MODEL-DRIVEN FAULT-TOLERANCE PROVISIONING FOR COMPONENT-BASED DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

By

Sumant Tambe

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, 2010

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Douglas Schmidt

Dr. Gabor Karsai

Dr. Jeffrey Gray

Dr. Janos Sztipanovits

To my parents Uma and Uday, brother Siddharth, and wife Archana
for their love and encouragement over the years

ACKNOWLEDGMENTS

While embarking the journey to the doctoral degree more than five years ago, I was anything but certain. The marathon that started with the fears of becoming the ”nameless hero” has elevated me to the new heights of expanded horizon. Looking back, however, I have realized that all these years I was never lonely and isolated because of a wonderful gift bestowed upon me: the people around! My graduate studies at Vanderbilt University would not have been possible without the personal and technical support of numerous people, who shaped me as a person and have led me where I am now.

First and foremost, I would like to thank my advisor Dr. Aniruddha Gokhale (Andy) for providing me tremendous support and timely advice for my graduate study, research, and career development. Andy spent countless hours with me discussing deep technical issues, developing and criticizing new research ideas, reviewing research papers and presentations, and guiding me towards the successful completion of the PhD. I am very grateful to Andy for cultivating an exceptionally open and stress-free advisor-student relationship with every student in the group. Next, I would like to thank Dr. Douglas Schmidt (Doug), for providing me an opportunity to work with him in the Distributed Object Computing (DOC) group at Vanderbilt. Doug’s pioneering work has been a constant source of inspiration for me from the early days of my study at Vanderbilt University.

I would like to express my thanks to the rest of my committee members, Dr. Janos Sztipanovits, Dr. Gabor Karsai, and Dr. Jeff Gray for agreeing to serve on my dissertation committee. I am especially grateful for the time Jeff devoted to reviewing my dissertation and would like to thank him for his help.

Early in my PhD I was fortunate to work with highly talented researchers from industrial laboratories, such as Lockheed Martin Advanced Technology Laboratories (ATL)

and Telcordia Applied Research. Working with Thomas Damiano from ATL and Balakrishnan Dasarathy from Telcordia helped determine the direction of my work, which later came to fruition into some of the most insightful ideas presented in this dissertation.

My stay in the Vanderbilt University was very enjoyable due to the following past and present students: Arundhati Kogekar, Dimple Kaul, Krishnakumar Balasubramanian, Nishanth Shankaran, Amogh Kavimandan, Santosh Katwal, and Rohit Marawar. I would like to make a special mention of Jaiganesh Balasubramanian (Jai), Nilabja Roy, and Akshay Dabholkar for being an integral part of the daily evening coffee sessions, which brewed some of the most hilarious jokes on graduate student life we all shared.

Finally, I would like to acknowledge my parents, brother, and wife for providing support throughout the uphill course of my graduate study. To them, I dedicate this thesis.

ABSTRACT

Developing distributed real-time and embedded (DRE) systems require effective strategies to simultaneously handle the challenges of networked systems, enterprise systems, and embedded systems. Component-based model is gaining prominence for the development of DRE systems because of its emphasis on composability, reuse, excellent support for separation of concerns, and explicit staging of development phases. Despite the advances in component technology, developing highly available DRE systems remains challenging because of several reasons; First, availability concerns crosscut functional, deployment, and other QoS concerns of DRE systems, which makes reasoning about simultaneous QoS requirements extremely difficult. Second, fault-tolerance provisioning affects nearly all the phases of system lifecycle including specification, design, composition, deployment, configuration, and run-time. Codifying the availability requirements in system artifacts corresponding to the various lifecycle phases remains challenging due to lack of a coherent approach. Finally, multi-tier architecture and non-deterministic behavior of DRE systems combined with the need to meet end-to-end deadlines even during failures give rise to unique end-to-end reliability issues. General-purpose middleware infrastructures often do not support such highly domain-specific end-to-end reliability and failure recovery requirements.

This dissertation presents a model-driven framework to coherently address the issues arising during the development of highly available component-based DRE systems. First, a domain-specific modeling language called Component QoS Modeling Language (CQML) is presented that separates systemic concerns, such as composition, deployment, and QoS to enhance comprehension and design-time reasoning. Second, a multi-stage model-driven process named GeneRative Aspects for Fault Tolerance (GRAFT) is presented that synthesizes various system artifacts to provision domain-specific end-to-end reliability and recovery semantics using model-to-model, model-to-text, model-to-code transformations.

Finally, the orphan request problem arising due to the side-effects of replication in the context of non-deterministic stateful components is addressed. This dissertation presents *Group-failover* protocol that ensures that the data in multi-tier real-time systems is both consistent and timely even in the case of failures.

Although model-driven engineering (MDE) is used extensively in this dissertation, effective techniques for a key step in MDE, model traversal, are still maturing. In the course of this research, limitations in the current model traversal approaches were addressed in Language for Embedded Query and Traversal (LEESA), which is presented here as a language-centric solution for writing succinct, generic, reusable model traversals.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

ABSTRACT v

LIST OF TABLES xi

LIST OF FIGURES xii

Chapter

I. Introduction 1

I.1. Emerging Trends and Technologies 1
I.2. High-availability Requirements of DRE Systems 3
I.3. Overview of Research Challenges 5
I.3.1. Crosscutting High-availability Concerns 5
I.3.2. Reconciling High-availability Semantics in System Architecture 7
I.3.3. Run-time Side-effects of Replicated Invocation in Non-deterministic Systems 8
I.4. Research Approach 9
I.5. Resolving Solution Domain Challenges: Object Structure Traversal 11
I.6. Dissertation Organization 12

II. Aspect-oriented Modeling for Modularizing QoS Concerns 14

II.1. Related Research 14
II.2. Unresolved Challenges 18
II.2.1. Tight coupling of functional and QoS concerns 18
II.2.2. Lack of support for variable failover granularity 19
II.2.3. Lack of support for mixed-mode replication strategies . 19
II.2.4. Lack of intuitive mechanisms for network-level QoS specification 20
II.3. Solution Approach: Component QoS Modeling Language (CQML) 20
II.3.1. Overview of CQML 20
II.3.2. Identifying Invariant Properties of Component-based Structural Modeling Languages 21
II.3.3. Extensible Design of CQML 23

II.3.4. An Abstract Join Point Model for Component Modeling Languages 25
II.3.5. Instantiating Abstract Join Point Model Using A Concrete Structural Modeling Language 25
II.4. Modeling Fault-tolerance Requirements using CQML 27
II.4.1. Design Considerations 27
II.4.2. Modeling Notation 30
II.5. Modeling Network-level QoS Requirements using CQML 33
II.6. Evaluating Composability of CQML 34

III. Weaving Dependability Concerns in System Artifacts 38

III.1. Related Research 38
III.2. Unresolved Challenges 40
III.2.1. Lack of support for incremental model refinement for multi-QoS-aware modeling 40
III.2.2. Lack of middleware support for domain-specific recovery semantics 41
III.2.3. Lack of support for auto-generation of full spectrum of fault-tolerance infrastructure 43
III.2.4. Lack of support for deployment-time network QoS provisioning 44
III.3. Solution Approach: GeneRative Aspects for Fault Tolerance 45
III.3.1. Overview of GRAFT 45
III.3.2. Stage 1: M2M transformation for multi-QoS-aware refinement of availability models 47
III.3.3. Stage 2: Automated replica placement for high-availability 50
III.3.4. Stage 3: M2M transformation for weaving monitoring infrastructure 52
III.3.5. Stage 4: Automatic weaving of code for fault-masking and recovery 56
III.3.6. Stage 5: Automatic synthesis of deployment metadata for high-availability 58
III.4. Evaluation of GRAFT 62
III.4.1. Case-study for GRAFT 62
III.4.2. Evaluating savings in effort to specialize middleware . . 64
III.4.3. Qualitative validation of runtime behavior 67
III.5. Deployment-time Network QoS Provisioning Framework 68
III.6. Evaluation of Network QoS Provisioning Framework 70
III.6.1. Case-study for NetQoS 70
III.6.2. Evaluating Model-driven Network QoS Provisioning . . 72

IV. End-to-end Reliability of Non-deterministic Stateful Components 77

IV.1. Introduction 77
IV.2. Related Research 80

IV.2.1. Integrated transaction and replication services 80
IV.2.2. Enforcing determinism 82
IV.3. Unresolved Challenges 83
IV.4. System and Fault Models 84
IV.5. The Architecture of the Group-failover Protocol 86
IV.5.1. Transparent failover 87
IV.5.2. Eliminating orphans 88
IV.5.3. Ensuring state consistency 91
IV.6. Evaluating the Merits of the Group-failover Protocol 97
IV.6.1. Overhead measurements in fault-free scenarios 98
IV.6.2. Client-perceived failover latency in failure scenarios . . 100

V. Resolving Solution Domain Challenges: Object Structure Traversal 101

V.1. Introduction 101
V.2. Related Research 102
V.3. Unresolved Challenges and Overview of the Solution Approach . 103
V.4. Language for Embedded Query and Traversal (LEESA) 105
V.4.1. Hierarchical Finite State Machine (HFSM) Language: A Running Example 105
V.4.2. An Axes-Oriented Notation for Object Structure Traversal 106
V.4.3. Programmer-defined Processing of Intermediate Results Using Actions 112
V.4.4. Generic, Recursive, and Reusable Traversals Using Strategic Programming 113
V.4.5. Schema Compatibility Checking 116
V.5. The Implementation of LEESA 118
V.5.1. The Layered Architecture of LEESA 118
V.5.2. Externalizing Static Meta-information. 120
V.5.3. The Implementation of Strategic Traversal Schemes. . . 122
V.6. Domain-specific Error Reporting using C++ Concepts 125
V.6.1. Early Type-checking of C++ Templates using Concepts 126
V.6.2. Schema Compatibility Checking Using Concepts and Metaprogramming 126
V.7. Evaluation of LEESA 128
V.7.1. Case-study 1: Evaluating Programmer Productivity . . . 128
V.7.2. Case-study 2: Evaluating Compile- and Run-time Performance 130
V.8. Comparing LEESA with Related Research 136

VI. Concluding Remarks 140

Appendix

A. Underlying Technologies 142

A.1. Overview of Lightweight CCM 142
A.2. Overview of Component Middleware Deployment and Configuration 144
A.3. Overview of Generic Modeling Environment (GME) 146
A.4. Overview of Constraint-Specification Aspect Weaver (C-SAW) . 147
A.5. Overview of C++ Template Metaprogramming 148
A.6. Overview of C++ Concepts 149

B. List of Publications 150

B.1. Refereed Journal Publications 150
B.2. Refereed Conference Publications 150
B.3. Refereed Workshop Publications 151
B.4. Technical Reports 152
B.5. Submitted for Publication 152

REFERENCES 153

LIST OF TABLES

Table Page

1. Comparison of Capabilities of Selected Three Modeling Languages . . . 36

2. Enhanced QoS Aspect Modeling Capabilities of Composite Languages PICML′, J2EEML′, and ESML′ 37

3. Savings in Fault-tolerance Modeling Efforts in Developing MHS Casestudy Without/With GRAFT 64

4. Savings in Fault-tolerance Programming Efforts in Developing MHS Casestudy Without/With GRAFT 65

5. Comparison of Manual Efforts Incurred in Conventional and Model-driven NetQoS Approaches 75

6. Overhead of the eager strategy (fault-free) (jitter +/- 3%) 98

7. Difference in the actual and perceived execution times in the lag-by-one strategy (fault-free) (jitter +/- 3%) 99

8. Client-perceived failover latency of the state synchronization strategies . 100

9. Child and parent axes traversal using LEESA (v can be replaced by an instance of a programmer-defined visitor class.) 108

10. The set of basic class template combinators 114

11. Assertions in LEESA for checking schema compatibility 117

12. Reduction in code size (# of lines) due to the replacement of common traversal patterns by LEESA expressions. 129

13. Comparison of the static metrics. (A) = LEESA and (B) = Object-oriented solution 133

LIST OF FIGURES

Figure | Page

1. An operational string 2
2. Process Model for Reusing CQML for QoS Modularization and Weaving 21
3. A Feature Model of Composition Modeling Language 22
4. Declarative QoS Aspect Modeling Capability of CQML 23
5. Metamodel of CQML. 24
6. Composing CQML's Abstract Component Model with a Base Language Using Inheritance 26
7. Availability Requirements Modeling in CQML 30
8. Shared Risk Group Hierarchy Modeling in CQML 33
9. Network-level QoS Modeling Capabilities of CQML 35
10. GRAFT's Multi-stage Process for Weaving Fault-tolerance Concerns in System Architecture Models 46
11. Automated Model Weaving Using C-SAW and FailOverUnit Replication Specification Using ECL 48
12. Automatic Weaving of Monitoring Components Using Embedded Constraint Language Specification 53
13. Automated Generation of Failure Detection and Handling Code 57
14. Complexity of connection generation 61
15. A Distributed Processing Unit Controlling Conveyor Belts 63
16. Runtime Steps Showing Group Recovery Using GRAFT 67
17. NetRAF's Network Resource Allocation Capabilities 69
18. Network Configuration in an Enterprise Security and Hazard Sensing Environment 71

19. An orphan request caused by the failure of non-deterministic component A 78

20. A group of orphan components 86

21. A failover unit spanning two components (B and C). 90

22. Eager state synchronization strategy 93

23. Lag-by-one state synchronization strategy 95

24. Meta-model of Hierarchical Finite State Machine (HFSM) language (left) and a simple HFSM model (right) 106

25. Outlines of child/parent axes traversals (Squares are statemachines, circles are states, triangles are time objects, and shaded shapes are visited.) . 109

26. Graphical illustration of FullTD and FullBU traversal schemes. (Squares, circles, and triangles represent objects of different types) 115

27. Layered View of LEESA's Architecture (Shading of blocks shown for aesthetic reasons only.) 118

28. The software process of developing a schema-first application using LEESA. (Ovals are tools whereas shaded rectangular blocks represent generated code) . 121

29. Compile-time recursive instantiations of the `children` function starting at `All<Strategy>::apply<State>(arg)` when arg is of type `State`. 124

30. Comparisons of compilation times with LEESA and the pure object-oriented solution 134

31. Comparison of meta-programming performance of different C++ compilers 134

32. Run-time performance comparisons of LEESA and the pure object-oriented solution 135

33. Layered LwCCM Architecture 143

34. An Overview of OMG Deployment and Configuration Model 145

35. Overview of GME 147

CHAPTER I

INTRODUCTION

I.1 Emerging Trends and Technologies

The software systems in several mission-critical domains, such as shipboard computing environments [120], avionics mission computing [124], multi-satellite missions [135], and intelligence, surveillance and reconnaissance missions [123] are known as *Distributed Real-time and Embedded* (DRE) systems. Such systems combine the challenges of networked systems (*e.g.,* distribution, dynamic environments, and non-determinism), enterprise systems (*e.g.,* high throughput, high availability, and security), and embedded systems (*e.g.,* resource constrained and stringent quality of service (QoS) such as low latency and jitter). Moreover, these systems exhibit *static* as well as *dynamic* variations in their QoS requirements in response to the planned and unplanned events in their mission. Examples of planned events include mission mode changes whereas unplanned events include failure of resources and transient overloads.

