on a mobile device may have the option of sending an SMS with a fax number to an enterprise server that would then send out supporting text documentation to that fax number.

5. Extending customizable mappings to new endpoints. The steady evolution of communication media and endpoints, in particular mobile devices and more powerful enterprise-class hard- and softphones, increases the complexity of managing dialogs. For example, suppose the insurance company in our example had upgraded their office phone system to IP telephones with large touch-screens. A dialog sent to such a phone may now best be rendered not as a phone call but as a rich text popup on the display with user response options rendered as touch buttons. Thus the programmatic, customizable mappings from workflow decision points to dialogs must be flexible enough to accommodate new endpoints with relatively minor changes to the mappings and negligible workflow execution downtime.

Section VI.3.1.1 describes how our approach helps modularize these mappings and separate their variabilities, and Section VI.3.2 discusses how developers can incorporate new endpoints by (partially) using existing mappings.

VI.3 Templatized Model Transformation for Dialog Customization

Section VI.1 discussed the design challenges in synthesizing context-sensitive dialogs. This section discusses details of MTS which consists of the following two stages: (1) SCV analysis, and (2) Transformation specialization. We show how it allows designing of templatized model transformations that can be used as the basis for developing and maintaining customizable, reusable, and flexible mappings for dialogs. Our approach is similar in concept to C++ class templates and Java generics but is applicable more widely to developing generalized transformation rules.

We analyzed our enterprise communications case study. We identified that the case study exhibits a number of commonalities and variabilities in communication dialogs for

various endpoints. The variabilities can stem from application structure pattern(s) in the dialog, various attribute values, and mapping rules of that map these dialogs from communication workflows.

Using MTS for templatized model transformations involves the following two main phases:

This section describes how we have used *Model transformations Templatization and Specialization (MTS)* [55], which is our templatized transformation framework, for dialog customization. At the heart of our dialog synthesis approach are two phases shown in Figure VI.1 and described below:

1. **Phase I: SCV analysis.** In this offline phase, developers analyze their transformations and identify their commonalities and variabilities across workflow structure patterns, various attribute values, and mapping rules. The result of this analysis phase is fed into the transformation in terms of a simple *constraint notation specification* discussed in detail in Section VI.3.1.1. This phase is similar to coding templatized functions in C++ that captures the pattern of the code to be generated later by the compiler.
2. **Phase II: Transformation specialization.** In this phase, the developers use higher order transformations defined in MTS to generate a variability metamodel (VMM) from their templatized model transformations. VMM is useful in creating a *specialization repository* of a particular product-line and is created in terms of VMM models. The specialization repository contains a VMM model for each communication endpoint. A combination of templatized transformation and a VMM model (corresponding to that endpoint) is used for generating the communication dialog for a specific endpoint. This phase is similar to template instantiation in C++ when the compiler automatically generates the code specific to the actual type of parameters passed to a template function.

We have used the Generic Modeling Environment (GME) [2] as the modeling environment in MTS. GME provides a general-purpose editing engine, a separate view-controller GUI, and a configurable persistence engine. GME is meta-programmable, and thus the

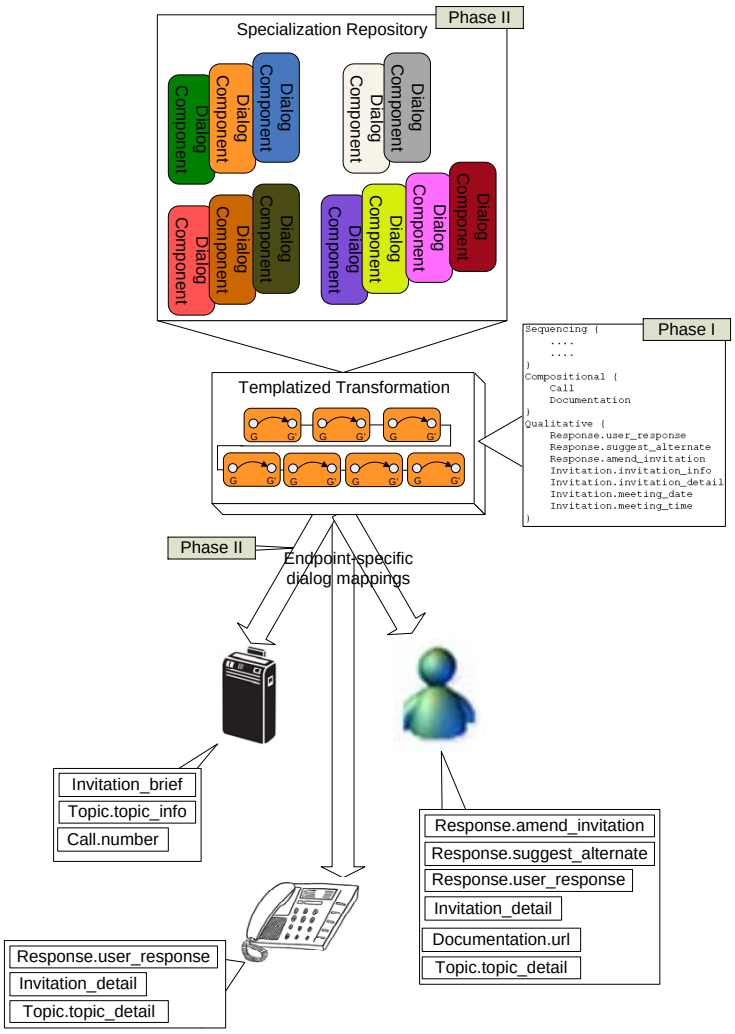


Figure VI.1: MTS: Model Transformation Templatization and Specialization

same environment used to define modeling languages is also used to build models, which are instances of the metamodels.

For defining transformation rules we have used the Graph Rewriting And Transformation (GReAT) [53] language. GReAT is developed using GME and can be used to define model-to-model transformation rules using its visual modeling language. It also provides the GReAT Execution Engine (GR-Engine) for execution of these transformation rules for generating target models.

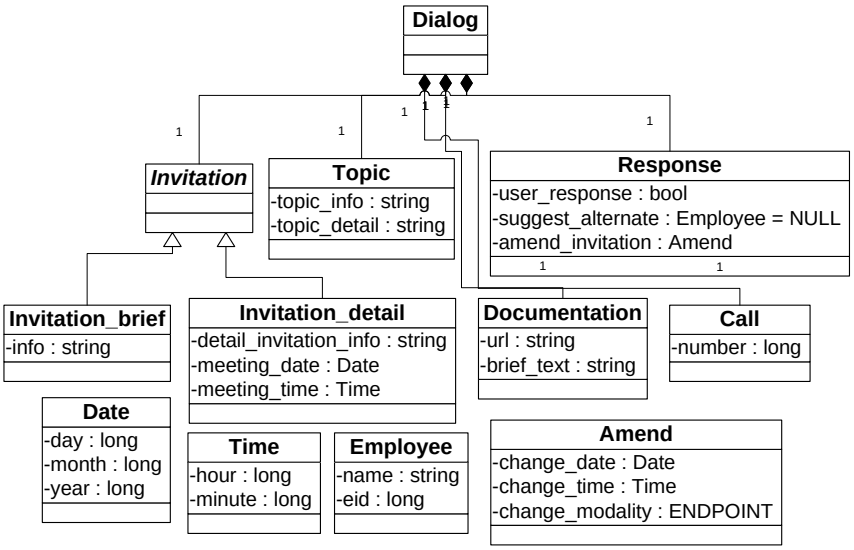


Figure VI.2: Generic dialog structure for supporting enterprise communication

Model transformations in GReAT require source and target domain-specific modeling languages (and their corresponding metamodels). A transformation developer uses the GReAT visual transformation language to define various translation rules in terms of patterns of input and output modeling objects. Finally, developers execute GReAT's transformation engine called GR-engine that translates an input model using the specified rules into an output model. The remainder of this section describes the details of our approach.

Although our templatization idea has been realized using specific modeling and transformation environments, the concepts we discuss in this chapter for supporting variabilities in transformation rules are generic and can be re-applied to other model transformation tool suites. The remainder of this section described the details of our approach.

VI.3.1 Applying MTS for Context-Sensitive Dialog Synthesis

In this section, we first explain the details of a generic dialog generated by the workflow which acts as the input for our templatized transformation. We then discuss the constraint notation in MTS, and how it can be used for separating transformation variabilities for our

customizable mappings. Finally, we show how VMM models in the specialization repository are incorporated into the middleware to yield context-sensitive dialogs for individual endpoints.

VI.3.1.1 Phase I: SCV Analysis

In Figure VI.2, we revisit the UML notation of a generic dialog in an enterprise workflow in the Hermes middleware we introduced earlier in Chapter V. The following are some of the properties/attributes in this communication dialog: (1) `User_endpoint` indicates the most active endpoint the employee has used, (2) `Call` specifies a number to call to retrieve a dialog, (3) `Documentation` contains further details about documentation pertaining to the claim in question, (4) `Topic` specifies details about the claim itself, (5) `Invitation` contains details about a conference call for discussing the claim, and (6) `Response` allows an employee to reply to the invitation in the current dialog.

Additionally, the `Response` element allows an employee to suggest an alternate employee that can be contacted about the claim in question. The latter can be done by setting `user_response` to FALSE and populating the `suggest_alternate` attribute with an appropriate value. Similarly, a user can send a request for a modified invitation to the workflow that initiated this dialog by using the `amend_invitation` enumerated data type.