To develop such DRE systems, *quality of service (QoS)-enabled* middleware based on standards like Real-time Common Object Request Broker Architecture (RT-CORBA) [96] and the Real-Time Specification for Java (RTSJ) [22] have been used. More recently, QoS-enabled component middleware, such as the Lightweight CORBA Component Model (CCM) [98] and PRiSm [125], have been used [124] to build DRE systems.

Component-based development model [61, 138] is particularly suitable for large-scale DRE systems because of its emphasis on composability, reuse, excellent support for separation of concerns, and explicit staging of the development phases. Composability enables the reuse of commodity-off-the-shelf (COTS) software components. Separation of concerns, particularly QoS concerns, simplifies the development of mission-critical functionality while provisioning QoS strictly via configuration of the underlying middleware

in the subsequent stages of system development. As a result, the ability to provision the QoS *transparently* (*i.e.,* without affecting component source code) becomes one of the most sought after features of QoS-enabled component middleware such as CCM and PRiSm.

The support for composability in contemporary component middleware allows system integrators to realize the functionality by orchestrating the components in a workflow and *wiring* them together by means of configuration metadata. In the case of QoS-intensive DRE sytems, this workflow of components gives rise to a so-called *operational string model* [77]. An example operational string is shown in Figure 1.

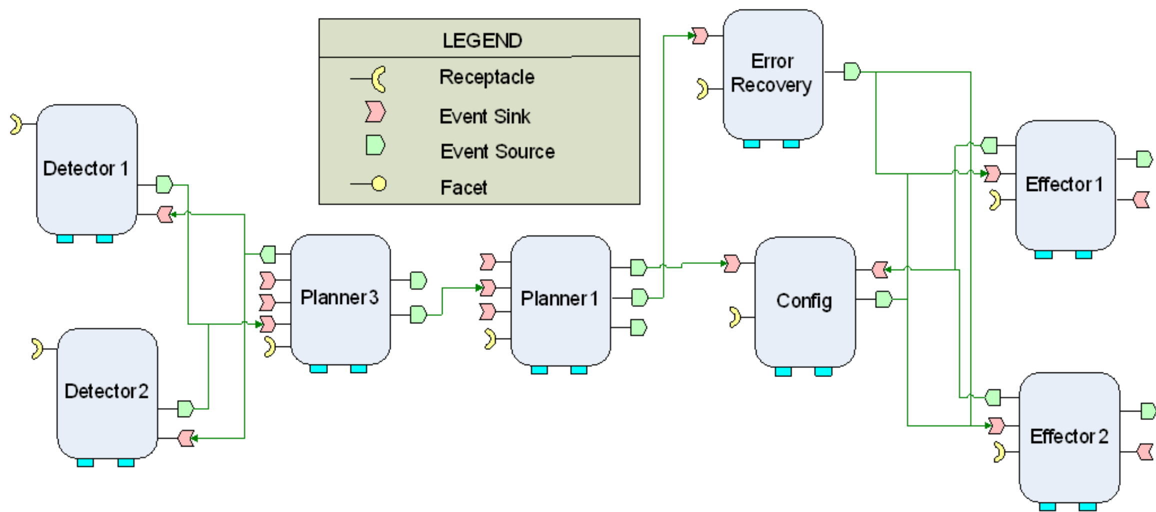

Figure 1: An operational string

The operational string model is a form of multi-tier distributed computing that is focused on end-to-end QoS. More formally, the operational strings model can be expressed as end-to-end task chains [82]. The functionality in an operational string is distributed across multiple components and the execution of the functionality often has a soft real-time deadline that must be met. For instance, the operational string in Figure 1 has a domain-imposed deadline for the sequence of execution from the *Detectors* to *Effectors*. The deadline is considered a *soft* deadline because unlike the *hard* deadline, the consequences of missing a soft

deadline are not catastrophic. Instead, missing a soft deadline diminishes the value to the client gradually to zero. After a deadline is missed (say due to failures), earlier completion of the request has higher value to the client than later completion.

To satisfy the end-to-end response time, not only the end-to-end schedulability [134] of computing resources is ensured but also the network-level QoS via bandwidth reservations [31]. DRE systems often involve dozens of distinct operations strings, which execute various domain-functionalities at different QoS requirements. Resources in such systems are managed at the granularity of operational strings as opposed to individual components. As a consequence, the operational strings become the foci of functionality, deployment, and QoS configuration management.

I.2 High-availability Requirements of DRE Systems

To ensure the end-to-end response time at run-time, DRE systems need to plan for potential failures and also recover in a timely fashion when failures actually occur. Anything that compromises the end-to-end response time of the deployed operational strings is a threat to the system's reliability and must be resolved at run-time. Several factors such as fail-stop failures of processes, computing resources, and lack of network resources may halt the execution of an end-to-end operational string. Likewise, an abrupt increase in the dynamic workload may cause network and/or CPU overload leading to missed deadlines.

Realizing reliable DRE systems requires addressing these problems at the level of operational strings as they are the unit of functionality, deployment, and QoS management. Clearly, the fault-tolerance requirements and network-level resource also need to be managed at the same level of abstraction so that the distributed functionality, its end-to-end QoS, and its reliability can be reasoned about, configured, and managed coherently.

Reasoning about fault-tolerance of an operational string is significantly harder than that of a single component for the following reasons. With distribution, the probability of a complete failure reduces but the probability of a partial failure increases [69]. A

partial execution of a client's request due to a failure in a middle-tier component may render the state across the operational string inconsistent. Upon recovery, the state in the subset of components that executed the client's request must be made consistent with the components that did not execute the request at all. The problem is exacerbated when one or more components in an operational string are non-deterministic. Recovering such a group of distributed components involves messaging overhead, which is often not permissible in real-time systems with stringent timeliness requirements.

To meet the soft real-time deadlines, client components need to failover to one of the surviving replicas of the operational string. Such failover of a group of components may be triggered even if the primary operational string is not able to satisfy its end-to-end response-time requirements due to resource overloads or unavailabilities. Moreover, group failover has to happen atomically to ensure state consistency of the group of components that failover. Contemporary fault-tolerant middleware infrastructures often lack the support for a *failover unit* that is larger than a single component and therefore lacks support for the recovery of the same.

Finally, supporting *application-transparent* failover of a group of components is important to extend the benefits of separation of concerns [37] provided by component-based middleware to highly available operational strings. Separation of concerns not only expedites the development of individual software components but also simplifies QoS planning necessary in the later stages of the DRE system lifecycle. DRE systems require such flexibility because it simplifies planning for a graceful degradation in their QoS as opposed to an abrupt denial of service. For instance, redundant operational strings could be deployed in a surveillance system differing only in their QoS. A primary operational string and its underlying resources could have been configured for high-resolution, low-latency image processing whereas one or more alternate operational strings could be configured using gradually inferior QoS to be used only if the primary operational string fails.

In summary, transparent fault-tolerance provisioning for operational strings with stringent QoS requirements is a hard problem, which requires resolution of several research challenges described next.

I.3 Overview of Research Challenges

The research challenges in provisioning fault-tolerance for component-based DRE systems are encountered in all stages (*e.g.,* specification, composition, deployment, and runtime) of the development lifecycle. Some of the key research issues include:

I.3.1 Crosscutting High-availability Concerns

Replication [60] is the most fundamental approach for provisioning high-availability in software-based systems. Multiple *identical* instances of system functionality are deployed so that the system can switch to the surviving functionality in case of failures. Replicated functionality has an impact on the non-replicated functionality because the client components (presumably non-replicated) need to failover (transparently or otherwise) on the surviving functionality in case of failures. Consequently, system availability concern is tangled with the system composition concerns. Replication is often applicable at multiple levels of granularity such as single component, group of components, an operational substring, the whole operational string, and hierarchical operational strings. Moreover, design decisions such as the *degree of replication*, which consists of the initial, minimum, and maximum number of replicas also affect the composition of the overall system.

Highly available distributed systems also require fault-monitoring infrastructure that detects failures in subparts of the system and help initiate recovery. Depending upon the types of faults, the monitoring infrastructure also scales in granularity from component-level to operational string level. Therefore, the monitoring infrastructure must also be composed with the original functionality.

High-availability provisioning also implies careful planning of the placement of system functionality across CPU and network resources so that the overall availability of the system improves while reducing the probability of simultaneous failure. Clearly, the availability concern crosscuts the deployment decisions. Deployment further affects how the CPU and network resources are allocated to ensure the end-to-end QoS of all the redundant instances of system functionality. To ensure QoS upon failure, the surviving operational strings also require predictable CPU and network-level resource availability. Therefore, static and dynamic resource management [71, 87] also need to consider the impact of replication on the scheduling of resources.

The resource allocation concern is also affected by the style of replication involved. *Active* and *passive* replication [60] are the two predominant ways of implementing high-availability for software systems. Active replication often consumes more resources than passive replication because redundant computations are performed concurrently, often at different hosts. Network resources must also be reserved for these redundant computations. Due to the resource-intensive nature of active replication, DRE systems often favor passive replication, which does not involve concurrent execution but need to recover the failed functionality by reconstituting the application-level state. Passive replication trades recovery time to improve resource consumption and utilization of limited resources. However, even in the case of passive replication network-level resource reservations are necessary to ensure timely synchronization of application-state from the primary operation string to their replicas.

Clearly, to prevent the scattering of availability concerns across other concerns, novel modularization mechanisms are necessary.

I.3.2 Reconciling High-availability Semantics in System Architecture

An instance of a component-based DRE system can be partitioned into two subparts: (1) the component instances that implement the functionality of the system and (2) the meta-data for deploying, inter-connecting, and configuring the components prior to the activation of the system. To achieve transparent provisioning of fault-tolerance for component-based DRE systems, both kinds of artifacts must be instrumented *automatically* in a coherent fashion. Automatic instrumentation of the components masks the faults from the programmer and improves the client-perceived availability of the system.

Fault-masking must be achieved with with minimal impact on the end-to-end QoS (*i.e.*, response time). Depending upon the style of replication, the fault-masking strategy varies. For instance, passive replication often requires re-invocations of the remote call if it fails. In the case operational strings, however, the failure of the operational string may not be immediately apparent to the client components that are not directly connected to the failing component. Such indirectly connected components need to failover to the replica functionality in a timely manner to begin re-execution of the failed invocation. As a result, enabling transparent failover of a group of components requires coordination between fault-masking and fault-detector modules unlike single component failover.

Interception is a dominant technique applied to achieve application-transparent failover. Several flavors of interception such as linker-level [89], ORB-level [93], container-level [26], service-level [103], and aspect-oriented [68] have been used in the past. However, in the case of QoS-intensive component-based DRE systems, a low-overhead interception mechanism that can be integrated seamlessly and automatically in the fabric of deployment infrastructure is needed.

To realize the availability requirements transparently, the metadata used by the underlying component middleware must also be instrumented automatically. The metadata determines how the components are deployed and how their interconnections are setup for

remote invocations. As noted earlier, the placement of the operational strings and their participant components must be designed carefully to improve the probability of survival of the overall system. The decisions of the intelligent placement algorithms must be codified in the deployment descriptors. Furthermore, non-replicated parts of the system need to establish redundant connections to the replicated functionality it may failover to upon failure. Codifying these decisions manually is often error-prone and time-consuming. Therefore, automated support is highly desirable.

Finally, the configuration metadata not directly related to the availability of an operational string may need manipulation to ensure proper CPU and network resource allocations for the redundant functionality. Therefore, a *multi-QoS-aware* mechanism is needed that provides opportunities to manipulate metadata throughout the development lifecycle.

I.3.3 Run-time Side-effects of Replicated Invocation in Non-deterministic Systems

Component-based DRE systems often exhibit non-determinism and maintain internal state across invocations. Moreover, systems based on operational strings are *multi-tier* in the sense that a component that serves as a provider of an interface itself acts as a client of another component. Such functional dependencies give rise to a nested chain of invocations at run-time.

Stateful multi-tier systems, when replicated for high availability, must be guarded against the side-effects of replication. Depending upon the style of replication the side-effects may vary. For instance, in active replication where a request is concurrently executed at multiple replicas, multiple nested invocations as well as multiple replies to the client must be suppressed. Moreover, consistency of the state of each replica must be ensured particularly in the case of non-deterministic behavior of replicated components. Slember *et al.* [128] present a way of compensating effects of non-determinism on state using per-request adjustment to the state.

Passive [23] replication (a.k.a. primary-back replication) is widely considered as a

silver bullet to address the problems in active replication arising from non-determinism. However, prior work [44, 69, 133] has shown that passive replication is not a *cure-all* for non-determinism particularly in the case of replicated multi-tier environment. Failures in the middle-tier are often responsible for the side effects, which manifest themselves in the forms: (1) multiple invocations of partially completed executions, (2) *orphan request* [69, 133] and orphan state in the system, and (3) loss of global state consistency due to re-executions of non-idempotent operations.

As operational strings are essentially stateful multi-tier systems, they are also subject to the side-effects of replication due to various sources [108] of non-determinism (*e.g.,* load balancing, sensors, clocks, timeouts, thread scheduling and preemption). The state of the components of DRE systems often evolves in non-deterministic fashion as the mission progresses. The state of an operational string, which is distributed across multiple components, must be synchronized with the replica operational string without violating the global replica state consistency in fault-free and faulty situations.

Existing solutions [36, 44, 46, 69, 133] that address this problem rely on transactions as supported by Object Transaction Service (OTS) [99] to ensure state consistency. Transaction service, however, poses an overhead which adversely affects the end-to-end deadline. Moreover, fault-recovery becomes more complex due to *rollback* operations across multiple components to ensure system consistency. Ensuring schedulability of these rollback operations in case of failures further increases the tangling of timeliness and availability concerns.

I.4 Research Approach

To address the challenges identified in Section I.3, this dissertation describes: (1) a QoS modeling and modularization framework for fault-tolerant component-based DRE systems

to address the problem of crosscutting availability concerns, (2) a multi-stage model transformation process to support application-transparent failover of a group of stateless components, and (3) design and implementation of a group-failover protocol to rectify orphan components and maintain system timeliness and data consistency without the overhead of transactions. A brief summary of the different aspects of this dissertation is presented below.