Table VI.1: Dialog profiles for representative communication endpoints

Communication endpoint	Modality	Dialog properties/Attributes	
		Commonalities	**Variabilities**
Cell phone/Office phone	VXML	invitation_info, topic_info, User_endpoint	invitation_detail, meeting_date, meeting_time, user_response, topic_detail, claim_id, customer_name, customer_id, claim_date
Pager	text	invitation_info, topic_info, User_endpoint	call
Web browser	SMIL	invitation_info, topic_info, User_endpoint	user_response, suggest_alternate, Documentation, invitation_detail, meeting_date, meeting_time, topic_detail, claim_id, customer_name, customer_id, claim_date, amend_invitation

As the first step in using MTS for the templatization of a set of model transformations, i.e., endpoint-specific dialog customization mappings, the commonalities and variabilities across various elements (instances) in the set must be identified. This is crucial for the following reasons: (1) the commonalities constitute the *invariants* among the transformation instances. These commonalities can be directly used to construct common but templatized transformation rules; and (2) the variabilities constitute the dissimilarities of individual transformation instances and therefore must be separated from the templatized transformation rules so that both common transformation rules and individual mapping instance variabilities can evolve independently.

As a first step in the automated synthesis of dialogs, we must first determine the commonalities and variabilities across the elements (instances). In our model transformations approach the commonalities constitute the templatized transformation rules while the variabilities constitute the specializations. This is achieved via the SCV analysis.

SCV analysis for Dialog profiles (i.e., properties necessary to create a dialog for an endpoint) are shown in Table VI.1 for three representative communication endpoints. To cover wider range of handheld devices, we have used two endpoints `handheld_1` and `handheld_2` the later having better hardware and software capabilities and data connection. The Modality field in the Table is a static characteristic of an endpoint and indicates

the communication type used to deliver the dialog. Notice that a given endpoint may support more than one modality. The modality clearly affects the format and rendering of a dialog. For example, VXML and SMIL are W3C standard XML formats for designing interactive voice- and multimedia-based dialogs, respectively. As shown in Table VI.1, cell phones and office phones use VXML while Web browsers use the SMIL modality. Note that the modality will affect the format and rendering of a dialog.

In Table VI.1, all profiles contain at a minimum the `invitation_info`, `topic_info` and `User_endpoint` attributes. However, only the cell phone, browser, and office phone endpoint profiles contain all of the attributes in the `Topic` and `Invitation` elements of a dialog. Only the Web browser endpoint allows the user to respond with an alternate employee that can be contacted for the claim in question, or request a change in the invitation, and can optionally present documentation about the claim. Similarly, the `Call` element is present only for a pager. Note that the `User_endpoint` attribute is common for all endpoint profiles and is used in selecting the appropriate VMM in the specialization repository. We will explain the details on this model selection in Section VI.3.1.2. The general idea in using MTS for developing templatized transformations is that all the common features of a product family, here a family of dialogs, get mapped directly from the input specification as family instance-independent transformation rules. The results of the SCV analysis must then be mapped to templatized transformation rules (for the commonalities) and constraint specifications (for the variabilities). For example, during our synthesis of a dialog family the common model elements in Table VI.1 get mapped directly from a generic dialog in Figure VI.2. The variabilities from our SCV analysis results, on the other hand, must be incorporated into the transformation so that they can be subsequently used by MTS. In our insurance case study, the variabilities can be categorized as follows:

1. **Compositional variabilities,** where model elements in each family instance/member are different and variability stems from these instances getting composed using distinct model elements. For example, compositional variability exists between pager and Web browser

endpoints. Recall from Table VI.1 that the dialog profile of a pager endpoint consists of `Call`, `Invitation`, `Topic`, and `User_endpoint` elements. On the other hand, the profile of a Web browser endpoint is composed of `Invitation`, `Topic`, `Response`, `Documentation`, and `User_endpoint` elements.

2. **Qualitative variabilities,** where two family instances may share a common model element but not the absolute values of attributes of that element. The term quality here refers to what a system model describes as a whole, which is a collective aggregate of values of all of its attributes. Thus, even though the `Response` element is present in both the cell phone and the Web browser, the `suggest_alternate` and `amend_invitation` attributes are not applicable and are omitted for the cell phone (because of limited capabilities of the endpoint). For a Web browser endpoint, however, these attributes can be used in its dialog and are available. A similar variability exists for the attributes of the `Invitation` element for the office phone and pager endpoints.

In our MTS approach, the constraint notation specification is inserted as a special comment in the transformation rules that is transparent to the GR-engine and thus does not interfere with the engine's execution and translation logic. The constraint specification captures the variabilities in a model transformation as simple implication relations between source ($s \in S$) and target ($t \in T$) model elements as follows:

$s \overset{\phi}{\longmapsto} t$, where $s = \{P1(s_1, s_2, .., s_m) \mid s_{j=1..m} \in S\}$ and

$t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$

$P1$ and $P2$ define patterns of source and target elements.

The constraint specification captures the variabilities in a model transformation as simple implication relations between the source and target model elements. In our insurance case study both the source and target model elements belong to the Dialog modeling language and hence the same input dialog specification is specialized and transformed into a dialog for individual endpoints. As such, the variabilities are captured in terms of generic

dialog model elements. Below we show excerpts from a complete constraint specification for capturing the variabilities that we discussed in Items (a) and (b) above:

```
Sequencing {
   ....
   ....
}
Compositional {
   Call
   Documentation
}
Qualitative {
   Response.user_response
   Response.suggest_alternate
   Response.amend_invitation
   Invitation.invitation_info
   Invitation.invitation_detail
   Invitation.meeting_date
   Invitation.meeting_time
}


Sequencing {
    ....
    ....
}
Compositional {
    Call
    Documentation
```

```
}
Qualitative {
    Response.user_response
    Response.suggest_alternate
    Response.amend_invitation
    Invitation.invitation_info
    Invitation.invitation_detail
    Invitation.meeting_date
    Invitation.meeting_time
}
```

The specification is quite self explanatory – the `Qualitative` block captures all the attributes while the `Compositional` block captures all the model elements that vary between family members. The `Sequencing` block is used later in the specialization in Phase II and will be explained in Section VI.3.1.2.

VI.3.1.2 Phase II: Transformation Specialization

In this section, we explain the higher order transformation algorithm for generating the variability metamodel (VMM) from the constraint specification introduced in Section VI.3.1.1. We also explain the use of VMM in creating a family-specific specialization repository that easily captures all the variabilities in terms of VMM models. A combination of templatized transformation and instance-specific VMM model is used to create a model transformation for that instance.

The auxiliary function $initializeVMM(V)$ on Line 5 creates VMM V and initializes its attributes required in order to define a new language in GME. As shown in the Algorithm, for all the compositional mappings in the transformation, the source and target patterns are read in Lines 13 and 18. Next the types of each modeling object, for both source and target patterns is found by parsing the respective modeling languages as shown in Lines 15 and

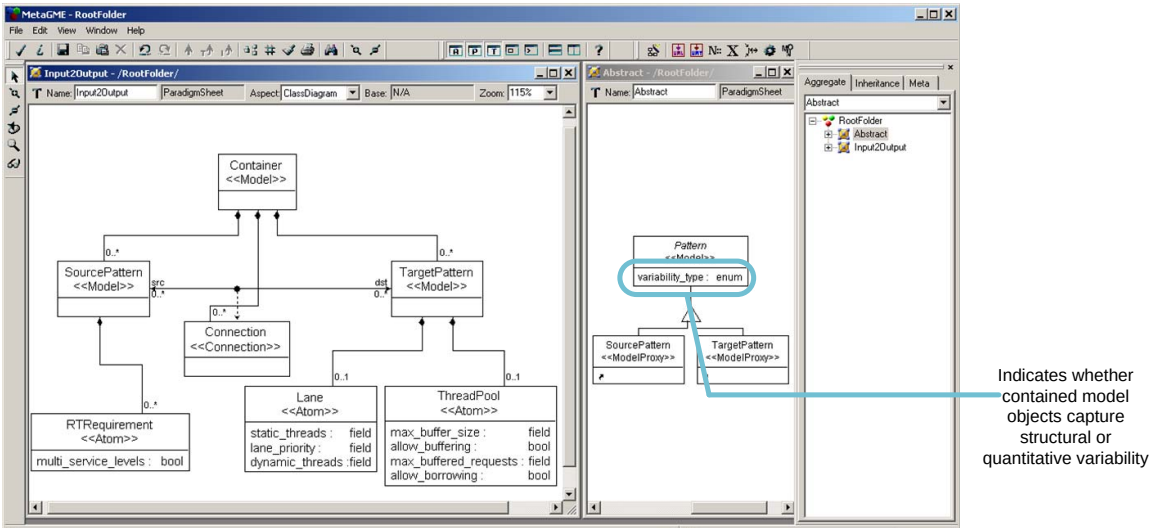


Figure VI.3: Auto-generated variability metamodel using SCV analysis results from Phase I

20. The type information is used to create appropriate modeling objects corresponding to the specified source and target patterns in Lines 16 and 21. A similar approach is taken in generating modeling objects in VMM for qualitative variabilities in constraint specification as shown in Lines 25–42. Additionally, attributes of the corresponding modeling objects are also created as shown in Lines 31–33 and Lines 38–40.

Thus, when higher order transformation represented by Algorithm 4 is applied to the transformation for our insurance case study from Section VI.3.1.1, a VMM is generated automatically for the dialog family. Figure VI.3 shows a screenshot of the generated VMM in GME. In this Figure, `InputPattern` denotes the source language pattern (e.g., it is generated in Algorithm 4 in Lines 13–16) while `OutputPattern` denotes the target language pattern (e.g., it is generated in Algorithm 4 in Lines 18–21). As stated earlier, since the same input dialog specification is refined as it is transformed in our case study, the `InputPattern` model does not contain any elements.

Variabilities are separately modeled and contained in the `Compositional` and `Qualitative` elements. The specialization repository can now be easily synthesized by developers in

terms of VMM models, where each model captures an individual dialog profile (for example, as shown in Table VI.1). Finally, VMM models are used in conjunction with our original (templatized) transformation to create context-sensitive dialogs as shown in Figure VI.1.

VI.3.2 Discussion

The mappings from workflow decision points to dialogs are highly use case-specific and largely depend on the characteristics of individual dialog family members. An MTS constraint specification allows capturing all the commonality and variability properties of these family members. Our somewhat simplistic communication dialog case study showed how MTS can be used successfully to define and maintain mappings that are customizable across compositional and qualitative dimensions as identified in its SCV analysis.

The rendering of communication dialogs can be affected for individual endpoints by modifying the specialization repository instance at modeling level (i.e., the VMM model for that endpoint). In particular, we showed how static characteristics of an endpoint (in our example its modality), can be be used to format dialogs.

Our approach can easily be extended to include *dynamic* endpoint information, such as current data connectivity levels (bandwidth) or remaining battery power, simply by updating the specialization repository. For example, Table VI.2 shows two dialog profiles for handheld devices. The `handheld_1` profile can be used for devices known to have sufficient battery and bandwidth. On the other hand, `handheld_2` does not contain detailed claim information and therefore is more applicable for devices with low batter power and bandwidth.

With MTS it is possible to control the degree of flexibility in responding to an invitation. Thus, on one hand, it can be used to define a wide range of response options set for the Web browser endpoint, and on the other, a minimal response options set (with only "Yes" and "No" allowed options) for the cell phone. Some of these response options

Table VI.2: Using dynamic endpoint characteristics in dialog formatting & rendering

comm. endpt.	modality	Dialog Prop./Attribs.
handheld_1	text	invitn_detail, meeting_date, invitn_info, topic_info, meeting_time, user_resp., User_endpt., topic_detail, claim_id, Doc., amend_invitn., customer_id, claim_date, customer_name, suggest_altern.
handheld_2	text	invitation_info, topic_info, user_response, User_endpoint

are dictated by the type of endpoints (e.g., options such as `suggest_alternate` and `amend_invitation` can not be used for the pager endpoints). For others however, the transformation developers need to perform a careful tradeoff analysis, between providing a rich feature set and increasing employee productivity. MTS allows rapid synthesis of dialogs (e.g., each with a separate response set) and thus can be a very effective tool in a tradeoff analysis.

Finally, using VMM models for specifying variability in dialog synthesis allows developers to reuse dialog customization mapping rules for new endpoints. In addition, VMM offers the following advantages: (1) both transformations and VMM models can evolve independently, and (2) changes in requirements for dialogs targeted at a particular endpoint does not require recompilation of the model transformation.

CHAPTER VII

RELATED WORK

In this Chapter we discuss our research wih existing works in middleware QoS configuration & templatization of model transformation techniques.

VII.1 Research on Middleware QoS Configuration

This section compares our work on QUICKER with existing literature on performing QoS configuration activity.

Ritter *et.al.* [100] describe CCM extensions for generic QoS support and discusses a QoS metamodel that supports domain-specific multi-category QoS contracts. Their approach allows CCM components to negotiate QoS properties (QoS contract) before they can start their normal operational interactions. The negotiation mechanism they have proposed is independent of any application domain and thus can be reused by a variety of applications. The concrete QoS contracts that are subject of the negotiation are, on the other hand, domain specific. The authors have also proposed the implementation of QoS architecture by incorporating some changes in the CCM container architecture and its language mapping. The authors claim it is possible to switch the QoS support on or off for a given component implementation.

The QML QoS specification language [31] specifies component-level QoS properties. QML is mainly concerned with how to specify the required or provided QoS for servers implementing CORBA IDL interface. It provides three main abstraction mechanisms for QoS specification: contract type, contract and profile. QML allows definition of contract types that represent specific QoS aspects, such as performance or reliability. A contract type defines the dimensions that can be used to characterize a particular QoS aspect. A

dimension has a domain of values that may be ordered, and there may be three kinds of domains: set domains, enumerated domains and numeric domains. A contract is an instance of a contract type and represents a particular QoS specification. In order to be useful with the overall application interface specification, QML profiles allow association of contracts with interfaces, operations, operation arguments and operation results.

Authors in [47] detail an approach to capture user requirements that are translated into corresponding network and system parameters. The authors propose modeling & concept for development support in the mapping activity of end user QoS onto system and network QoS. They discuss QoS agents in structured object middleware that relate end-user QoS specifications to multimedia stream application domain bindings. The authors note that the end user QoS requirements, generally a set of nonorthogonal specifications, should be supported using the available middleware QoS classes.

The work in [50], on the other hand, focuses on capturing QoS properties in terms of *interaction patterns* amongst system components that are involved in executing a particular service and supporting runtime monitoring of QoS properties by distributing them over components (which can be monitored) to realize that service. In this work, the authors have demonstrated the specification and monitoring of end-to-end QoS properties – specifically, interaction deadlines Ű by employing models that capture the cross-cutting interaction aspects of distributed, reactive systems. The authors put the functionalities (or features, services) provided by the system – as opposed to the execution components/modules that implementing them – in the center of the development process. The authors also demonstrate how to create an infrastructure for simulation and validation based on RT-CORBA. The authors have shown through this demonstration how monitoring of deadline violations can be achieved in executable specifications, which can be an important capability for checking various application properties during conformance testing of supplier-provided components. Their work on monitoring execution can be used in conjunction with the existing

techniques for system testing and formal validation techniques, such as model checking and theorem proving.

An approach that uses an aspect-oriented specification techniques for component-based distributed systems is discussed in [13]. This work deals with specification of functional behavior, non-functional behavior, QoS management policies, and requirements of the application and synthesis of QoS management components for that supporting application-level adaptation strategies. The technique discussed in [13] called aspect-oriented specification (AOS), has the specifications broken up into four different aspects: functional, nonfunctional, QoS management policies and requirements. The QoS management policies category is concerned with the ongoing management policies related to the (non-functional) QoS properties. Their approach allows different formal languages to be used, if appropriate, to specify different aspects. For such cases, the authors propose that different specifications can be brought together using formal composition rules built on a common operational semantics. Their formal composition process is similar to the aspect-weaving process of aspect-oriented programming, and the join-points of an aspect-oriented program are mirrored by the cross-synchronization of events in their composition process. A declarative approach is used to specify the system, *e.g.*, real-time temporal logic and timed automata notations are used to describe the application requirements and QoS management policies, respectively. This aspect-oriented technique is similar to QuO [135], which uses several high-level languages to capture different aspects of QoS support.

In contrast to the above works, QUICKER focuses on automating the error-prone activity of middleware QoS configuration, *i.e.*, mapping QoS requirements to QoS configuration options. Such an automation along with a flexible and intuitive QoS requirement specification mechanism naturally supports application QoS evolution during its development cycle. An interesting side effect of using model transformations for QoS configuration is that since the changes to application QoS are made only at QoS requirement specification time, the implementation platform details (*i.e.*, middleware QoS options) always remain

in-sync with the application QoS requirements, thereby addressing the productivity problem [65] at the middleware level. Finally, since the specification of the QoS requirements itself is platform-independent, it allows for reconfiguring the QoS mappings to suit other middleware platforms.

Model-driven techniques in [8, 9, 29, 38, 39, 72, 81, 133] rely on a visual interface to help developers select a wide array of middleware QoS options for their applications. Such information is later used for generating testsuites for purposes of empirical evaluation. For example, the work in [72] extended two existing languages (1) Options Configuration Modeling Language (OCML) [123], which is an MDD tool that simplifies the specification and validation of complex DRE middleware and application configurations, and (2) Benchmark Generation Modeling Language (BGML) [71], which is an MDD tool that synthesizes benchmarking testsuites to analyze the QoS performance of OCML-configured DRE systems. This work illustrated how these two languages can be used to measure the impact of middleware configurations on end-to-end DRE system performance, and evaluated how these tools help alleviate the complexities of configuring QoS-enabled middleware to support particular DRE system requirements. This work describes a process to systematically document and validate how different configurations of QoS-enabled middleware affect DRE system QoS.

The work in [133] discusses a benchmark application generation toolsuite that exploits MDA development techniques and existing cartridges using AndroMDA [69]. In this approach, the load testing behavior is modeled using a modified version of the UML Testing profile. To achieve this, they implemented various stereotypes in the UML 2.0 Testing Profile including SUT(System under Test), Test Context, Test Component etc. Each stereotype includes a set of tagged values for various purposes including correlations to the SUT, test data modeling, test scenario modeling (such as transaction mix) and performance testing configurations (number of runs, ramp up time and etc.). The AndroMDA extension for load test modeling and generation results in a new cartridge. The design and implementation of

the cartridge combines OO-based meta-modelling and domain specific language design. The authors also discuss a complete template for generating a default implementation of the loadTestAll() test case with randomly generated data based on a data pool model and transaction mix.

In contrast, our configuration process does not expose the developers to all of the configuration space of underlying middleware and relies on platform-specific heuristics for generating QoS configurations. Futher, using our process, the correctness of generated configurations is established in the design time. We argue that since our transformation algorithms codify best practices and patterns in middleware QoS configuration, QoS design and evolution throughout the system lifecycle using our approach is faster.

Analysis tools such as VEST [116], Cadena [45] and AIRES [66] evaluate whether certain timing, memory, power, and cost constraints and functional depenencies of real-time and embedded applications are satisfied. VEST [115, 117] focuses on the development of effective composition mechanisms, and the associated dependency and nonfunctional analyses for real-time embedded systems. It is based on extending the notion of aspects. Aspects [21, 62, 63] are defined as those issues that cannot be cleanly encapsulated in a generalized procedure, and typically include issues that affect the performance or semantics of components. For example, many real-time, concurrency, synchronization, and reliability issues are aspects of a distributed system. In this work, the authors have extended the notion of aspects to language independent notions and applied them at system design time. They introduce two kinds of language-independent aspects: aspect checks and prescriptive aspects. Together these permit the benefits of aspects to be exercised early in the composition process rather than in the implementation phase. The proof-of-concept has been implemented in the VEST (Virginia Embedded Systems Toolkit). VEST provides an environment for constructing and analyzing component-based DRE systems. VEST helps developers select or create passive software components, compose them into a product, map the passive components onto active structures such as threads, map threads onto

specific hardware, and perform dependency checks and nonfunctional analyses to offer as many guarantees as possible along many dimensions including real-time performance and reliability. DRE systems issues are explicitly addressed in VEST via the mapping of components to active threads and to hardware, the ability to include middleware as components, and the specification of a network and distributed nodes.