1. **QoS modeling and modularization framework** uses model-driven engineering [119] (MDE) methodology to provide higher level declarative abstractions for capturing various QoS concerns such as granularity of replication, degree of replication, and network bandwidth reservations. The QoS framework is implemented as a domain-specific modeling language named Component QoS Modeling Language (CQML). CQML modularizes the QoS concerns away from the system's composition concerns. To weave the concerns in system artifacts, aspect-oriented transformations have been developed. Chapter II describes the QoS modeling framework in detail.

2. **Application-transparent group failover of stateless components** has been demonstrated using the component QoS modeling capabilities of CQML and generation of aspect-oriented code for fault-masking. The generative tools are implemented in Generative Aspects for Fault Tolerance (GRAFT), which is a multi-stage process for model-to-model, model-to-text, model-to-code transformations. Availability requirements of operational strings modeled using CQML are automatically transformed by GRAFT into component-specific fault-detection and recovery advice, which are later weaved into component stubs using AspectC++ [131]. Chapter III describes the approach in detail.

3. **A protocol for end-to-end reliability of stateful non-deterministic components** has been discussed, which addresses the state consistency issues of a stateful non-deterministic components that participate in replicated nested invocations. The design of the protocol ensures state consistency in fault-free and faulty scenarios without the overhead of transactions. It is implemented in the context of Component Integrated ACE ORB [64] (CIAO) middleware. The middleware transparently manages fault-masking, fault-correlation, globally consistent state synchronization, while rectifying orphan components. Chapter IV describes the group-failover protocol in detail.

I.5 Resolving Solution Domain Challenges: Object Structure Traversal

MDE has been used extensively in this dissertation to address the systemic issues of distributed real-time and embedded systems. MDE allows developers to express the system requirements at higher level of abstraction using domain-specific models [53]. These models are often represented in memory using heterogeneously typed hierarchical object structures in the form of either a tree (*e.g.,* XML document) or a graph. The necessary type information that governs such object structures is encoded in a schema. For example, metamodels [137] serve as schema for domain-specific models.

The programs (*e.g.,* model interpreters and transformations) that accept the domain-specific models as input need to perform several operations, such as traversal, iteration, selection, accumulation, sorting, and transformation on the objects in the model. Existing techniques [39] to write these programs often use language-specific data binding [70, 83] tools to generate object-oriented API for traversal and manipulation of the models. Unfortunately, such object structure traversals are often verbose due to schema-specificity of the API. Intuitive and succinct traversal notations, such as XPath [154] can not be used without sacrificing the type-safety of the generated object-oriented API. Traversal programming

idioms (*e.g.,* XPath child/parent axes and wildcards) are not natively supported in general-purpose programming languages. Moreover, the traversal logic and type-specific computations are tightly coupled due to the lack of generic, reusable mechanisms to express traversal over in memory object structures represented in third generation programming languages.

To address these limitations, this dissertation presents a multi-paradigm programming [27, 150] approach to develop a domain-specific language (DSL) for specifying traversals over object graphs governed by a schema. An expression-based [29] pure embedded (C++) DSL called Language for Embedded quEry and traverSAl (LEESA) is presented in Chapter V. LEESA demonstrates how generic programming, static metaprogramming [3], generative programming [28], and strategic programming [75, 147] in combination with C++ operator overloading can coexist in a single framework to resolve the gap between XPath-like notation and the type-safety of object-oriented programing. LEESA addresses the challenges in MDE identified during the course of the research work on model-driven fault-tolerance provisioning for distributed real-time and embedded systems.

I.6 Dissertation Organization

The remainder of this dissertation is organized as follows: each chapter describes a single focus area, describes the related research, the unresolved challenges, our research approach to solve these challenges, and evaluation criteria for this research. Chapter II describes aspect-oriented modeling (AOM) techniques for modularizing QoS requirements of DRE systems. Chapter III presents a multi-stage model transformation approach to automatically synthesize models, code, and configuration to weave the availability requirements into system artifacts. Chapter IV presents the group-failover protocol to rectify the undesirable side effects of replication in the context of multi-tier, stateful, non-deterministic components. Chapter V describes how the challenges encountered in the solution domain of MDE are addressed using a novel domain-specific embedded language that simplifies

object structure traversal commonly needed in model-driven tool chains. Finally, Chapter VI presents the concluding remarks.

CHAPTER II

ASPECT-ORIENTED MODELING FOR MODULARIZING QOS CONCERNS

This chapter addresses the first challenge outlined in Chapter I – capturing crosscutting QoS concerns of enterprise DRE systems. First, an overview of the existing research in the field of QoS modeling for component-based distributed systems is presented. Second, unresolved challenges in the existing research are identified. Finally, a solution approach that captures QoS requirements of enterprise DRE systems using domain-specific modeling is discussed.

II.1 Related Research

The related research has been categorized across the following dimensions: (1) QoS modeling using the Unified Modeling Language (UML) [105] and (2) QoS modeling using domain-specific modeling (DSM) [53, 119].

1. **QoS modeling using UML:** Lightweight and heavyweight extensions for UML are possible to create QoS profiles using extensibility mechanisms provided by UML. Lightweight extensions use only the mechanism of stereotypes, tagged values, and constraints. Heavyweight extensions require modification to the UML metamodel, which is naturally more intrusive than lightweight approaches. The OMG has adopted the UML profile for schedulability, performance, and time (SPT) [97] specification, which is based on lightweight extensibility mechanisms of UML. OMG has also adopted a more general profile for modeling QoS&FT [102]. This UML profile provides a way to specify the QoS ontology with QoS characteristics. It has support for annotating UML activity diagrams with QoS requirements.

Espinoza *et al.* [43] compare the SPT and QoS&FT profiles and proposes to combine the simplicity of the SPT profile with the generality of the QoS&FT profile for the "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems" (MARTE) [104] profile. The MARTE profile extends UML using its lightweight mechanisms with concepts for modeling and quantitative analysis of real-time and embedded systems (specifically, for schedulability and performance analysis). MARTE provides a general analysis framework called the General Quantitative Analysis Model (GQAM), which can be specialized for quantitative schedulability and performance analysis. A through literature survey of UML profiles that facilitate quantitative analyses using formalisms such stochastic Petri nets, Markov chains, and timed automata are presented in [21].

Another prior effort called Component Quality Modeling Language [2], developed by Aagadel *et al.* is a platform-independent, general-purpose language for defining QoS properties. It allows both interface annotation as well as component type annotation. Moreover, it has support for UML integration based on a lightweight QoS profile and has QoS negotiation capabilities. Previous work on QoS specification languages including QML [47] (QoS Modeling Language) and QuO [158] (Quality Objects) is superseded by [2]. This language allows QoS annotations at the type level (IDL interface and component definition) only and therefore, cannot be used to specify QoS requirements on components on a per-instance basis.

Michotte *et al.* [85] present an aspect-oriented approach for modeling *recovery blocks* generalized to component architectures represented using UML. The functional view is separated from the fault-tolerance aspect view, which can be composed later using *composition directives*. In this approach, however, every client component that needs fault masking requires manual instantiation of the *componentized recovery block pattern*. Moreover, it does not appear to support important fault-tolerance concerns such as state synchronization, deployment, and grouping of components.

With the advent of Service-oriented Architecture [42] (SOA), many researchers have developed UML profiles for modeling SOA non-functional concerns. A thorough survey is presented by Wada *et al.* [149]. They propose a UML profile to graphically specify SOA non-functional aspects in an implementation independent manner and an MDE tool that accepts UML models defined with the proposed profile and transforms them into application code and deployment descriptors.

The research on software systems reliability [58] using model driven architecture (MDA) [94] focuses on a platform-independent means to support reliability design following the principles of MDA. The research aims to systematically address dependability concerns from the early to the late stages of software development by expressing dependability architectures using profiles. Design profiles are mapped to deployment domains, where the reliability configurations of how the components communicate and are distributed is explained.

UML has also been used to perform model-driven dependability analysis [157] for composite web services. The UML representation is based on Business Process Execution Language (BPEL), and extensions are added to characterize the fault behavior of the elements comprising the web services. Model transformations are used to map the UML models to Block Diagrams, Fault Trees and Markov models to analyze the dependability characteristics of the composite web services.

2. **QoS modeling using domain-specific languages:** The SysWeaver [35] approach is an MDE-based technique for developing real-time systems. It supports design-time timing behavior verification of real-time systems and also supports automatic code generation and weaving for multiple target platforms. SysWeaver, however, does not address tangling of availability concerns into structural concerns. The replicas of protected components need to be explicitly modeled in the functional view of the Simulink model.

The Embedded Systems Modeling Language (ESML) [67] used for avionics mission computing provides modeling support for fault management and mitigation strategies that are not unlike passive replication. However, the fault model is restricted to processor failures only and the modeling language requires explicit modeling of replicated functionality, which is an example of cross-cutting fault-tolerance concerns. JReplica [63] proposes *Disguise Replication*, which is a Java extension to capture object-level replication aspects such as state, guards, execute-around actions to customize replication. It is not application transparent but allows programmer-defined extensions to recovery strategies. A UML extension is also proposed using stereotypes. However, it is restricted to the object-oriented paradigm.

Bajohr *et al.* [9, 10] present a model-driven approach for self-reconfiguration of highly available enterprise services. Behavior of services is encoded using service logic graphs – a domain-specific behavior modeling language with deterministic finite state automaton as its underlying model of computation. A graph-transformation-based technique is used to enhance the behavioral models of services with high-availability blocks for reading/writing checkpoints and managing failover. The approach is being used for enterprise systems where primary-backup replication of a single service (*e.g.,* email, news, www) is built using a cluster of nodes. Although the overall system is multi-tier, the business-tier does not have deeply nested invocations as found in the operational strings of enterprise DRE systems. Also, it is not clear whether the high-availability transformation is customizable via some sort of parameter mechanism.

Architecture Analysis and Design Language (AADL) [143] provides a standardized textual and graphical notation for describing software and hardware system architectures and their functional interfaces. The AADL Error Model Annex [144] has also been standardized to be used for describing dependability related characteristics in AADL models (faults, failure modes, repair policies, error propagations). The AADL

Error Model Annex mentions that stochastic automata such as fault trees and Markov chains can be generated from AADL specifications enriched with dependability-related information.

Modeling of reliable messaging in Service Oriented Architectures (SOA) is shown in [49] where messages can be annotated with specifications such as needsAck, filterDuplicates, and timeout values. A graph transformation is used to generate *envelopes* that wrap the original messages with necessary reconfigurations for reliable delivery mechanisms.

II.2 Unresolved Challenges

Despite a large body of existing research described in Section II.1, designing operational QoS-intensive DRE systems remains a significantly hard problem due to multiple crosscutting non-functional characteristics such as fault-tolerance, timeliness, authentication, authorization and network level QoS that must be decoupled from the system's functional composition concerns. Moreover, existing work lacks support for capturing QoS requirements of enterprise DRE systems that consider fault-tolerance, timeliness, and network-level QoS holistically. In the next section we highlight the prevailing gap.

II.2.1 Tight coupling of functional and QoS concerns

To assist in designing systems where non-functional concerns crosscut with structural concerns, DSM tools are promising but they must provide strong decoupling between the system's structural and non-functional concerns and must combine them when the final system is realized. Such decoupling should not only provide different views for different concerns (*view-per-concern*) but should also enable independent evolution of the modeling capabilities of each view. Evolution of the modeling capabilities of a concern view often requires enhancements to the metamodel of the view. Supporting independent evolution of metamodels of each concern view shortens the development lifecycle by allowing parallel

enhancements to the modeling capabilities (*i.e.*, the metamodel) and models pertaining to the view.

Platform-independent notion of QoS requirements is largely independent of the structural capabilities of the chosen implementation platform. Despite QoS being a platform-agnostic concept, DSM tools tend to tightly couple QoS with the structural characteristics. However, the variability in the structural capabilities of the contemporary component platforms need not prevent their corresponding DSM tools from having platform-independent modeling support for QoS. However, contemporary DSM tools are based on ad-hoc designs of metamodels for modeling QoS that couple them tightly with structural capabilities, preventing their reuse in other component platforms and limiting extensibility.

II.2.2 Lack of support for variable failover granularity

In enterprise DRE systems, traditional approaches to fault tolerance that rely on replication and recovery of a single server process or a single host are not sufficient since the fault management schemes must account for the timely and simultaneous failover of groups of entities while also improving the system availability by minimizing the risk of simultaneous failures of groups of replicated entities. Fault-tolerance requirements of operational strings may be specified at several different levels of granularity, such as per component, across a group of components, and across nested component groups.

II.2.3 Lack of support for mixed-mode replication strategies

Enterprise DRE systems require support for mixed-mode availability wherein parts of the system that can not tolerate the recovery time needed for passive replication, may require high-availability solutions based on active replication schemes. On the other hand, some parts of the system may demand passive form of replication to overcome issues

with non-determinism. Standardized middleware solutions to fault-tolerance, such as FT-CORBA [92], provide a *one-size-fits-all* approach, which do not support mixed-mode dependability semantics.

II.2.4 Lack of intuitive mechanisms for network-level QoS specification

Each operational string in an enterprise DRE system can specify a required level of network QoS (*e.g.*, high priority vs. low priority), the source and destination IP and port addresses, and bandwidth and delay requirements. This information is used to allocate and configure network resources to provide the required QoS. These network QoS requirements can change depending on the deployment context. Conventional techniques, such as hard-coded API approaches [33] are not application transparent. Writing this code manually to specify network QoS requirements is tedious, error-prone, and non-scalable.

II.3 Solution Approach: Component QoS Modeling Language (CQML)

II.3.1 Overview of CQML

This section describes a novel solution to address the limitations of DSM design tools for CBSE. Component QoS Modeling Language (CQML) is a reusable, aspect-oriented modeling (AOM) [54] framework developed using the Generic Modeling Environment (GME) [78]. CQML is designed to be superimposed on a wide range of structural composition modeling languages as long as they conform to a small set of invariant structural properties defined by CQML. Based on these invariant properties, CQML defines an abstract join point [68] model for associating QoS aspects to the structural elements. The join point model defines where the QoS aspects meet structural elements.

Around its abstract join point model, CQML has an extensible QoS modeling framework that allows declarative QoS requirements to be associated with structural component models. The QoS requirements are modularized using what we call *declarative QoS aspects.* They bind the QoS advice to the join points of the underlying structural modeling

language. This chapter demonstrates how abstract syntax of purely structural modeling languages can be *retroactively* enhanced with the QoS modeling capabilities of CQML by superimposing CQML's join point model. CQML's capabilities are evaluated using three different structural modeling languages for component-based systems.