AIRES [42, 43, 66] proposed an approach for eliminating the inter-task dependencies using shared buffers between dependent tasks in DRE systems. The system correctness, with respect to data-dependency, is ensured by having each dependent task poll the shared buffers at a fixed rate. Therefore individual tasks can be allocated as well as scheduled independent of their predecessors. To meet the timing constraints of the original dependent-task system, the authors iteratively derive the polling rates based on end-to-end system deadline constraints. The overheads associated with the shared buffers and the polling mechanism are minimized by clustering tasks according to their communication and timing constraints. The authors also give simluation-based proof with the task allocation based on a simple first-fit bin packing algorithm that their approach scales almost linearly with the system size, and clustering tasks greatly reduces the polling overhead.

Cadena [17, 45, 99, 113, 114] is an integrated environment built using Eclipse for building and analyzing CCM based systems. Cadena provides a framework for lightweight dependency analysis (including both intra-component and inter-component) of behavior of components. It supports an integrated model-checking infrastructure (using Bogor) dedicated to checking global system properties using event-based inter-component communication via real-time middleware. The framework is targetted specifically towards avionics mission computing systems, though it can be applied in the context of general, CCM-based applications. It provides a number of capabilities including the following: (1) A collection of light-weight specification files that complement the IDL specification to describe

mode variable domains, intra-component dependencies, and component state-transition semantics. These files have a natural refinement order so that useful feedback can be obtained with little annotation effort, and increasing the precision of annotation yields more precise analysis. In addition, Cadena specifications allow DRE developers to specify the same information in different ways, achieving a form of checkable redundancy that can be useful for exposing design flaws. (2) Dependency analysis capabilities such that tracing inter/intra-component event and data dependencies can be done easily. It also provides as well as algorithms for synthesizing dependency-based real-time and distribution aspect information. (3) An integrated model-checking infrastructure for event-based inter-component communication via realtime middleware that enables system design models (derived from CCM IDL, component-assembly descriptions and annotations) to be model-checked for global system properties. (4) Java component stub and skeleton code generated using the CCM IDL to Java compiler. (5) A component assembly framework supporting a variety of visualization and programming tools for developing component connections. (6) A CCM deployment facility dedicated to the Boeing Bold Stroke architecture (static component connections with a real-time event-channel) that allows deployment code to be automatically generated. (7) The toolchain is implemented as plug-ins to IBMŠs Eclipse IDE, thus providing an end-to-end integrated development environment for CCM-based Java systems.

QUICKER is similar to Cadena in terms of usage of Bogor for model-checking. The difference is that Cadena applies model-checking to verify functional behavior of components, whereas QUICKER applies model-checking to verify QoS configuration options of component middleware in the presence of dynamic adaptation of these options via RACE. Our configuration process can be used as a complementary QoS design and analysis technique to these efforts since it emphasizes on mechanisms to (1) translate design-intent into

actual configuration options of underlying middleware and (2) verify that both the transformation and subsequent modifications to the configuration options remain semantically valid.

The Adaptive Quality Modeling Language (AQML) [89] provides QoS adaption policy modeling artifacts. AQML generators can (1) translate the QoS adaption policies (specified in AQML) into Matlab Simulink/Stateflow models for simulations using a control-centric view of QoS adaptation and (2) generate Contract Definition Language (CDL) specifications used in QuO [135] from AQML models. QUICKER differs with AQML in several ways, including the application of QoS adaption and the precision of the middleware modeling. For example, QUICKER models the configuration of standards-based QoS-enabled component middleware technologies, such as real-time CORBA and RT-CCM, whereas AQML targets QuO. Moreover, QUICKER's middleware model is precisely abstracts the actual real-time CORBA implementation so it does not need a two-level declarative translation (from AQML to CDL to potentially CCM using QuO delegates [135]) to achieve QoS adaptation. Finally, we employ automated model-checking in QUICKER to analyze the QoS adaptation as a function of QoS configuration options of middleware.

The Distributed QoS modeling environment (DQME) [28, 78, 129] is a DSML that enables the design of QoS adaptive applications in combination with using QoS provisioning frameworks, such as QuO [135]. DQME uses a hierarchical representation for modeling QoS adaptation strategies and supports design of controllers based on state machines. The primary difference is that DQME focuses on a high-level design of QoS adaptation strategies, whereas QUICKER's emphasis is more fine-grained and focuses on the runtime configuration options of the underlying middleware. Operating at a high-level of abstraction with respect to QoS adaptation strategies ultimately requires mapping of the design adaptation strategies to implementation-specific options. QUICKER focuses on translating high-level QoS adaptation design intent into actual QoS configuration options that exists in tools like DQME. QUICKER also helps configure QoS adaptation strategies dynamically

at runtime by feeding RACE information about valid QoS configuration states from the analysis results obtained using model-checking.

A classic example of model transformations is the Model Driven Architecture (MDA) [93] development process, which centers around defining PIMs of an application and applying (typed, and attribute augmented) transformations to PIMs to obtain PSMs. The COMQUAD project [103] discusses extensions to MDA in order to allow application developers to refine non-functional aspects of their application from an abstract point of view to a model closer to the implementation. Model transformations are defined between different non-functional aspects and are applied to QoS characteristics (*i.e.*, measurement of quality value) definitions to allow for such a refinement. In this work, the central idea of QoS specifications is the measurement or characteristic. A measurement is defined as a mapping from states, objects, or events of a physical system to a formal system. Measurements thus can be the response time (a mapping from an operation call in a running system to a real number representing the time taken from invocation to return), or confidentiality (a mapping from a channel used to transfer information to a value indicating the level of confidentiality achieved by this channel). The authors claim that by using models of the relevant aspects of target applications for the definition of measurements, the definitions themselves can be made independent of specific applications. Therefore, it follows from the above claim that they will be applicable to any system model that can be viewed as an instance of the models used in the definition of the measurement. Non-functional specifications essentially constrain measurements applied to a functional model of a system, and are thus, application specific.

Authors in [3] attempt to clearly define platform-independent modeling in MDA development by introducing an important architectural notion of *Abstract Platform* that captures an abstraction of infrastructure characteristics for models of an application at some platform-independent level in its design process. An important observation of the authors is that design languages should allow for appropriate levels of platform-independence to

be defined at each development steps. An abstract platform defines an acceptable or, to some extent, ideal platform from an application developerŠs point of view. It is an abstraction of infrastructure characteristics assumed for models of an application at some point of (the platform-independent phase of) the design process. Alternatively, an abstract platform defines characteristics that must have proper mappings onto the set of concrete target platforms that are considered for an MDA design process, thereby defining the level of platform-independence for this particular process. Defining an abstract platform forces a designer to address two conflicting goals: (i) to achieve platformindependence, and (ii) to reduce the size of the design space explored for platform-specific realization. In this work, the authors have presented some guidelines for platform-independent design and have defined requirements for design languages intended to support platform-independent design. The authors also discuss how the architectural concept of abstract platform can be supported in UML.

QUICKER differs from the above projects as follows: COMQUAD allows for specification and transformation of non-functional aspects at different levels of abstraction as the application itself evolves. For example, response time of a function call may be expressed more clearly as the time between reception of a request and sending the corresponding response, or time between reception of a request and reception of the corresponding response. Successive refinement models in COMQUAD are exposed to the application developers such that more details can be added.

Similarly, work discussed in [3] proposes that design languages should support platform-independence at each abstract platform levels. QUICKER, on the other hand, deals with mechanisms to translate QoS requirements a system places on the implementation platform onto QoS configuration options of that platform. Output models of QUICKER can be treated as *read only* models. Application developers model and modify only the high-level requirements models, and are thus shielded from the low-level details about the middleware platform. Finally, we focus on QoS requirements (and mappings thereof) of an application

at the middleware level while COMQUAD focuses on QoS characteristics for an application (*e.g.*, response time, delay, memory usage).

Research presented in [82] maps application models captured in the *Embedded Systems Modeling Language* (ESML) to UPPAAL timed automata [75] using graph transformation to verify automata using graph transformation to verify properties like schedulability of a set of real-time tasks with both time- and event-driven interactions, and absence of deadlocks in the system. Other related efforts include the *Virginia Embedded Systems Toolkit* (VEST) [116] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [42], which are model-driven analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied.

QUICKER focuses on a different level of abstraction (*i.e.*, QoS policy mapping tools) than [42, 82, 116] which are QoS analysis tools. QUICKER is complementary to these efforts since it emphasizes on mechanisms for (1) capturing system QoS requirements at domain-level abstractions to simplify QoS requirements specification, and (2) correctly translating design-intent into QoS configuration options of underlying middleware platform.

This work deals with specification of functional behavior, non-functional behavior, QoS management policies, and requirements of the application and synthesis of QoS management components for that supporting application-level adaptation strategies. A declarative approach is used to specify the system, *e.g.*, real-time temporal logic and timed automata notations are used to describe the application requirements and QoS management policies, respectively. This aspect-oriented technique is similar to QuO [135], which uses several high-level languages to capture different aspects of QoS support.

In contrast to the above works, QUICKER focuses on automating the error-prone activity of middleware QoS configuration, *i.e.*, mapping QoS requirements to QoS configuration options. Such an automation along with a flexible and intuitive QoS requirement specification mechanism naturally supports application QoS evolution during its development

cycle. An interesting side effect of using model transformations for QoS configuration is that since the changes to application QoS are made only at QoS requirement specification time, the implementation platform details (*i.e.*, middleware QoS options) always remain in-sync with the application QoS requirements, thereby addressing the productivity problem [65] at the middleware level. Finally, since the specification of the QoS requirements itself is platform-independent, it allows for reconfiguring the QoS mappings to suit other middleware platforms.