II.3.2 Identifying Invariant Properties of Component-based Structural Modeling Languages

Our focus is on general component-based systems, which are composed using multiple components orchestrated to form application workflows. Contemporary component models often have first-class support for primitives, such as components, connectors, and methods. The structural artifacts of a component-based system can be realized using these primitives in a language specifically designed for modeling system structure.

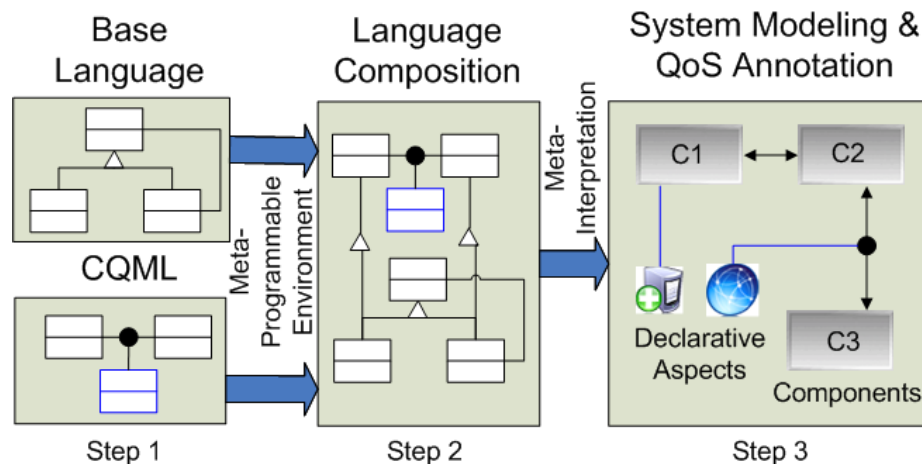

Figure 2: Process Model for Reusing CQML for QoS Modularization and Weaving

Since CQML is aimed specifically at modularizing non-functional concerns of component-based systems in a platform-independent manner, CQML requires an underlying base composition modeling language that allows construction and manipulation of platform-specific structural models. Many platform-specific as well as platform-independent component

structural modeling languages, such as Embedded Systems Modeling Language(ESML) [67] for embedded systems, J2EEML [152] for Enterprise Java Beans, and Platform-Independent Component Modeling Language (PICML) [14] for Light-weight CORBA Component Model (LwCCM) [98] exist today with the capability to capture various composition semantics. In this chapter we have focused on languages developed using GME because CQML is also developed using GME. However, the concepts behind CQML can be applied in other tool environments.

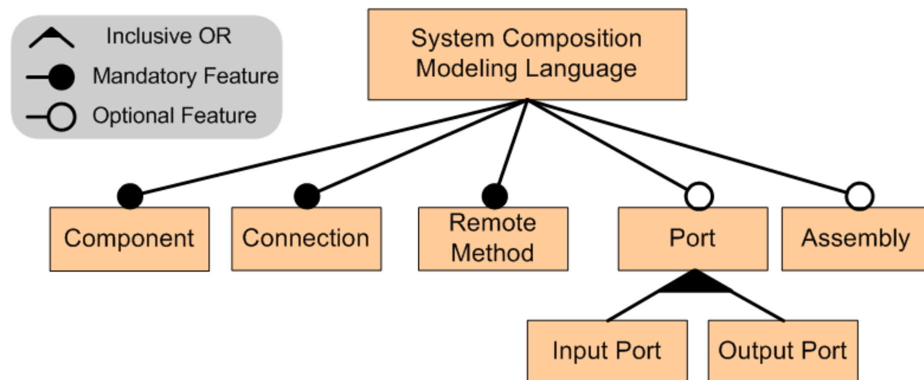

Figure 3: A Feature Model of Composition Modeling Language

We refer to such a structural modeling language as a *system composition modeling language* (or *base language* in short.) We formalize the features of a *base language* in a feature model [28] shown in Figure 3. CQML is designed taking into account the mandatory and optional features present in such languages. The base language should have first-class modeling support for components, connectors, and remotely invocable methods at the minimum. ESML, J2EEML, and PICML support all the mandatory entities mentioned in Figure 3 and therefore these languages can play the role of a base language for CQML as shown in Figure 2 (Step 1). In step 2, metamodel composition [16] techniques are used to *mix-in* the metamodel of CQML with that of the base composition modeling language producing a composite language, which has the capabilities of both constituent languages.

In step 3, the composite language is used to model component-based systems with QoS aspect modeling capabilities of CQML.

II.3.3 Extensible Design of CQML

Based on the feature model of component-based modeling languages, CQML builds an extensible QoS modeling layer. CQML associates declarative QoS aspects to one or more of the invariant properties of the underlying base language. We have designed several declarative QoS aspects that are applicable to a general class of component-based systems. We have developed (1) *FailOverUnit* [141], which modularizes fault-tolerance requirements of components and assemblies, (2) *SecurityQoS*, which modularizes role-based access control policies of port-based communication between components, and (3) *NetworkQoS* [11], which modularizes network-level QoS requirements while invoking remote methods. Some examples of the above concrete QoS characteristics are shown in Figure 4. A *FailOverUnit* is used to annotate component A as a fault-tolerant component. For connections between components B and C, network level QoS attributes (*e.g.,* priority of communication traffic) are associated using a *NetworkQoS* modeling element.

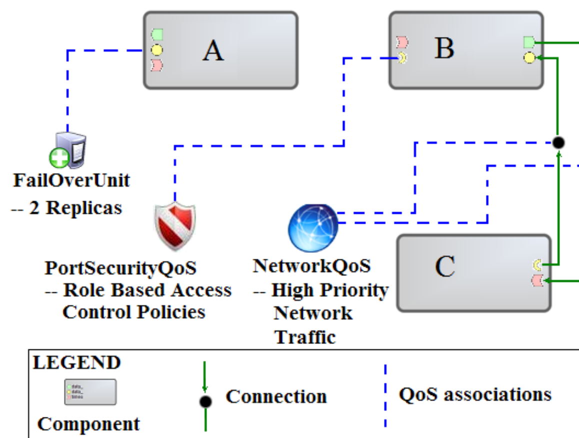

Figure 4: Declarative QoS Aspect Modeling Capability of CQML

To support extensions of a QoS metamodel, CQML defines a set of abstract QoS elements: *Component-QoS*, *Connection-QoS*, *Port-QoS*, *Assembly-QoS* and *Method-QoS*. CQML can be extended with new concrete declarative QoS modeling capabilities by inheriting from these basic abstract QoS elements. To enhance CQML with a concrete QoS aspect, a language designer has to extend the metamodel of CQML at the well-defined points of extension represented by the five abstract QoS elements. By doing so, the concrete modeling aspects inherit the (1) abstract syntax, (2) associations, (3) cardinality constraints, and (4) visualization constraints of the abstract QoS entities. For example, as shown in Figure 5, *FailOverUnit* inherits association constraints from the abstract *ComponentQoS* and *AssemblyQoS*. Therefore, *FailOverUnit* can be associated with components and assemblies only and never with ports or connections.

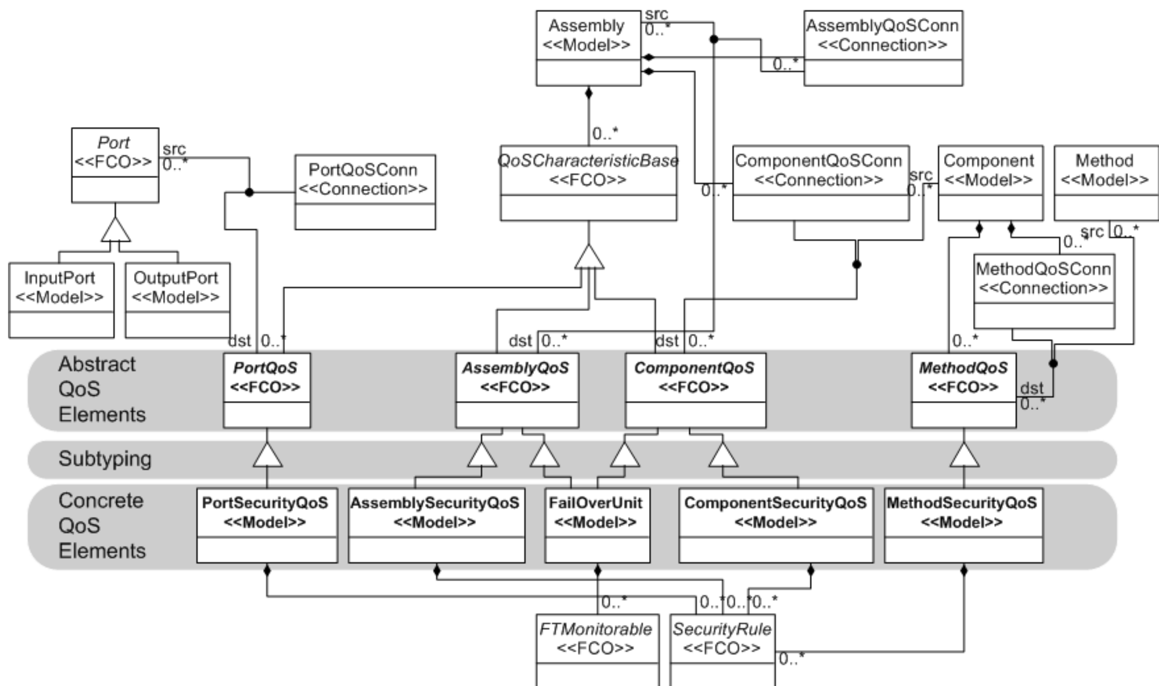

Figure 5: Metamodel of CQML.

Separation of structural concerns and QoS concerns is achieved using separate views for QoS and structural elements (*view-per-concern*). The visibility of concrete QoS modeling

elements is controlled using visualization constraints defined on abstract QoS elements. CQML defines visibility constraints on them such that they project QoS concerns in the *QoS* view of the GME model editor, which is different from the view where structural concerns are edited and manipulated. These constraints are inherited by all the concrete QoS elements that are derived from one or more abstract QoS elements as shown in Figure 5. Due to inheritance of these constraints, the concrete QoS elements are also projected and manipulated in the QoS view. Thus, CQML metamodel not only provides QoS modeling capability, it does so while achieving separation of concerns at the modeling level.

II.3.4 An Abstract Join Point Model for Component Modeling Languages

Along with the abstract QoS elements in the previous section, CQML defines an abstract representation of the mandatory and optional features of a generic structural modeling language. For example, CQML defines *AbstractComponent*, *AbstractConnection*, *AbstractMethod*, *AbstractPort*, and *AbstractAssembly*. These abstract types do not have semantics of their own except being able to associate QoS aspects with them. Moreover, the abstract nature stems from the fact that they cannot exist without a concrete instantiation in the underlying base modeling language. In the following section we describe how a concrete instantiation is done using a technique called metamodel composition.

II.3.5 Instantiating Abstract Join Point Model Using A Concrete Structural Modeling Language

CQML's support for QoS aspect modeling can be superimposed on a structural modeling language by composing the metamodel of CQML with the metamodel of the base language to create a composite language as described by Step 2 in Figure 2. Domain abstractions in the base language (*e.g.,* component, assembly, port) inherit from the corresponding abstract elements in CQML. Due to such inheritance, the domain abstractions in the base language inherit all the QoS related associations and constraints from CQML

elements. Models of the new composite language can use the associations defined in the original language and also the associations inherited from CQML. Thus, all the concrete QoS aspects and their constraints are *mixed-in* with the underlying structural modeling language.

Note that the abstract join point model achieves a strong decoupling between structural and CQML metamodels. The structural metamodel of the base language can be enhanced without affecting the CQML metamodel and vice-versa. Therefore, the abstract join point model is the key to support independent evolution of the structural as well as CQML metamodel. Moreover, using the abstract join point model, multiple composite languages can be created by composing CQML with different structural modeling languages using the same process. Figure 6 shows an example of how CQML is composed with PICML to create a composite language using inheritance. After composition, PICML's *component* and *assembly* can be associated with everything that CQML's *abstract component* and *abstract assembly* can be associated with (*e.g., FailOverUnit*).

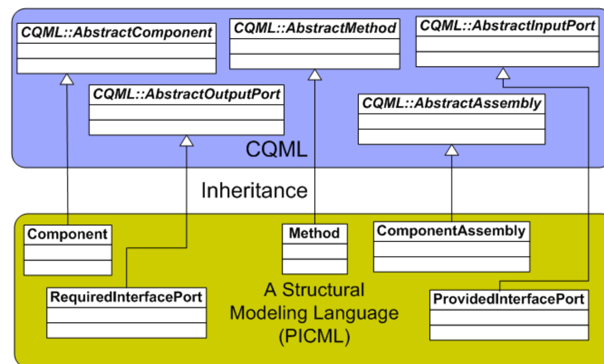

Figure 6: Composing CQML's Abstract Component Model with a Base Language Using Inheritance

An important benefit of our approach is that CQML introduces QoS modeling capability in a base language without affecting its original syntax and semantics. CQML can be composed flexibly with the underlying base language even though it does not support some optional features shown in Figure 3. Using CQML with a base language that supports less

number of primitives gives rise to a smaller concrete QoS model. On the other hand, composing CQML with a base language with all the mandatory as well as optional primitives gives rise to a larger QoS model.

Later in this chapter we show how CQML is composed with three different base languages (PICML, J2EEML, and ESML) that have different structural modeling capabilities. Composing CQML with them gives rise to different QoS modeling capabilities in each composite language: PICML′, J2EEML′, and ESML′. Reuse promoted by CQML's generic QoS entities and its design thus lends itself to easier development of component-based systems modeling languages with QoS support. It reduces the need of reinventing previously designed artifacts for every new QoS aspect that is added.

II.4 Modeling Fault-tolerance Requirements using CQML

In this section, we describe the design considerations and the CQML-based solution for a transparent fault-tolerance provisioning tool for enterprise DRE systems.

II.4.1 Design Considerations

- **Variable granularity of system protection.** Enterprise DRE systems are composed of several independently deployable assemblies of components that communicate together in a workflow fashion to carry out the system's functionality. Quite often the unit of modularity in the system design is larger than a single deployed component and results in some critical functionality of the system being spread across multiple components and/or assemblies. In terms of the availability perspective, the entire critical functionality which is spread across multiple components must be protected from failures. Moreover, failure of any one component in the workflow implies the failure of the entire flow. In such a situation, the system must failover to a redundant workflow as opposed to a single component. One strategy for the failover mechanism could be to allow graceful degradation. The functionality of the replica components

may not be the exact duplicate of the original. For example, the replica component can possibly implement an algorithm that is less resource hungry compared to the primary.

A design-time tool must allow the specification of these requirements of enterprise DRE system. Section II.4.2 describes how MDDPro provides intuitive abstractions to capture these fault-tolerance requirements of enterprise DRE systems.

- **Mixed-mode dependability requirements.** Enterprise DRE systems are large-scale and comprise several different components, each of which accomplish specific tasks of the entire system functionality. Some parts of the system may require fast failure recovery mandating active replication schemes. However, due to the overhead associated with active replication and the non-determinism issues [45, 90], it may be necessary to restrict the use of active replication to a small part of the enterprise DRE systems. Other parts of the system may then use other forms of replication, such as passive replication, or depend on simple restart mechanisms depending on the criticality of the component and available resources in the system.

  The design-time tool must enable enterprise DRE system developers to capture these mixed-mode dependability semantics of the system. When combined with the granularity of protection units and other performance requirements of the system, this provisioning task becomes complex to perform manually using ad hoc and programmatic techniques. Section II.4.2 describes how MDDPro provides intuitive abstractions to capture mixed-mode dependability requirements of enterprise DRE systems.

- **Effective replica deployment for maximizing availability.** Redundancy in the system improves system availability, however, high levels of reliability are realized only when replicas are placed in such a way that the risk of simultaneous failures of replicas is minimized. Effective replica placement also impacts several other performance characteristics of the entire system. For example, effective replica placement may be necessary to maintain a bounded and fast state synchronization among the replicas.

A design-time tool can be used to ensure that the system simultaneously satisfies multiple QoS requirements such as performance, predictability and availability, by incorporating deployment state space search algorithms that automatically find effective deployments. This problem is known as the constraint satisfaction problem. Optimality is a harder problem than constraint satisfaction, however, we do not consider it yet in our design. Chapter III describes how we have designed our tool that can plug in different replica placement algorithms that find effective deployments for enterprise DRE systems.

- **Transparent provisioning of fault-tolerance.** Even though the modeling techniques can help capture dependability requirements while replica placement algorithms can provide effective deployment decisions, these must ultimately be realized in the context of the underlying hosting platforms, such as the component middleware. Component middleware often use XML metadata that describes how components of an enterprise DRE system should be hosted in the middleware and how they must be connected to each other. For large-scale systems, the amount of metadata becomes very large and ad hoc techniques, such as handcrafting these descriptors, becomes infeasible and error prone.

  Dependability provisioning makes this task harder because the metadata must account for the protection units and provisioning the multiple replication schemes within the enterprise DRE system. This requires a substantial degree of middleware configuration by allocating different resources end-to-end. Replication adds to the number of connections that must be established between the different protection units and their replicas. The replication style makes this task even harder. For example, when active replication is used, the middleware must be configured to use a group communication substrate that is used by the communication between replicas. On the other hand, in passive replication, the secondary replicas must be provisioned on the middleware to accept periodic state updates from the primary. Chapter III

describes how generative programming [28] techniques used within our tool automates the metadata generation to provision dependability for enterprise DRE systems within the middleware platforms.

II.4.2 Modeling Notation

CQML allows an enterprise DRE system deployer to model the fault-tolerance requirements in the QoS view of the DRE system as shown in Figure 7. The QoS view leverages the basic structure of the DRE system in terms of the component instances in an assembly, component ports and their interconnections. CQML allows FT elements to be modeled orthogonally to the system components and therefore achieves separation of dependability concerns from the primary system composition and functionality concerns. The following fault-tolerance modeling elements are supported in CQML:

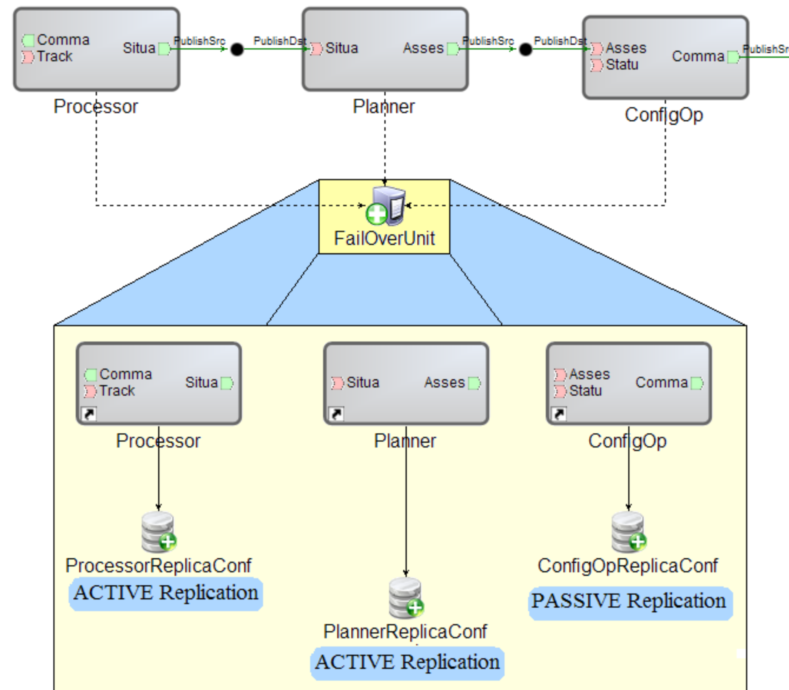

Figure 7: Availability Requirements Modeling in CQML

- **Failover units (FOUs),** which enable control over the granularity of protected system components, such as software components, component assemblies, or entire component workflows. Failure of any one element belonging to a FOU is treated equivalent to the failure of all the elements in the FOU and the system effectively fails over to another replica of the FOU. This modeling abstraction not only captures the failover granularities of system entities, but also the degree of replication for each FOU and other systemic requirements, such as the period of liveness monitoring for FOUs. The degree of replication is represented as a pair of numbers representing minimum and maximum number of replicas. The programming language artifacts that implement the replica components could be different from that of the primary components allowing graceful degradation of the functionality if the dependability solution requires it.

  Frequently, the liveness of distributed components is monitored using a *heart beat* protocol. The frequency of the heartbeat is one configurable parameter in the liveness monitoring, which can be configured in CQML. The heartbeat itself is configurable in two ways: *push model* or *pull model*. Thus, the directionality of the heartbeat can also be configured in CQML. In Chapter III we show how modeling of FOUs enable us to automatically synthesize and configure liveness monitoring components as well as heartbeat producing components. Conceptually, a FOU is an abstraction to capture the availability requirements at the control plane of the dependability solution.

- **Replication groups (RGs),** which allows capturing the replication requirements of software components within a FOU. These models specify replication strategies, such as active, passive or other variants, and the state synchronization policies for components. A replication group captures the configuration parameters related to the data plane of the deployment solution. Multiple replicas of the system components synchronize their state with each other as per the configuration of the data plane. For example, data synchronization frequency of the replicas is configurable.

Moreover, the topology of state synchronization among replicas is also a data plane level configuration issue handled in MDDPro.

- **Shared Risk Groups (SRGs),** provide a way of grouping resources in the target network of the applications that share a risk of simultaneous failure. Application components share a risk of simultaneous failure by virtue of the failure of the resources they share, such as processes, nodes, racks or even data centers on which they are hosted. Risk factors are determined by assigning the metrics, such as co-failure probabilities to a hierarchy of the network resources in a risk group that affects the availability of the system. The computation of the co-failure probabilities is assumed to be done a priori using reliability engineering methodologies.

  The primary purpose behind modeling the shared risk groups and their respective co-failure probabilities is to facilitate automated deployment decisions of the components in the system such that the probability of failure of the entire system is minimized thereby increasing the availability. One way of reducing the co-failure probability is to increase the physical distance between the nodes where the components are deployed. Here, the physical distance can be thought of as the distance from a remote host, a remote blade or a remote data center. An advantage of using the distance metric is that it is simpler and quite intuitive compared to the co-failure probability. In Chapter III we show how the shared risk group model is used by the CQML model interpreter to determine a suitable and effective deployment that satisfies the availability requirements and minimizes risks of simultaneous failures. In our prototype implementation of the algorithm we use the simpler distance metric to guide the decision of the replica placement.

  Figure 8 shows a model of the Shared Risk Group hierarchy. Hosts 1 to 5 are part of a domain and are contained under a common "RootRiskGroup" at the top. A RootRiskGroup represents comparatively larger structures such as a ship or an entire building. All the hosts in the domain share a common risk of failure of the

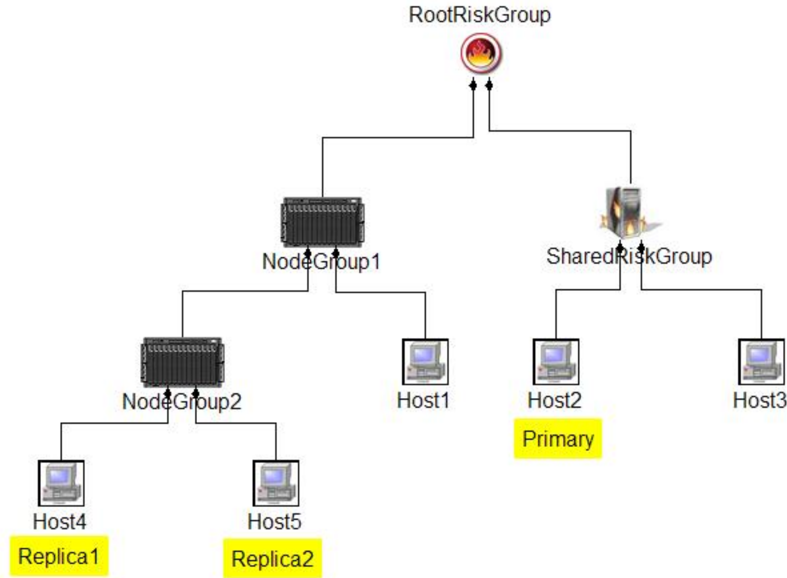

Figure 8: Shared Risk Group Hierarchy Modeling in CQML

largest composing structure represented by a RootRiskGroup. We limit the scope of our dependability solution at that level. The RootRiskGroup is further divided into smaller units of Shared Risk Groups as shown in the figure. For example, Host1, Host4 and Host5 share a common risk of a failure of the NodeGroup1, but failure of NodeGroup2 that consists of Host4 and Host 5 does not affect Host1.

The distance between hosts is simply computed as the number of tree edges between two hosts. For example, the distance between Host2 and Host3 is 2. Similarly, the distance between Host2 and the Host4 or Host5 is 5. Based on such a Shared Risk Group hierarchy, deployment decisions are taken to maximize the distance between the primary component and its replicas, as shown in Figure 8.

II.5 Modeling Network-level QoS Requirements using CQML

CQML provides a QoS modeling annotation called the *Network QoS* (NetQoS). DRE system developers can use NetQoS to declaratively specify the following deployment context-specific network QoS requirements on the modeled application flows: (a) network QoS

classes, such as HIGH PRIORITY (HP), HIGH RELIABILITY (HR), MULTIMEDIA (MM), and BEST EFFORT (BE), (b) bi-directional bandwidth and delay requirements, and (c) selection of transport protocol.

NetQoS's network QoS classes correspond to the DiffServ levels of service provided by our Bandwidth Broker [30].[1] For example, the HP class represents the highest importance and lowest latency traffic. The HR class represents traffic with low drop rate (*e.g.*, surveillance data). NetQoS also supports the MM class for sending multimedia data and the BE class for sending traffic with no QoS requirements.

NetQoS supports two models for controlling the Diffserv markings: (1) the CLIENT_PROPAGATED network priority model that allows the clients to dictate the bi-directional priorities, and (2) the SERVER_DECLARED network priority model that allows servers to dictate the bi-directional priorities. Figure 9 shows a NetQoS model that highlights many of its key capabilities.

Multiple instances of the same reusable application components can be annotated with different QoS attributes using an intuitive drag and drop technique. This method of specifying QoS requirements is thus much simpler than modifying application code for each deployment context, as demonstrated later in this chapter. Moreover, the same QoS attribute (*e.g.*, HR_1000 in Figure 9) can be reused across multiple connections. NetQoS thus increases the scalability of expressing requirements for large numbers of connections that are prevalent in enterprise DRE systems.

II.6 Evaluating Composability of CQML

This section describes our evaluation of CQML. We demonstrate how purely structural modeling languages can be enhanced with QoS annotation capabilities by composing them with CQML. We show this capability with three different component-based structural modeling languages.

[1]NetQoS's capabilities can be extended to provide requirements specification conforming to a different network QoS mechanism, such as IntServ.

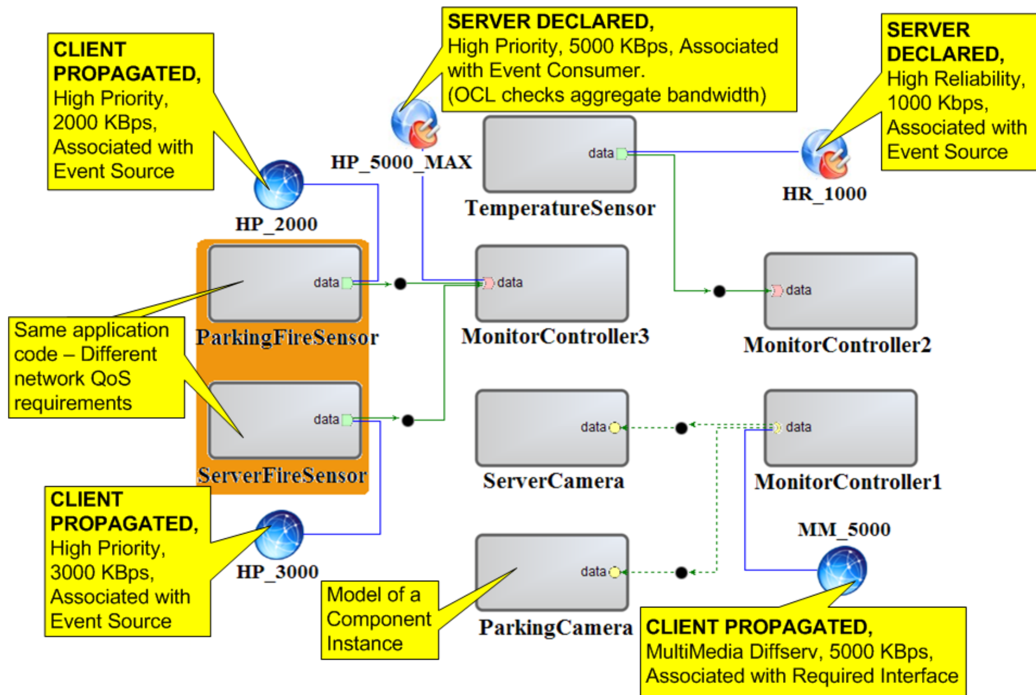

Figure 9: Network-level QoS Modeling Capabilities of CQML

To evaluate composability of CQML we chose three component-based structural composition languages: the Platform Independent Component Modeling Language (PICML) [14] for Light-weight CORBA Component Model (LW-CCM) [98], J2EEML [152] for Enterprise Java Beans (EJB), and the Embedded Systems Modeling Language (ESML) [67] for embedded systems.