Design-time approaches to component middleware optimization include eliminating the dynamic loading of component implementation shared libraries and establishing connections between components done at runtime, as described in the static configuration of CIAO [118]. Our approach is different since it uses model transformations of application QoS configurations at design-time. Our approach is thus not restricted to optimizing just the inter-connections between components. Moreover, the static configuration approach can be applied in combination to our approach.

An approach to optimizing the middleware at design/development-time employs context-specific middleware specializations for product-line architectures has been discussed in [70]. The central idea of this work is based on utilizing various application-, middleware- and platform-level characteristics that remain *constant* or are invariant and do not vary during the normal application execution in order to reduce the excessive overhead caused by the generality of middleware platforms. Some research also exists in the area of Aspect-Oriented Programming (AOP) that relies on automatically deriving most appropriate subsets of middleware platforms depending on the application use-case requirements [48], and modifying applications so as to bypass the middleware layers using aspect-oriented extensions to CORBA Interface Definition Language(IDL) [96]. Additionally, other researchers have *constructed* the middleware in a "just-in-time" fashion by integrating source code analysis, and inferring features and synthesizing implementations [130] for achieving optimizations.

Contrary to the above approaches, our model transformation-based technique relies only on the specified (1) application QoS configuration and (2) the initial deployment plan, in order to optimize the QoS policies. Our approach does not necessitate any modifications to the application, *i.e.*, the application developer need not design his/her application tuned for a specific deployment scenario. As our results in Section IV.3.3 have indicated, our approach can be used in a complementary fashion to any of the design/development-time approaches discussed above, since there exist several opportunities for QoS optimization at various stages in application development.

Research on approaches to optimizing middleware at runtime on the other hand, has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context [27]. The work on QuO [52, 97, 124, 134, 135] is relevant in this context as it is a dynamic QoS framework that allows dynamic adaptation of desired behavior specified in *contracts*, selected using proxy objects called *delegates* with inputs from runtime monitoring of resources by *system condition* objects. QuO has been integrated into component middleware technologies, such as LwCCM.

Other aspects of runtime optimization of middleware include domain-specific middleware scheduling optimizations for DRE systems [36], using feedback control theory to affect server resource allocation in internet servers [132] as well as to perform real-time scheduling in Real-time CORBA middleware [80]. Our work 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 runtime.

Runtime approaches to application-specific optimizations have focused on data replication for edge services, *i.e.*, replicating servers at geographically distributed sites [35]. 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 [87] for dynamic negotiation of best communication mechanisms between any requester and provider of a service. Other research [112] 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 approach makes use of the application deployment information on each node of the target domain and is thus focused on optimizing the execution of the components at each end-system as opposed to optimizing the on-the-wire protocol.

Deployment-time optimizations research such as [77] have focused on optimization of web services. This research is aimed at 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. It uses techniques such as *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 QoS optimizations achieved in our transformation-based approach are non-trivial to perform using the above mentioned approach. Italso relies on modification to the application source code to rewrite the application code, while our approach is non-intrusive and does not require application source code modifications, and it only relies on the specified application policies and deployment plans.

VII.2 Research on Model Transformation Templatization

Existing model transformation tools [15, 32, 107] support some form of higher order transformations. PROGRES and ATL allow specification of type parameters while VIATRA allows development of meta-transformations, *i.e.*, higher order transformations that can manipulate transformation rules and hence model transformations. Unlike MTS however, these tools do not provide mechanisms for separation of variabilities from model transformations to facilitate automated development of application families.

The model driven architecture (MDA) [65, 93] development process is centered around defining application platform-independent models and applying (typed, and attribute augmented) transformations to these models to obtain application platform-specific models. In the context of MDA, requirements and challenges in generating specialized transformations from generic transformations are discussed in [68].

Reflective model driven engineering (MDE) approach [12] proposes a two dimensional MDA process by expressing model transformations in a tool- or platform-independent way and transforming this expression into actual tool- or platform-specific model transformation expressions.

There is return on investment (ROI) associated with developing and maintaining mappings from platform-independent transformations to platform-specific transformations in terms of reuse, composition, customization, maintenance etc. The authors argue that the ROI for a two-dimensional MDA process is greater than conventional one dimensional MDA.

Although reflective MDE focuses on having durable transformation expressions that naturally facilitate technological evolution and development of tool-agnostic transformation projects, mappings still have to be evolved with change in platform-specific technologies. MTS, however, is concerned with managing and evolving model transformation

variability in systems developed using an MDA process. Parameterization techniques supported by MTS can be highly effective in managing variability of mappings from platform-independent to specific forms in the context of the above body of work.

Asset variation points discussed in [104] deal with expressing variability in models of product lines [18]. A variation point is identified by several characteristics (*e.g.* point reference, and context, use and rationale of the variation point) that uniquely identify that point in the product lines. These asset variation points capture variation rules of implementation components of a product-line member. The authors define processes, methods and techniques investigated in expressing the variability between products and its usage to derive new products from the software product line. In order to describe the variability, the authors identify various development concerns:

1. Expression of what can vary in the asset, called the variable asset;
2. Expression of why it can vary, rather than how it can vary using variability rules tated using variability attributes;
3. Decision to take to realize the variability using variability mechanism relating the transformation to apply in order to select or to build a variant;
4. Decision to take to select variants, supported by decision points ordered in a decision model.

In addition, they also list two different kinds of variabilities that must be accounted for:

1. Under-specification: leaving variability unspecified. This solution guides the application engineer but leaves great flexibility to the programmer.
2. Provision: specifying variability and providing elements to help the choice of solutions at derivation time by engineers. All elements should be documented.

The authors also proposed to clearly identify the product characterization and the product building as separate processes. The authors further claim that product characterization can be done with the help of a decision model, that can be provided with a simple semantic of a graph made of decision points with inclusion/exclusion relationships – the graph in turn

forms a decision plane. From this product characterization an application model can be drawn that forms the ground from which the strategy of variability resolution is built. The resolution of variability inside the domain engineering assets is driven by the variability resolution plane. The derivation attributes come out of this plane, which are consumed by the variation points. From these attributes, variability resolution rules that are part of variation pointŠs specification drive the transformation of the assets they are attached to.

An aspect-oriented approach to managing transformation variability is discussed in [125] that relies on capturing variability in terms of models and code generators. This work explores an approach that integrates model-driven and aspect-oriented techniques in order to facilitate variability implementation, management and tracing in SPLE. The general approach is as follows: (1) Express the set of artifacts in software development in terms of application models. It is beneficial to maximize the size of this set as it lends itself to the use of model transformations, (2) the automated translations from problem space to solution space are encoded as model transformations, that enable formal descriptions of mappings and repeatability in their execution, (3) variable parts of the resulting system are either assembled from pre-build assets generated from models or implemented via interpreters, (4) aspect-oriented modeling is used to implement variability in models for supporting the selective adaptation of models, (5) aspect-oriented programming is used to implement crosscutting features on code level that cannot easily be modularized in the generator, (6) certain parts of a product will still be implemented manually because, for economic reasons, developing a custom generator is too costly. The manually written code is integrated with the generated code in well-defined ways.

Another approach is model weaving [41], which is used in the composition of separate models that together define the system as a whole. Aspect models allow specifying variability that is weaved into a base model to form an instance of a product-line. Using the aspect-oriented approach requires developers to learn a new modeling language for creating aspect models for their product-line.

In contrast, the VMM models generated by MTS use modeling objects that are part of the source (or target) modeling languages requiring no additional learning curve. MTS generates VMM from variability specification in the templatized transformation to automate the entire process. The population of VMM models itself, as shown in Section VI.3, does not involve learning an entirely new language since all its modeling objects are part of a source (or target) modeling language of the transformation.

Research presented in [19, 20] discusses how user interfaces can be customized based on user context information. The authors employ a model-based development process to model user communication (in terms of interactions) with a context-aware system. Services such as context-sensitive guided tours using users' mobile devices can be developed using their prototypical approach. In [19], the authors extend their earlier approach to designing context-sensitive user interfaces for static context, such that it is possible to design and provide runtime support for user interfaces that can be affected by dynamic context changes. The dynamic context changes takes into account the target platform, network properties and other environmental conditions. Additionally, the authors also provide solutions on how to design a UI for a service, and how to cope with this service when it becomes available to the application on the portable device of the user. The proof-of-concept DynaMo-AID which is part of the Dygimes User Interface Creation Framework has been applied to a representative case study to illustrate its practical use.

A number of context-aware services and frameworks have been proposed over the years [24, 86, 122] that incorporate users' location and availability, and awareness information while establishing communications between them. For example, the connector service in [24] aims at establishing communication between two users at the most appropriate time, using the most appropriate endpoints, and takes factors such as physical location, social relations and current state of the users into account [24]. Our previous work in

this area [58] discussed a Web browser-based dialog system that facilitated user communications in response to events in the enterprise workflow so as to improve and accelerate decision-making.

In contrast to the above body of work, our research focuses on customizing dialogs such that they can be appropriately rendered on user endpoints. User context, in our case, is very specific to the users' endpoints and communication devices and encompasses endpoint characteristics such as hardware, software, and network capabilities, remaining battery life, keyboard support and other parameters, as opposed to user location etc.

Our work can be used in conjunction with user interface customization approaches such as [20] when different kinds of user endpoints are allowed to use the context-aware service.

CHAPTER VIII

CONCLUDING REMARKS

Middleware QoS Configuration With the trend towards implementing key DRE system infrastructure at the middleware level, achieving the desired QoS is increasingly becoming more of a configuration problem than (just) a development problem. The flexibility of configuration options in QoS-enabled component middleware, however, has created a new set of challenges. Key challenges include determining the correct set of values for the configuration options, understanding the dependency relations between the different options, and evolving the QoS configurations with changes to application functionality.