There are many commonalities and differences among these languages that stem from the differences in the underlying component model that they model. Table 1 summarizes the similarities and the differences between these three languages. All of them are component-based system modeling languages, which treat components as first class entities and have varying degree of support for assemblies (nesting of components and assemblies.) For example, J2EEML and PICML support hierarchical composition of assemblies but ESML has a flat, single level structure of components. All the three languages support the notion of a connection. The notion of provided interfaces (an implementation of a particular interface) is present in PICML and ESML but not quite explicit in J2EEML. It

manifests itself in a weaker form of just a set of invocable methods on a bean. Similarly, the notion of required interfaces[2] is present in PICML and ESML but is absent in EJB and hence in J2EEML. In summary, the PICML′ feature set turns out to be a super-set of the features of the other two languages.

Supported Features	PICML	J2EEML	ESML
Component, Methods, and Connections	Yes	Yes	Yes
Provided Interface Ports	Yes	No	Yes
Required Interface Ports	Yes	No	Yes
Assemblies	Yes	Yes	No

Table 1: Comparison of Capabilities of Selected Three Modeling Languages

Using specializations to the join point model, we composed CQML with the above three languages giving rise to three composite languages: PICML′, J2EEML′, and ESML′. The concrete join point model of the three composite languages varies because of the varying structural capabilities of the underlying base languages. The richness of the join point model determines the ability of the composite language to attach declarative QoS aspect to the structural elements in a model.

Table 2 summarizes the enhanced QoS aspect modeling capabilities of the composite languages. All three composite languages had new support for modeling *FailOverUnit*s, which are associated with components. However, J2EEML′ could not support the QoS advice association with required interfaces like PICML′ and ESML′ because J2EEML has no support for ports built into it. Similarly, in ESML, *FailOverUnit* cannot be associated with assemblies because assemblies are not supported.

All the above QoS modeling enhancements are projected into and manipulated from the *QoS* view. This graphical view of the model is separate from the *structural* view where

[2]It describes an ability of a component to use an interface implementation supplied by some external component.

Structural Elements	PICML$'$	J2EEML$'$	ESML$'$
Component	FailOverUnit	FailOverUnit	FailOverUnit
Assembly	FailOverUnit	FailOverUnit	N.A.
Connections	NetworkQoS	NetworkQoS	NetworkQoS
Provided Interface Ports	SecurityQoS	N.A.	SecurityQoS
Required Interface Ports	SecurityQoS	N.A.	SecurityQoS

Table 2: Enhanced QoS Aspect Modeling Capabilities of Composite Languages PICML$'$, J2EEML$'$, and ESML$'$

hierarchical systems are composed using components. This feature provides visual separation of structural concerns from QoS concerns. Moreover, the metamodel of the structural view and CQML can be enhanced in parallel, if needed, and can be composed again as shown in Figure 2.

The results indicate that CQML can be composed with a variety of component-based structural composition languages to introduce QoS modeling support in them while supporting strong separation and independent evolution of QoS and structural concerns.

CHAPTER III

WEAVING DEPENDABILITY CONCERNS IN SYSTEM ARTIFACTS

The previous chapter proposed aspect-oriented domain-specific modeling techniques to modularize crosscutting fault-tolerance concerns. Even though the modeling techniques can help capture availability requirements, these must ultimately be realized in the context of the underlying hosting platforms, such as the component middleware. Realizing the requirements transparently requires instrumentation in several system components such as source code, connection and deployment metadata.

This chapter addresses the second challenge outlined in Section I – *weaving dependability concerns in system artifacts.* First, an overview of the existing research in the field of transparent fault-tolerance provisioning is presented. Second, a list of challenges that are still unresolved is presented. Finally, a solution approach that automatically synthesizes system artifacts from high-level requirements models is presented.

III.1 Related Research

Alexandersson *et al.* [6] recognizes the benefits of applying aspect-oriented programming (AOP) techniques to modularize the crosscutting fault-tolerance concerns and also identifies the limitations of existing AOP languages (*e.g.* AspectC++ [131]) to do the same. AspectC++ language is extended to support five fault tolerance mechanisms including recovery cache, time redundant execution, recovery blocks, run-time checks, and control-flow checking. The mechanisms proposed here could be used for *incremental checkpointing* to reduce state synchronization overhead.

Sevilla *et al.* [122] propose an aspect-oriented code generation approach for transparently providing fault-tolerance and load-balancing in CORBA-LC component model. Code is generated from annotations in higher level graphical models of system composition.

Their technique uses active replication but does not propose any way to deal with non-determinism. Also, they do not discuss how fault-monitoring, passive replication, state-synchronization infrastructure can be synthesized and deployed.

Polze *et. al.,* [110] propose a framework that uses design-time and configuration-time information for automatic distributed, replicated instantiation of components. The requirements are specified declaratively using a graphical textual interface. The proposed *aspect weaver* needs to combine fault-tolerance, timing, and consensus aspects at or before run-time. However, the details of AOP mechanisms that compose multiple, possibly overlapping, non-functional aspects is not discussed.

The CORRECT [20, 24] project describes a project that is looking at applying step-wise refinement and OMG's Model Driven Architecture [94] to automatically generate Java code used in a fault tolerant distributed system. The project uses UML to describe the software architecture in both a platform-independent and platform-specific form. Model-to-model transformations are used to incrementally enrich the models with platform-specific artifacts until the Java skeleton code is generated.

Automatic aspect generation is used in [136] to shift method call logging from FT-CORBA [103] middleware to application level to improve performance. Thread-level synchronization aspects are automatically weaved into the application code from a textual component description provided by the developer. Finer granularity of thread synchronization is shown to improve performance than method-call level synchronization of FT-CORBA.

JReplica [63] uses AOP to modularize the replication aspect of fault-tolerance. JReplica replication primitives extend the Java language so that modularized fault-tolerance aspects can be weaved around the classes implementing the business functionality. It ensures that only the required method invocation paths are intercepted as opposed to all. However this optimization is not possible while being completely application-transparent.

Afonso *et al.* [4] propose an AOP-based approach for modularizing fault tolerance code from threaded applications in distributed embedded systems. Their approach is used to

inject fault tolerance at the application thread level and considers several fault tolerant mechanisms (*e.g.,* recovery blocks, distributed recovery blocks, and N-version programming [8]). Although they provide "base" aspect with reusable pointcuts, concrete aspect implementation must be provided by the application developer.

Meta-object protocols (MOP) have been used [116, 139] to introduce fault-tolerance transparently in dependable systems. Taiani *et al.* [139] propose a MOP for communication of context information from middleware to the operating system using thread-local storage (TLS). They exploit the introspection and interception capabilities of the operating system to coordinate operations on mutex for ensuring determinism on actively replicated multi-threaded servers.

Rubel *et al.* [115] demonstrate the applicability of a group communication protocol (Spread [156]) for fault-tolerant hierarchical DRE systems developed using component middleware. Their choice of active replication (and hence group communication) is primarily due to rapid recovery requirements. While their approach appears to be application-transparent, it requires costly modifications to MEAD [90] as well as CIAO [64] middleware. Also, it is not clear how non-determinism issues of active replication are handled.

III.2 Unresolved Challenges

Despite a large body of existing research described in Section III.1, transparent provisioning of fault-tolerance for DRE operational strings remains a significantly hard problem due to the following reasons.

III.2.1 Lack of support for incremental model refinement for multi-QoS-aware modeling

Prior work on automatic provisioning of fault-tolerance from models consider fault-tolerance as the only dominant non-functional concern in the system. Moreover, they assume that the replica is structurally identical to the primary mainly because their lack of

support for different levels of granularity of failover. These modeling tools do not consider the possibility of *non-isomorphic replication* of operational strings, which is a valuable fault-tolerance provisioning technique based on the principle of *diversity*.

Replica operational strings are often not the exact clones of the primary. Potential ways they might be different are: (1) the replica operational string may have different implementation of components to allow graceful degradation in case of failures, (2) number of components might be fewer or more than the primary, (3) end-to-end deadline might be different and as a consequence, network-level QoS requirements could be different. Therefore, the existing modeling techniques fail to incorporate not only the structural diversity of fault-tolerant operational strings but also their *simultaneous* QoS requirements.

A desirable solution should not only automate fault-tolerance provisioning from the models, but also provide an opportunity to the system designers to *incrementally refine* other QoS concerns in the model. This requires a more *step-wise* approach as suggested in OMG's MDA [94], where the system is incrementally obtained by instantiating and refining a specification of system structure, behavior, and QoS requirements.

III.2.2 Lack of middleware support for domain-specific recovery semantics

General purpose middleware have limitations in how many diverse *domain-specific* semantics can they readily support *out-of-the-box*. Since different application domains may impose different variations in fault tolerance (or for that matter, other forms of quality of service) requirements, these semantics cannot be supported out-of-the-box in general-purpose middleware since they are developed with an aim to be broadly applicable to a wide range of domains. Developing a proprietary middleware solution for each application domain is not a viable solution due to the high development and maintenance costs. The modifications necessary to the middleware are seldom restricted to a small portion of the

middleware. Instead they tend to impact multiple different parts of the middleware. Naturally, a manual approach consumes significant development efforts and requires invasive and permanent changes to the middleware.

Realizing these capabilities at application level impacts all the lifecycle phases of the application. First, application developers must modify their interface descriptions specified in IDL files to specify new types of exceptions, which indicate domain-specific fault conditions. Naturally, with changes in the interfaces, application developers must reprogram their application to conform to the modified interfaces. Modifying application source code to support failure handling semantics is not scalable as multiple components need to be modified to react to failures and provision failure recovery behavior. Further, such an approach results in crosscutting of failure handling code with that of the normal behavior across several component implementation modules.

Resolving this tension requires answering two important questions. First, how can solutions to domain-specific fault tolerance requirements can be realized while leveraging low cost, general-purpose middleware without permanently modifying it? An approach based on aspect-oriented programming (AOP) [68] can be used to modularize the domain-specific semantics as *aspects*, which can then be woven into general-purpose middleware using aspect compilers. This creates *specialized* forms of general-purpose middleware that support the domain-imposed properties.

Many such solutions to specialize middleware exist [66, 86], however, these solutions are often handcrafted, which require a thorough understanding of the middleware design and implementation. The second question therefore is how can these specializations be automated to overcome the tedious, error-prone, and expensive manual approaches? *Generative programming* [28] offers a promising choice to address this question.

III.2.3 Lack of support for auto-generation of full spectrum of fault-tolerance infrastructure

Transparent fault-tolerance provisioning for component-based DRE systems requires more than just code synthesis for fault-masking done in the prior work. As described in Section II.4.1, the design considerations for transparent fault-tolerance provisioning must account for automation of a broad range of concerns.

First, an effective placement of the component replicas must be determined to increase the overall availability of the system. Resource-aware algorithms such as [13, 34, 50, 59, 132] automate the placement decisions based on techniques such as response time analysis and CPU schedulability. Modeling tools need a framework that can use different constraint-based algorithms to determine an effective replica placement. In this chapter we discuss a heuristics-based placement algorithm for operational strings as well as a framework that can incorporate multiple algorithms.

Second, automatic instrumentation of the components is needed to achieve *fault-masking*. Fault-masking hides system failures from the clients with minimal impact on the end-to-end QoS (*i.e.*, response time). They are also the point where client-specific recovery actions are performed (*e.g.,* redirect call to the backup replica). In the case operational strings, the failure of the operational string may not be immediately apparent to the client components that are not directly connected to the failing component. Such indirectly connected components need to failover to the replica functionality in a timely manner to begin re-execution of the failed invocation. As a result, enabling transparent failover of a group of components requires complex coordination between fault-masking and recovery modules unlike single component failover.

Third, to realize the dependability requirements transparently, the metadata used by the underlying component middleware must also be instrumented automatically. The metadata determines how the components are deployed and how their interconnections are setup for remote invocations. The decisions of the intelligent placement algorithms must be

codified in the deployment descriptors. Furthermore, non-replicated parts of the system need to establish redundant connections to the replicated functionality it may failover to upon failure. Codifying these decisions manually is often error-prone and time-consuming. Therefore an automated support is highly desirable.

Finally, an important responsibility of a fault-tolerant system is to monitor the running system for faults and when faults are detected they must be reported to higher level components so that appropriate recovery procedure can be initiated. Developers of fault-tolerant DRE systems must also reason about how the monitoring subsystem will be deployed and configured so that failure of business components can be detected and reported in a timely and reliable way. This additional responsibility of designing, deploying, and configuring monitoring subsystem delays developers' main task of developing business logic. Therefore, an automated support for generating, deploying, configuring liveness monitoring infrastructure is highly desirable.

III.2.4 Lack of support for deployment-time network QoS provisioning

DRE systems must allocate and configure network resources based on the QoS requirements specified on their application flows so that network QoS assurance can be provided at runtime. Advanced network QoS mechanisms such as DiffServ Bandwidth Broker [31] had been used in the past by application developers to request a network service level and allocate and manage network resources for their remote invocations. However, in component-based systems, allocation of network-level resources must be transparent to application components to ensure separation of concerns.

Two steps must be executed successfully two ensure availability of bandwidth at runtime. First, the bandwidth requirements must be communicated to the programmatic network QoS provisioning mechanisms, such as DiffServ bandwidth broker. The bandwidth broker in turn configures the edge routers to provide differentiated service to packets marked

with specific code-points. Second, the application flows that desire the differentiated service, must ensure the proper code-points in every packet sent over the network. Clearly, the transport protocol sockets at the the source host must be configured with proper code-points at the time of connection establishment.

To ensure transparent execution of the above two steps requires sophisticated middleware infrastructures for resource allocation, deployment, and configuration. Prior work on integrating network QoS mechanisms with middleware [41, 113, 117, 151] focused on providing middleware APIs to shield applications from directly interacting with complex network QoS mechanism APIs. Middleware frameworks converted the specified application QoS requirements into lower-level network QoS mechanism APIs and provided network QoS assurances. These techniques, however, are not application transparent.

III.3 Solution Approach: GeneRative Aspects for Fault Tolerance

III.3.1 Overview of GRAFT

GRAFT is an automated multi-stage model-driven process to specialize middleware for provisioning of domain-specific fault-tolerance requirements. Figure 10 shows the steps in GRAFT in the form of a flowchart. At every successive stage, structural models of the component-based DRE system are refined into more concrete models ultimately realizing all the artifacts needed to deploy a transparent fault-tolerant component-based DRE system. The challenges identified in Section III.2 are addressed using overlapping stages, which involve automated *model-to-model*, *model-to-code*, and *model-to-text* transformations. Network QoS and end-to-end deadline requirements can also be specified manually on the intermediate models generated using the automated transformations resulting into a semi-automated approach.