QUICKER MDE Toolchain To address these challenges, we have developed the *QUality of service pICKER* (QUICKER) toolchain, which (1) uses model transformation to automate the mapping of application QoS policies into middleware-specific QoS configuration options and (2) applies model-checking to ensure that the QoS configuration options are valid at the individual component level as well as at the application level. To demonstrate the use of QUICKER, we applied it to address configuration challenges in representative DRE system case studies. We also showed how QUICKER's QoS mapping capabilities and validation of QoS options using model-checking enabled the successful configuration and deployment of DRE system components. The following is a summary of lessons learned from our experience using QUICKER to develop this prototype:

QoS mapping is critical to successful deployment of systems built using component middleware. With the increase in configuration complexity, the QoS mapping capabilities provided by QUICKER are essential to managing the complexity. Configuration of middleware options to achieve the desired QoS in DRE systems can be viewed as an directed acyclic graph whose root is the high-level mission requirements, edges are the individual

mappings joining the vertices in a top-down fashion, and the vertices correspond to the different options available at each intermediate layer of abstraction. QUICKER is a part of a chain of mappings starting from high-level mission requirements to the actual deployment platform, and resides between the application components and the underlying component middleware implementation. By employing MDE tools, QUICKER not only simplifies the QoS mapping process for DRE system developers, it also preserves the rich semantics associated with the mapping between the QoS policies and QoS configuration options at this level. Using MDE tools also helps QUICKER integrate with mapping technologies that exist both *above* (*e.g.*, mission requirement mapping tools, functional decomposition tools, and functional analysis tools) and *below* (*e.g.*, deployment planning tools) the level at which QUICKER operates in a component-based DRE system development lifecycle.

Integration of QoS mapping with runtime entities like runtime QoS controllers essential to ensure dynamic configuration. In addition to QUICKER toolchain capabilities described in this proposal, our ultimate goal is to provide inputs to runtime QoS controllers, such as those in RACE. The current version of the RACE controller [109] performs coarse-grained control of CCM components by changing component priorities to effect control. Managing resource utilization by controlling priority alone, however, does not cover the entire spectrum of resource control capabilities. In particular, response time of the controller is also critical for DRE systems. To enable fine-grained control of CCM components, therefore, we are extending the QUICKER toolchain to incorporate a cost model for dynamic resource adaptation and automatic generation of a RACE controller based on results of the Bogor model-checker.

Horizontal mapping of QoS is as important as vertical QoS mapping. QUICKER currently focuses on mapping application QoS policies onto a single underlying middleware technology: the CIAO and RACE RT-CCM platform. Large-scale DRE systems—particularly systems requiring dynamic resource management [74]—are often composed of heterogeneous technologies. It is therefore essential for QoS mapping tools to not only

support *vertical mapping* (*i.e.*, the mapping of policies and validation onto a single technology) but also *horizontal mapping* (*i.e.*, the mapping of QoS policies onto multiple heterogeneous technologies, while reconciling the differences between these technologies). Until such mapping is performed, QoS configuration and associated tools will remain as islands, which significantly complicates integration efforts for large-scale DRE systems.

GT-QMAP Model Transformation Algorithms Large-scale distributed systems are increasingly built using middleware technologies that provide reusable building blocks and services to support rapid software development by composition. In order to configure them correctly for different application needs, these middleware platforms provide highly customizable QoS mechanisms.

In this dissertation we introduced an automated, reusable model-driven QoS mapping toolchain that (1) raises the level of specification abstraction for system developers (who lack a detailed understanding of these QoS mechanisms and their inter-dependencies) such that system QoS requirements can be expressed intuitively, and (2) correctly maps these QoS specifications to middleware-specific QoS configuration options.

In this dissertation we discussed our approach to evaluating correctness and effectiveness of a QoS configuration process in the context of a representative DRE system. We showed how structural correspondence between input and output languages in our model-driven approach can be used to establish that initial system requirements are correctly mapped to middleware QoS options. We verified the correctness of generated QoS options using a model-checker and empirically showed that they are effective in satisfying system requirements.

Templatized Model Transformation In this dissertation we discussed a model transformation-based approach to customizing dialogs between enterprise workflows and users to a variety of user communication endpoints, from cell phones to Web browsers to office phones. Our two-phase approach to dialog specialization offers the following benefits: (1) It allows developers to separate variabilities in their dialog mappings in Phase I such that templatized

model transformations can be developed. (2) Through use of VMM models in the specialization repository in Phase II, developers can easily create family instance-specific dialogs for individual endpoints, and extend existing mappings such that dialogs for new endpoints could be synthesized.

The following is a summary of lessons learnt from our work:

• **Mapping of workflows to context-sensitive communication is essential to rapid decision-making in enterprises.** With the increasing reliance on automated processes in enterprises, there is an immediate need to accelerate the communication between workflows and enterprise employees. Such communication enables employees to make informed business decisions with lesser "turnaround time", which ultimately leads to increased overall productivity and efficiency of the enterprise. Our MTS approach provides a simple, extensible solution to context-sensitive dialog creation. The templatized transformation together with the specialization repository are useful in automatically mapping workflow decision points onto appropriate dialogs. Since the variabilities are expressed as VMM models, addition of dialogs, corresponding to new endpoints introduced in the enterprise, can be achieved simply by creating a new VMM model.

• **Templatized transformations & specializations can be more widely applicable to development of PLAs in other domains.** The MTS approach in this work has been demonstrated specifically in the context of context-sensitive dialog synthesis. However, the MTS toolchain, its various development processes and artifacts are not domain-specific and can be re-targeted for other domains. Thus, the toolchain itself can be applied in general to any product-line development scenario, without requiring any change. An effort is underway in applying the MTS approach to configuration of heterogeneous component-based distributed applications [55, 57].

This dissertation presented MTS (Model-transformation Templatization and Specialization), which is an enabling technology that seamlessly integrates with existing model transformation tools to support reusable model transformations for application families.

Existing model transformation tools lack support for reusable transformations which force developers to reinvent transformation rules. MTS overcomes these limitations while requiring no change to contemporary tools. MTS defines templatized transformations to factor out the commonalities, and uses the notion of a generated variability metamodel to capture the variabilities in the transformation process across variants of an application family. MTS defines two higher order transformations to specialize the transformations for different variants. Although our existing prototype is implemented in GReAT, it can be easily extended for other model transformation toolchains as long as they provide means to develop higher order transformations. Results of evaluating MTS indicate that it enhances developer productivity and effectiveness of model-based software development for application families.

APPENDIX A

LIST OF PUBLICATIONS

Our research on QUICKER and MTS has lead to the following conference and workshop publications.

A.1 Refereed Conference Publications

1. **Amogh Kavimandan**, Reinhard Klemm, and Aniruddha Gokhale, "Context-Sensitive Dialog Synthesis for Enterprise Workflows Using Templatized Model Transformations," in *The Twelfth IEEE International Enterprise Computing Conference (EDOC 2008)*, Munchen, Germany, September 15-19, 2008.

2. **Amogh Kavimandan**, and Aniruddha Gokhale, "Evaluating the Correctness and Effectiveness of a Middleware QoS Configuration Process in Distributed Real-time and Embedded Systems," in *The Eleventh IEEE International Symposium on Object/Component/Service oriented Real-time Distributed Computing (ISORC 2008)*, Orlando, FL, May 5-7, 2008, pp. 100–107.

3. **Amogh Kavimandan**, and Aniruddha Gokhale, "Automated Middleware QoS Configuration Techniques using Model Transformations," in *The Fourteenth IEEE International Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, April, 2008, pp. 93–102.

4. **Amogh Kavimandan**, Reinhard Klemm, Dort'ee Duncan Seligmann, and Aniruddha Gokhale, "Enhancing Enterprise User Productivity with Embedded Context-Aware Voice Applications," in *The IEEE International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2007)*, Papeete, French Polynesia, November 4-9, 2007, pp. 169–176.

5. **Amogh Kavimandan**, Aniruddha Gokhale, "A Model-driven QoS mapping tool for QoS-enabled Component Middleware," (short paper) in *Proceedings of The Tenth ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, Nashville, TN, September-October, 2007.

6. **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 *The Tenth IEEE International Symposium on Object/Component/Service oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 7-9, 2007, pp. 62–70.

7. **Amogh Kavimandan**, Reinhard Klemm, Ajita John, Dort'ee Duncan Seligmann, and Aniruddha Gokhale, "A Client-Side Architecture for Supporting Pervasive Enterprise Communications," in *The International Conference on Pervasive Services (ICPS 2006)*, Lyon, France, June 26-29, 2006, pp. 222–232.

8. **Amogh Kavimandan**, Wonsuck Lee, Marina Thottan, Aniruddha Gokhale, and Ramesh Viswanathan, "Network Simulation via Hybrid System Modeling: A Time-Stepped

Approach," in *The Fourteenth International Conference on Computer Communications And Networks (ICCCN 2005)*, San Diego, CA, October 17-19, 2005, pp. 531–536.

9. **Amogh Kavimandan**, Aniruddha Gokhale, "An Energy-efficient and Scalable Data Dissemination Protocol for Wireless Sensor Networks," (short paper) in *Proceedings of the Third International Conference on Mobile Systems, Applications and Services (MobiSys 2005),* Seattle, WA, June 6-7, 2005.

10. **Amogh Kavimandan**, and Aniruddha Gokhale, "Applying Model-driven Generative Programming to Communication Network Performance Evaluation," (short paper) in *Proceedings of The Global Telecommunications Conference (GLOBECOM 2005)*, St. Louis, MO, November-December, 2005.

A.2 Refereed Workshop Publications

1. **Amogh Kavimandan**, and Aniruddha Gokhale, "Templatized Model Transformations for Middleware QoS Configuration of Heterogeneous DRE Systems," in *Proceedings of OMG's Real-time Systems Workshop (OMG-RTWS 2008)*, Washington D.C., July, 2008.