Stage 1 in GRAFT leverages existing structural models of applications modeled as component assemblies, and annotates them with domain-specific fault tolerance requirements using CQML language. The C-SAW [130] aspect-oriented model weaver is then

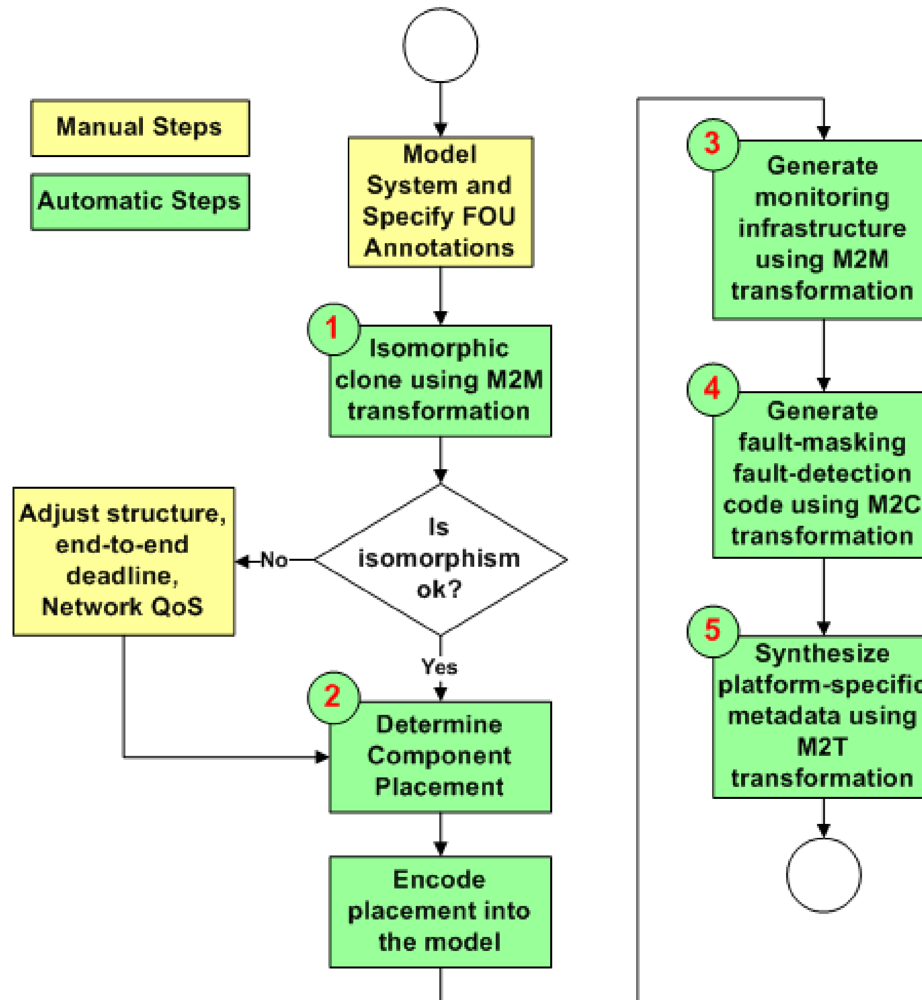

Figure 10: GRAFT's Multi-stage Process for Weaving Fault-tolerance Concerns in System Architecture Models

used to transform the annotated models into models comprising new structural elements corresponding to the replication degree requirements. This model-to-model transformation step is necessary because it allows system developers to further refine the model with potentially non-isomorphic replication of operational strings, network QoS, and end-to-end deadline requirements.

Stage 2 in GRAFT leverages the pluggable component placement framework to determine the placement of replicated components in the system. The outcome of the placement algorithm is reflected back in the model. Once again it is possible to override the automated decisions by manual decisions due to the availability of the intermediate models.

Stage 3 in GRAFT is another model-to-model transformation that generates the structural models of the monitoring infrastructure. This model transformation takes into account the placement of the primary as well as backup components and collocates the monitoring components. Generation of structural models is necessary to obtain platform-specific metadata (*e.g.,* XML deployment descriptors) for deployment and configuration engines from the output model of the transformation.

Stage 4 in GRAFT is a model-to-code transformation that generates client-specific fault-masking code that otherwise would have been written manually to carry out specialization of the middleware. The generated code is modularized using AspectC++ [131] language – an aspect-oriented extension to the traditional C++ language. Finally, GRAFT uses the AspectC++ [131] compiler to weave in the specialized code into the middleware.

Finally, stage 5 in GRAFT is a model-to-text transformation carried out by model interpreters that traverse the refined structural models and generate the platform-specific metadata. The metadata contains component deployment configuration, connection establishment configuration, and network QoS requirements for primary and backup components and the monitoring infrastructure.

Because GRAFT is a design-time process, it can not be used to deployment-time network QoS provisioning. Therefore a separate Network Resource Allocator Framework (NetRAF) has been developed to configure network resources based on the network QoS metadata generated by GRAFT for bandwidth assurance at runtime.

III.3.2 Stage 1: M2M transformation for multi-QoS-aware refinement of availability models

Although CQML modularizes recovery semantics using the FailOverUnit, CQML is at a higher level of abstraction than that of the existing modeling language tools, such as deployment and configuration and packaging tools that understand structural models only.

For this reason, high-level CQML models with fault tolerance requirements must be transformed into purely structural models, automatically, so that GRAFT can leverage existing generators for deployment and configuration metadata. Such a transformation requires several steps, including (1) duplicating models of the primary components participating in a FailOverUnit, and (2) duplicating their interconnections so that the necessary connections can be established at deployment time for the replica DPU.

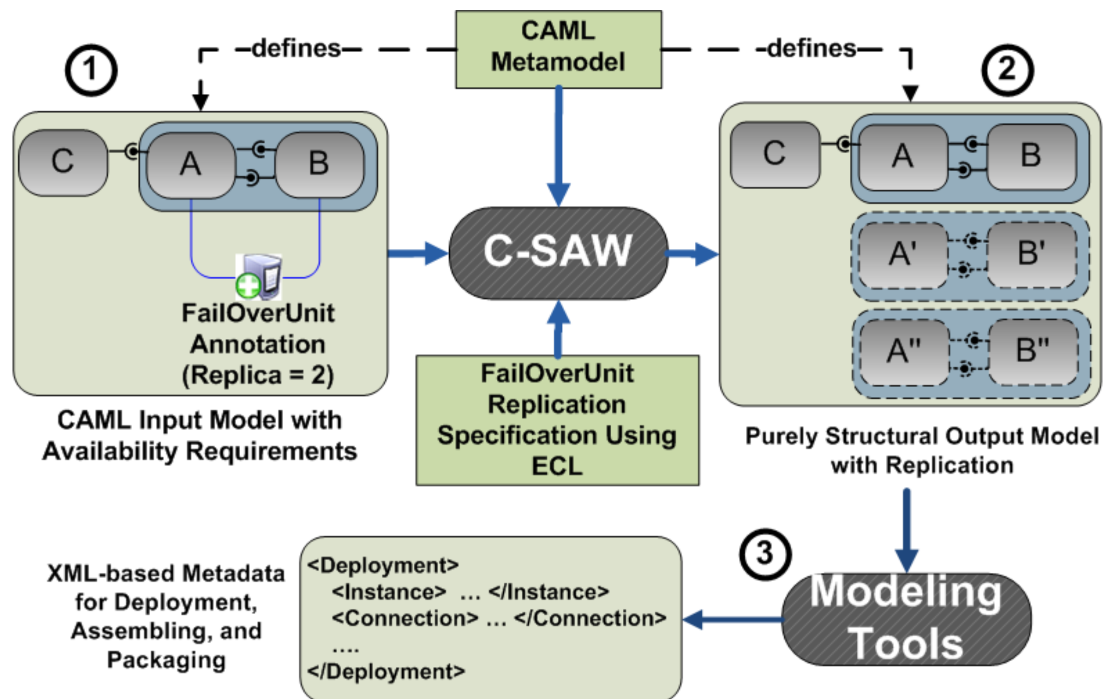

Figure 11: Automated Model Weaving Using C-SAW and FailOverUnit Replication Specification Using ECL

GRAFT uses aspect-oriented model weaving [52] support provided by Constraint-Specification Aspect Weaver (C-SAW) [130] tool. The C-SAW weaver is a generalized model-to-model transformation engine for manipulating domain-specific models. C-SAW uses Embedded Constraint Language (ECL) to specify transformations. C-SAW is used to instrument structural changes within the model according to some higher-level requirement that represents

a crosscutting concern. As shown in Figure 11, we developed a model-to-model transformation using C-SAW that takes a CQML model having a clear separation of structural design and the fault tolerance requirements as an input model and generates a structural output model in response to the fault tolerance requirements.

```
1:  aspect FailOverUnit() {
2:    // Apply FailOverUnit aspect on all the component assemblies
3:    rootFolder().models("ComponentAssembly")->copyAndWeave();
4:  }
5:  strategy copyAndWeave() {
6:    // Local declarations.
7:    declare assembly : model;
8:    declare fail_over_unit_list : modelList;
9:
10:    // "self" is equivalent to "this" in Java and C++.
11:    fail_over_unit_list := self.models("FailOverUnit");
12:
13:    // Weave aspect only if there is at least one failover-unit.
14:    if fail_over_unit_list->size() > 0 then
15:
16:       // Make a clone of the original assembly (self) and rename it.
17:       assembly := self.clone(self.parent());
18:       assembly.setName(self.name() + "_Weaved_Assembly");
19:
20:       // For each failover-unit, weave the aspect
21:       fail_over_unit_list->createReplicas();
22:
23:    endif;
24:  }
25:  strategy createReplicas() {
26:    // Local declarations
27:    declare r : integer;
28:    declare par_list : objectList; // List of participants.
29:
30:    // Get the desired number of replicas given as an attribute.
31:    r := self.getAttribute("Replica");
32:
33:    // Find the components participating in the failover-unit (self)
34:    par_list := self.parent().connections("Participates")
35:                             ->select (c | c.destination() == self)
36:                             ->source();
37:
38:    // Replicate the components followed by the connections.
39:    par_list->replicateComponent(r, 1);
40:    par_list->replicateConnections(par_list, r, 1);
41:  }
```

Listing 1: ECL Specification of FailOverUnit Aspect

We used Embedded Constraint Language (ECL) to specify the transformation because

it supports better modularization of the structural changes than an equivalent transformation written in third generation imperative languages such as Java and C++. Step 1 in Figure 11 represents a CQML model, which has fault-tolerance requirements modeled along with system's structure. Step 2 in Figure 11 show how a CAML model is automatically enriched using a transformation written using ECL. The transformation specification is parameterized and accepts the number of desired replicas as a parameter, which is specified as an integer attribute of a FailOverUnit in step 1.

The transformation is divided into multiple ECL strategies (shown in Listing 1 that perform two important steps. First, it creates clones of the participant components of a FailOverUnit. Second, it replicates the interconnections between the primary components into replica components. The result of these two steps is that structurally identical copies of the primary component models are created. This is necessary because the deployment and configuration tools do not distinguish between the primary components and the replica components. Therefore, structurally equivalent replicas of the primary compoenent models are created using automated ECL transformation.

As mentioend before, in component-based DRE systems based on operational string model may require non-isomorphic replication of the application workflow. The outcome of the above model-to-model transformation can be used as a starting point for making the necessary changes assuming that the replica operational string is not vastly different from the primary. At this stage, network level QoS requirement can also be specified on the primary as well as backup operational string models.

III.3.3 Stage 2: Automated replica placement for high-availability

High-availability solution for enterprise DRE systems must minimize the risk of simultaneous failures of replicated functionality. This requires effective replica placement algorithms, where replication is provided for protection units that are modeled as FailOverUnits.

GRAFT uses GME's plugin capabilities to add model interpreters. One such model

interpreter addresses the replica placement problem. The placement model interpreter provides a strategizable framework that can use different constraint-based algorithms to determine an effective replica placement plan to minimize the co-failure probability of the system as a whole.

Formulation of replica placement problem instance in GRAFT. In one instantiation of the formulation of the replica placement problem within our strategizable model interpreter, we use mathematical vectors to represent the distance of the replicas from the primary component. If the primary component has N replicas, then we form N orthogonal vectors, where each vector represents the distance from the primary component node in terms of hops captured in the shared risk group hierarchy. The magnitude of the resultant vector of the N orthogonal vectors is used to compare different deployment configurations and to find the one that satisfies the constraints.

In this formulation of the placement problem algorithm, we have taken care to avoid generation of some obviously undesirable deployment configurations of the system. For example, it does not allow deployment configuration where all the replicas of a component are located in the same host. This is obviously undesirable in dependable enterprise DRE systems because placing multiple replicas in the same host increases the risk of simultaneous failure of replicas.

Prototype heuristic algorithm using the distance metric. The prototype placement algorithm that we have developed maximizes the distance of the replicas from the primary replica but the pair-wise distance between replicas themselves can be small. In other words, the replicas themselves can group together in closely located hosts that are farthest from the primary host. Such a deployment configuration is skewed and undesirable. To alleviate the problem we apply a penalty function to the resultant magnitude of the vector. The penalty function gives more precedence to uniform deployments than highly skewed deployments. The penalty function that we have used is a simple standard deviation of the distances of individual replicas from the primary component. We can generate better

configurations by penalizing highly skewed deployment configurations heavily compared to the more uniform deployment configurations.

```
1. Compute the distance from each of the replicas to the primary for a placement.
2. Record each distance as a vector, where all vectors are orthogonal.
3. Add the vectors to obtain a resultant.
4. Compute the magnitude of the resultant.
5. Use the resultant in all comparisons (either among placements or against a threshold)
6. Apply a penalty function to the composite distance (e.g. pairwise replica distance)
```

Listing 2: Replica Placement Heuristics

For example, consider two resultant vectors $v1\{4,4,4\}$ and $v2\{1,1,8\}$ having 3 dimensions. Although the magnitude of v2 is much greater than v1, the deployment configuration captured in v1 is more desirable than v2 because the replicas are spread across more uniformly around the primary unlike v2. The heuristic algorithm for the prototype implementation of the deployment algorithm is illustrated in Listing 2.

III.3.4 Stage 3: M2M transformation for weaving monitoring infrastructure

in this section we present an AOM solution to automatically generate, deploy, and configure liveness monitoring infrastructure for fault-tolerant component-based systems from their requirements. We use CQML's FailOverUnit modeling capability to capture fault-tolerance requirements of one or more components.