2. **Amogh Kavimandan**, and Aniruddha Gokhale, "A Parameterized Model Transformations Approach for Automating Middleware QoS Configurations in Distributed Real-time and Embedded Systems," in *Proceedings of ASE workshop on Automating Service Quality, (WRASQ 2007)*, Atlanta, GA, November 6, 2007.

3. **Amogh Kavimandan**, and Aniruddha Gokhale, "Automated Middleware QoS Configuration Techniques using Model Transformations," in *Proceedings of the EDOC workshop on Advances in Quality of Service Management, (AQuSerM 2007)*, Annapolis, MD, October 15-19, 2007.

4. **Amogh Kavimandan**, and Aniruddha Gokhale, "Supporting Systems QoS Design and Evolution through Model Transformations," in *Proceedings of Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA Companion 2007)*, Montreal, Canada, October 21-25, 2007.

5. **Amogh Kavimandan**, Krishnakumar Balasubramanian, Nishanth Shankaran, Aniruddha Gokhale, and Douglas C. Schmidt, "A Model-driven QoS Mapping Tool for QoS-enabled Component Middleware," in *Proceedings of OMG's Real-time Systems Workshop (OMG-RTWS 2007)*, Washington D.C., July, 2007.

6. **Amogh Kavimandan**, Marina Thottan, Aniruddha Gokhale, Wonsuck Lee, and Ramesh Viswanathan, "SeMA: A model-driven Multi-paradigm Integrated Simulation Framework For Analysis of Communication Networks," in *Proceedings of the OMGŠs First Annual Model-Integrated Computing (MIC) workshop*, Exploring the synergy between MIC and MDA, Washington D.C., October, 2004.

REFERENCES

[1] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, Barcelona, Spain, 2004.

[2] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.963443.

[3] Joao Paulo Almeida, Remco Dijkman, Marten van Sinderen, and Luís Ferreira Pires. On the Notion of Abstract Platform in MDA Development. In *Proceedings of the 3rd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 253–263, September 2004.

[4] 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.

[5] Jaiganesh Balasubramanian, Balachandran Natarajan, Douglas C. Schmidt, Aniruddha Gokhale, Gan Deng, and Jeff Parsons. Middleware Support for Dynamic Component Updating. In *International Symposium on Distributed Objects and Applications (DOA 2005)*, Agia Napa, Cyprus, October 2005.

[6] Jaiganesh Balasubramanian, Sumant Tambe, Balakrishnan Dasarathy, Shrirang Gadgil, Frederick Porter, Aniruddha Gokhale, and Douglas C. Schmidt. Netqope: A model-driven network qos provisioning engine for distributed real-time and embedded systems. In *RTAS' 08: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–122, Los Alamitos, CA, USA, 2008. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/RTAS.2008.32.

[7] Krishnakumar Balasubramanian and Douglas C. Schmidt. Physical Assembly Mapper: A Model-driven Optimization Tool for QoS-enabled Component Middleware. In *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium*, pages 123–134, St. Louis, MO, USA, April 2008.

[8] 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, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.4.

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

[10] 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. *Journal of Computer Systems Science*, 73(2):171–185, 2007. ISSN 0022-0000. doi: dx.doi.org/10.1016/j.jcss.2006.04.008.

[11] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. The ATL Transformation-based Model Management Framework. In *Research Report, ATLAS Group, INRIA and IRIN*, September 2003.

[12] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective Model Driven Engineering. In *Proceeding of The* 5^{th} *International Conference on Unified Modeling Language, Modeling Languages and Applications*, pages 175–189, October 2003.

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

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

[15] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *Companion of the* 18^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003*. ACM, 2003.

[16] B. H. C. Cheng, R. Stephenson, and B. Berenbach. Lessons learned from automated analysis of industrial uml class models (an experience report). In *Model Driven Engineering Languages and Systems, 8th International Conference (MoDELS 2005)*, volume 3713, pages 324–338, 2005.

[17] Adam Childs, Georg Jung, Gurdip Singh, Jesse Greenwald, John Hatcliff, Matthew B. Dwyer, Prashant Kumar, Venkatesh Ranganath, and Xianghua Deng. Supporting Model-driven Development of Component-based Embedded Systems with Cadena, August 15 2003. URL citeseer.ist.psu.edu/664688.html;www.cs.iastate.edu/~leavens/SAVCBS/2003/papers/invited/dwyer.pdf.

[18] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[19] Tim Clerckx, Kris Luyten, and Karin Coninx. Dynamo-aid: A Design Process and a Runtime Architecture for Dynamic Model-based User Interface Development. In *Engineering Human Computer Interaction and Interactive Systems Lecture Notes in Computer Science*, volume 3425/2005, pages 77–95. Springer Berlin/Heidelberg, July 2005.

[20] Tim Clerckx, Chris Vandervelpen, Kris Luyten, and Karin Coninx. A Prototype-Driven Development Process for Context-Aware User Interfaces. In *Task Models and Diagrams for Users Interface Design Lecture Notes in Computer Science*, volume 4385/2007, pages 339–354. Springer Berlin/Heidelberg, August 2007.

[21] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using aop to improve os structure modularity. *Communications of the ACM*, 44(10):79–82, 2001. ISSN 0001-0782.

[22] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

[23] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006. ISSN 0018-8670.

[24] Maria Danninger, G. Flaherty, Keni Bernardin, Hazim Kemal Ekenel, T. Köhler, Robert Malkin, Rainer Stiefelhagen, and Alex Waibel. The Connector: Facilitating Context-aware Communication. In *Proceedings of the 7th International Conference on Multimodal Interfaces (ICMI 2005)*, pages 69–75, Trento, Italy, October 2005. ACM.

[25] Dionisio de Niz and Raj Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems*, 2(3):196–208, 2006.

[26] Gan Deng, Chris Gill, Douglas C. Schmidt, and Nanbor Wang. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. In I. Lee, J. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[27] 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.

[28] Jianming Ye *et al.* A Model-Based Approach to Designing QoS Adaptive Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2247-5. doi: dx.doi.org/10.1109/REAL.2004.7.

[29] Tao Lu *et al.* CoSMIC: An MDA Tool suite for Application Deployment and Configuration. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, October 2003. ACM.

[30] François Terrier Frédéric Thomas, Jérôme Delatour and Sébastien Gérard. Toward a Framework for Explicit Platform-Based Transformations. In *Proceedings of the* 11^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, FL, USA, May 2008.

[31] Svend Frolund and Jari Koistinen. Quality of Service Specification in Distributed Object Systems. *IEE/BCS Distributed Systems Engineering Journal*, 5:179–202, December 1998.

[32] G. Csertán and G. Huszerl and I. Majzik and Z. Pap and A. Pataricza and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of* 17^{th} *IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, 2002. IEEE.

[33] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, pages 446–453, Charlottesville, VA, September 2003.

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

[35] 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: doi.acm.org/10.1145/775152.775217.

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

[37] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002. ACM.

[38] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang. Model Driven Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*, pages 163–187. Wiley and Sons, New York, 2004.

[39] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind S. Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA)*, 73(1):39–58, 2008.

[40] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. Domain-Specific Modeling. In *CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.)*, pages 7.1–7.20. CRC Press, May 2007.

[41] Jeffrey Gray, Ted Bapty, and Sandeep Neema. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, pages 87–93, October 2001.

[42] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A model-based approach to system-level dependency and real-time analysis of embedded software. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 78, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1956-3.

[43] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A Model-Based Approach to System-Level Dependency and Real-time Analysis of Embedded Software. In *RTAS'03*, pages 78–85, Washington, DC, May 2003. IEEE.

[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.

[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*, pages 160–172, Portland, OR, May 2003.

[46] George T. Heineman and William T. Councill, editors. *Component-based Software*

Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70485-4.

[47] Cristian Hesselman, Ing Widya, Aart van Halteren, and Bart Nieuwenhuis. Middleware Support for Media Streaming Establishment Driven by User-Oriented QoS Requirements. In *Lecture Notes in Computer Science: Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'00)*, volume 1905, pages 158–171, Enschede, The Netherlands, October 2000. Springer-Verlag.

[48] 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.

[49] Institute for Software Integrated Systems. Component Synthesis using Model Integrated Computing (CoSMIC). www.dre.vanderbilt.edu/cosmic, Vanderbilt University, Nashville, TN.

[50] Jaswinder Ahluwalia and Ingolf H. Krüger and Walter Phillips and Michael Meisinger. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Fifth ACM International Conference On Embedded Software*, pages 100–109, Jersey City, NJ, September 2005. ACM.

[51] Ajita John, Reinhard Klemm, Ankur Mani, and Doree Seligmann. Hermes: A Platform for Context-Aware Enterprise Communication. In *Proceedings of the 3rd International PerCom Workshop on Context Modeling and Reasoning (CoMoRea)*, Pisa, Italy, March 2006. IEEE.

[52] David A. Karr, Craig Rodrigues, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt. Application of the QuO Quality-of-Service Framework to a Distributed Video Application. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001. OMG.

[53] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformations in the Formal Specification of Computer-Based Systems. In *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, Huntsville, AL, April 2003. IEEE.

[54] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003. www.jucs.org/jucs_9_11/on_the_use_of.

[55] Amogh Kavimandan and Aniruddha Gokhale. A Parameterized Model Transformations Approach for Automating Middleware QoS Configurations in Distributed

Real-time and Embedded Systems. In *Proceedings of ASE Workshop on Automating Service Quality, (WRASQ 2007)*, Atlanta, GA, November 2007.

[56] Amogh Kavimandan and Aniruddha Gokhale. Automated Middleware QoS Configuration Techniques using Model Transformations. In *Proceedings of the* 14^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 93–102, St. Louis, MO, USA, April 2008.

[57] Amogh Kavimandan and Aniruddha Gokhale. Automated Middleware QoS Configuration Techniques using Graph Transformations. Technical Report ISIS-07-808, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, May 2007.

[58] Amogh Kavimandan, Reinhard Klemm, Ajita John, Doree Seligmann, and Aniruddha Gokhale. A Client-Side Architecture for Supporting Pervasive Enterprise Communications. In *Proceedings of the IEEE International Conference on Pervasive Services (ICPS) 2006*, pages 222–232, Lyon, France, June 2006. IEEE.

[59] 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: dx.doi.org/10.1109/ISORC.2007.50.

[60] Amogh Kavimandan, Reinhard Klemm, and Aniruddha Gokhale. Automated Context-sensitive Dialog Synthesis for Enterprise Workflows using Templatized Model Transformations. In *Proceedings of the 12th International Conference on Enterprise Computing (EDOC '08)*, pages 159–168, Munchen, Germany, September 2008. IEEE.

[61] Amogh Kavimandan, Anantha Narayanan, Aniruddha Gokhale, and Gabor Karsai. Evaluating the Correctness and Effectiveness of a Middleware QoS Configuration Process in Distributed Real-time and Embedded Systems. In *Proceedings of the* 11^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2008)*, pages 100–107, Orlando, FL, USA, May 2008.

[62] 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.

[63] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. URL citeseer.nj.nec.com/kiczales01overview.html.

[64] John Kinnebrew, Nishanth Shankaran, Gautam Biswas, and Douglas Schmidt. A Decision-Theoretic Planner with Dynamic Component Reconguration for Distributed Real-Time Applications. In *Poster paper at the Twenty-First National Conference on Artificial Intelligence*, Boston, MA, July 2006.

[65] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture(MDA^{TM}): Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Apr 2003. ISBN 978-0321194428.

[66] 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.

[67] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Transformation Language. In *Proceedings of the* 1^{st} *International Conference on Theory and Practice of Model Transformations (ICMT 2008)*, pages 46–60, Zurich, Switzerland, July 2008.

[68] Jernej Kovse. Generic Model-to-Model Transformations in MDA: Why and How? In *Proceeding of* 1^{st} *OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, November 2002.

[69] Joel Kozikowski. A Bird's Eye View of AndroMDA. galaxy.andromda.org/docs-3.1/contrib/birds-eye-view.html.

[70] 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*, pages 205–218, Leuven, Belgium, April 2006.

[71] Arvind S. Krishna, Douglas C. Schmidt, Adam Porter, Atif Memon, and Diego Sevilla-Ruiz. Validating Quality of Service for Reusable Software via Model-integrated Distributed Continuous Quality Assurance. In *Proceedings of the 8th International Conference on Software Reuse*, pages 286–295, Madrid, Spain, July 2004. ACM/IEEE.

[72] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, March 2005. IEEE.

[73] Arvind S. Krishna, Nanbor Wang, Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Gautam Thaker. CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations. *Journal of Real-time Systems*, 24, 2005.

[74] 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.

[75] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[76] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proceedings of the IEEE Real-time Systems Symposium (RTSS 99)*, pages 315–326, Phoenix, AZ, December 1999.

[77] 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.

[78] Joseph Loyall, Jianming Ye, Sandeep Neema, and Nagabhushan Mahadevan. Model-based design of end-to-end quality of service in a multi-uav surveillance and target tracking application. In *Second RTAS Workshop on Model-Driven Embedded Systems (MoDES '04)*, May 2004.

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

[80] 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.

[81] Tao Lu, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. CoSMIC: An MDA Tool suite for Application Deployment and Configuration. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, October 2003. ACM.

[82] 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.

[83] Tom Mens, Pieter Van Gorp, Daniel Varro, and Gabor Karsai. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *Lecture Notes in Computer Science: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT'05)*, volume 152, pages 143–159, Tallinn, Estonia,

September 2006. Springer-Verlag.

[84] Microsoft. .NET Web Services Platform. www.microsoft.com/net.

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

[86] Allen E. Milewski and Thomas M. Smith. Providing Presence Cues to Telephone Users. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*, pages 89–96, Philadelphia, PA, December 2000.

[87] 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: doi.acm.org/10.1145/1013367.1013401.

[88] Anantha Narayanan and Gabor Karsai. Verifying Model Transformations by Structural Correspondence. Technical Report ISIS-07-809, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, Dec 2007.

[89] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale. Generators for Synthesis of QoS Adaptation in Distributed Real-time Embedded Systems. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 236–251, Pittsburgh, PA, October 2002.

[90] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*. Object Management Group, OMG Document formal/2008-01-08 edition, January 2008.

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

[92] Object Management Group. *Lightweight CCM RFP*. Object Management Group, realtime/02-11-27 edition, November 2002.

[93] *Model Driven Architecture (MDA)*. Object Management Group, OMG Document ormsc/2001-07-01 edition, July 2001.

[94] *Model Driven Architecture (MDA) Guide V1.0.1*. Object Management Group, OMG Document omg/03-06-01 edition, June 2001.

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

[96] 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: doi.acm.org/10.1145/1218563.1218567.

[97] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2000. IEEE/IFIP.

[98] I. Pyarali, D.C. Schmidt, and R.K. Cytron. Techniques for enhancing real-time corba quality of service. *Proceedings of the IEEE*, 91(7):1070–1085, July 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2003.814616.

[99] Venkatesh Prasad Ranganath, Adam Childs, Jesse Greenwald, Matthew B. Dwyer, John Hatcliff, and Gurdip Singh. Cadena: enabling CCM-based application development in Eclipse. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 20–24, 2003. URL doi.acm.org/10.1145/965665.

[100] Tom Ritter, Marc Born, Thomas Unterschütz, and Torben Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS'03)*, page 318, Honolulu, HI, January 2003.

[101] Robby, Matthew Dwyer, and John Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of the* 4^{th} *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003. ACM.

[102] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific Publishing Company, jan 1997. ISBN 9810228848.

[103] Simone Röttger and Steffen Zschaler. Model-Driven Development for Non-functional Properties: Refinement Through Model Transformation. In *Proceedings of the* 7^{th} *International Conference on Unified Modelling Language: Modelling Languages and Applications (UML 2004)*, pages 275–289, October 2004.

[104] Serge Salicki and Nicolas Farcet. Expression and usage of the variability in the software product lines. In *Proceeding of The* 4^{th} *International Workshop on Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2002. ISBN 3-540-43659-6.

[105] Douglas C. Schmidt and Fred Kuhns. An Overview of the Real-time CORBA Specification. *IEEE Computer Magazine, Special Issue on Object-oriented Real-time*

Computing, 33(6), June 2000.

[106] 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.

[107] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Progres: Language and environment. In H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, pages 487–550. World Scientific Publishing Company, 1999.

[108] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[109] 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.

[110] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions*, volume 576, pages 353–370, Aug 2000.

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

[112] 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: dx.doi.org/10.1109/ISORC.2005.23.

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

[114] Gurdip Singh, Bob Maddula, and Qiang Zeng. Event Channel Configuration in Cadena. In *Proceedings of the IEEE Real-time/Embedded Technology Application Symposium (RTAS)*, Toronto, Canada, May 2004. IEEE.

[115] John A. Stankovic, Hexin Wang, Marty Humphrey, Ruiquing Zhu, Ramasubramaniam Poornalingam, and Chenyang Lu. VEST: Virginia Embedded Systems Toolkit. In *Proceedings of the IEEE Real-time Embedded Systems Workshop*, London, UK,

December 2001. IEEE.

[116] 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.

[117] 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 *Proc. of RTAS'03*, page 58, Washington, DC, USA, 2003. ISBN 0-7695-1956-3.

[118] 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: dx.doi.org/10.1109/REAL.2004.53.

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

[120] Dipa Suri, Adam Howell, Nishanth Shankaran, John Kinnebrew, Will Otte, Douglas C. Schmidt, and Gautam Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[121] Sumant Tambe, Akshay Dabholkar, Amogh Kavimandan, and Aniruddha Gokhale. A Platform Independent Component QoS Modeling Language for Distributed Real-time and Embedded Systems. Technical Report ISIS-07-809, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, June 2007.

[122] John C. Tang, Nicole Yankelovich, James Begole, Max Van Kleek, Francis C. Li, and Janak R. Bhalodia. ConNexus to Awarenex: Extending awareness to mobile users. In *Proceedings of the International Conference on Computer Human Interaction (CHI 2001)*, pages 221–228, Seattle, WA, April 2001. ACM.

[123] Emre Turkay, Aniruddha Gokhale, and Bala Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd Annual Southeast Conference*, pages 166–170, Huntsville, AL, April 2004. ACM.

[124] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. Quo's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed*

Processing (Middleware'98), The Lake District, England, September 1998. IFIP.

[125] Markus Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC)*, Kyoto, Japan, September 2007.

[126] 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.

[127] Xiaorui Wang, Yingming Chen, Chenyang Lu, and Xenofon Koutsoukos. FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization controlstar, open. *Journal of Systems and Software*, 80(7):938–950, 2007.

[128] Duangdao Wichadakul. *Q-Compiler: MetaData QoS-Aware Programming and Compilation Framework*. PhD thesis, University of Illinois at Urbana Champaign, 2003.

[129] Jianming Ye, Joseph Loyall, Richard Shapiro, Richard Schantz, Sandeep Neema, Sherif Abdelwahed, Nagabhushan Mahadevan, Michael Koets, and Denise Varner. A Model-Based Approach to Designing QoS Adaptive Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2247-5. doi: dx.doi.org/10.1109/REAL.2004.7.

[130] 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: doi.acm.org/10.1145/1052898.1052904.

[131] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

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

[133] Liming Zhu, Ngoc Bao Bui, Yan Liu, and Ian Gorton. MDABench: Customized benchmark generation using MDA. *Journal of Systems and Software*, 80(2):265–282, February 2007.

[134] John A. Zinky, David E. Bakken, and Richard Schantz. Overview of Quality of Service for Objects. In *Proceedings of the Fifth Dual Use Conference*. IEEE, May 1995.

[135] 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.