Our solution uses Constraint Specification Aspect Weaver (C-SAW) [130], which is a generalized model-to-model transformation engine for manipulating domain-specific models, which is implemented as a plug-in for the Generic Modeling Environment. It can also be used to instrument structural changes within the model according to some higher-level requirement that represents a crosscutting concern.

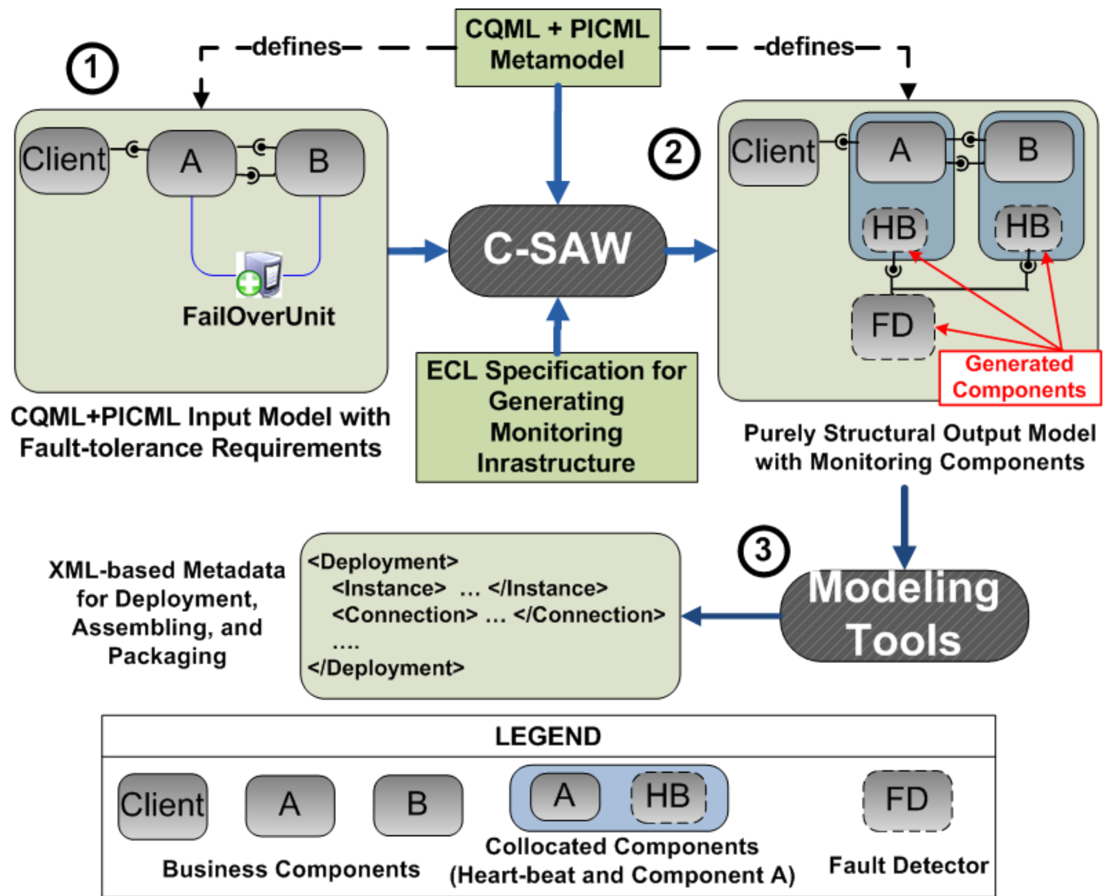

Figure 12: Automatic Weaving of Monitoring Components Using Embedded Constraint Language Specification

As shown in Figure 12, we have developed model transformation specifications for CQML models using C-SAW's input language: Embedded Constraint Language (ECL). These specifications transform CQML models (shown by (1) in Figure 12) with *FailOverUnit* into structural models (shown by (2) in Figure 12) containing monitoring components, their interconnections, and their deployment information. The model-to-model transformation bridges the gap between the higher-level fault-tolerance requirements captured using *FailOverUnit* and lower-level structural models used by existing modeling tools, such as deployment, configuration and packaging tools that generate platform-specific metadata based on structural models. Such a transformation requires several steps, including (1)

generating models of monitoring components, (2) generating the necessary interconnections between the instances of monitoring components, and (3) generate deployment and configuration models for instances of monitoring components so that they are deployed along with the business components when the system is deployed.

Algorithm 1: Transformation Algorithm for Generating Monitoring Infrastructure

1: M : Systems's structural model with annotations.
2: D : Deployment model of the system
3: M_e : Extended M with monitoring components
4: D_e : Deployment model of M_e
5: c : A business component
6: S_c : A set of collocated components such that $c \in S_c$
7: HB_c : Heartbeat component monitoring c
8: F : Fault Detector component.
9: **Input:** M, D
10: **Output:** M_e, D_e (Initially empty)
11: **begin**
12: $M_e := M$
13: $D_e := D$
14: $S_F := \emptyset$
15: F := New fault detector component
16: $M_e := M_e \cup F$
17: $S_F := S_F \cup F$
18: $D_e := D_e \cup S_F$
19: **for** each component c in M **do**
20:     **if** a FailOverUnit is associated with c
21:         **let** HB_c := New heartbeat component for c.
22:         $M_e := M_e \cup HB_c$
23:         **let** i := New connection from F to HB_c.
24:         $M_e := M_e \cup i$
25:         **let** $c \in S_c$ and $S_c \in D$
26:         $S_c := S_c \cup HB_c$
27:         $D_e := D_e \cup S_c$
28:     **endif**
29: **end for**
30: **end**

The algorithm behind the transformation is shown in Algorithm 1. The transformation

accepts system's structural model and a deployment model as input and produces an extended structural model with monitoring components and an extended deployment model with placement of monitoring components as output. A *deployment model* can be viewed as a simple mapping of components to physical hosts in a system. Components are called *collocated* components when they are hosted in the same process on the same host. When a process or a host dies, all the components hosted in that process/host become unavailable, which can be detected remotely using monitoring infrastructure.

The algorithm begins with a copy of system's structural and deployment model in the corresponding extended models. For every structural model, a new Fault Detector component is added in the extended model along with its placement in the deployment model. Followed by that, for every business component in the original structural model, a new *Heartbeat* component is added that is collocated with the business component. The collocated components are placed in the same process as that of the business component at run-time. A connection from the *FaultDetector* component to every new *Heartbeat* component is created so that the former can poll the liveness of the later at runtime. As a result of the algorithm, monitoring components are weaved-in the original structural and deployment models of the system.

```
module Monitor {  // A module defines a namespace.
interface Monitorable { // An interface for checking liveness.
   bool isAlive(void);  // Returns true if component is alive.
};
component Heartbeat { // Implements Monitorable interface.
   provides Monitorable alive; // Used by the FaultDetector.
};
component FaultDetector { // Polls liveness
   requires Monitorable poll; // Uses Heartbeat components.
};
}
```

Listing 3: OMG's IDL 3.0 Interfaces Used by The Fault Monitoring Infrastructure Generated by ECL Transformation

The actual implementation of the *Heartbeat* and *FaultDetector* components is provided

as a library using Component Integrated ACE ORB (CIAO), which is our open-source implementation of OMG's Light-weight CORBA Component Model specification. The component library uses the OMG's IDL 3.0 interfaces shown in Listing 3. The *FaultDetector* component periodically invokes *isAlive* remote method on all the instances of the *Heartbeat* component. The implementation of *isAlive* method in *Heartbeat* component returns true indicating the fact that it is "alive". If a *Heartbeat* component does not respond within a configurable timeout period, the *FaultDetector* concludes that the *Heartbeat* component and the collocated business component have failed. It initiates a recovery procedure after detecting the failure.

III.3.5 Stage 4: Automatic weaving of code for fault-masking and recovery

The MDE tools assist in deploying the entire system and configuring the middleware, however, they do not specialize the middleware. It is necessary for the middleware to be specialized using the domain-specific fault tolerance semantics specified in the MDE tools, without expending any manual effort. To address this challenge, GRAFT uses a deployment-time *generative* approach that augments general-purpose middleware with the desired *specializations*.

GRAFT specializes the client-side middleware stubs. Client-side middleware stubs are used to communicate exceptions to client-side applications so that they can initiate appropriate recovery procedure in response to that. As mentioned in Section III.4.1, these exceptions could be raised because of (1) hardware faults detected by the server or (2) software failure of the server side component itself. Both are examples of *catastrophic* exceptions, in response to which clients must initiate group recovery. To simplify developers' job, GRAFT generates code at deployment-time that augments the *behavior* of the middleware-generated stubs to catch failure exceptions, and initiate domain-specific failure recovery actions.

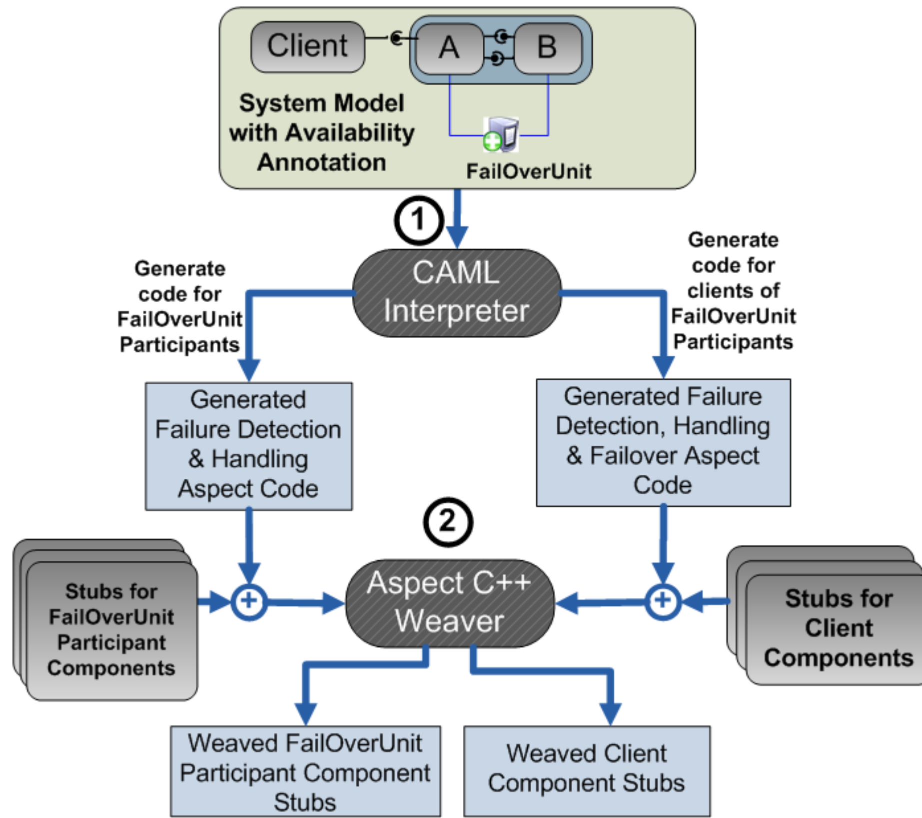

Figure 13: Automated Generation of Failure Detection and Handling Code

GRAFT provides a model interpreter, which (1) traverses the CAML model, (2) identifies the components that participate in FailOverUnits, (3) identifies the components that are clients of the FailOverUnit participant components, and (4) generates modularized source code that provides failure detection and recovery as shown by Step 1 in Figure 13. Depending upon the role of the component, two different types of behaviors are generated by the interpreter.

We have identified two different roles of components with respect to a FailOverUnit: (1) participants of a FailOverUnit (*e.g.,* FC component) and (2) non-participant client components that are directly communicating with one or more participants of the FailOverUnit (*e.g.,* MFC component). The participants of a DPU do not failover, however, clients of a DPU fail over to a replica FailOverUnit. To allow this difference in the behavior, failover

code is generated only for the client components whereas the code for FailOverUnit participant components do not perform failover; instead they trigger failover in the client components of the FailOverUnit.

GRAFT encodes this difference in behavior by generating different AspectC++ code for each component associated with a FailOverUnit depending upon whether the component is a participant or a client. For participant components, for every method in the interface that can potentially raise a catastrophic exception, an *around* advice is generated that catches exceptions representing *catastrophic* failure and initiates a shutdown procedure for all the participant components. For the client components, however, a different around advice is generated that not only detects the failure and initiates a group shutdown procedure but also performs an automatic failover to a replica FailOverUnit.

To modularize and transparently weave the failure detection and recovery functionality within the stubs, GRAFT leverages Aspect-oriented Programming (AOP) [68] support provided by the AspectC++ [131] compiler. The CAML model interpreter generates AspectC++ code,[1] which is then woven by the AspectC++ compiler into stubs at the client side producing specialized stub implementations as shown by Step 2 in Figure 13. Finally, the specialized source code of the stubs are compiled using a traditional C++ compiler.

III.3.6 Stage 5: Automatic synthesis of deployment metadata for high-availability

The model interpreters and generative tools in GRAFT use the dependability requirements captured in the models for synthesizing metadata used to provision dependability for enterprise DRE systems. In order to realize such an automation in the provisioning process several artifacts of dependability must be addressed: (a) the designer of the dependable system has to annotate the desired degree of replication of the protected components in the model, (b) the generative tools have to process the replication requirements and produce deployment metadata that reflects the number of physical software components that

[1] Due to space restrictions we are not showing the generated aspect code.

will actually be deployed, and (c) derive the complex connection topology interconnecting the generated components, which is dictated by the degree and style of replication of the primary component as well as replication requirements of the components it interacts with.

Deployment metadata generation framework As noted earlier, the component middleware platforms used to host the enterprise DRE systems use standardized XML-based metadata descriptors to describe the deployment plans of the entire system, which the runtime system uses to actually deploy the different components of the system. Our challenge involved enhancing the metadata descriptors to include high availability provisioning decisions. For this goal to realize, GRAFT's generative capabilities had to be integrated with the existing generators available in CQML without obtrusive changes to existing capabilities. This approach ensures that generators for QoS issues beyond dependability, such as network QoS, security, can seamlessly be integrated with CQML.

To address these concerns, we have developed an extensible framework called *The Deployment Plan Framework* that allows augmentation of metadata generation "on-the-fly" as it is being generated. The framework exposes a fixed set of hooks to be filled in by the developer of the existing and any new CQML model interpreters including the GRAFT model interpreters. The main job of the deployment framework is to generate the standardized metadata describing the components, their implementations, their inter-connections and so on. Additionally, it invokes predefined hook methods implemented by different QoS model interpreters of CQML. The GRAFT model interpreter implements a subset of a large set of different possible hook methods. The hook methods "inject" auto-generated standardized metadata in response to the availability requirements captured in the model. The metadata generated on-the-fly blends into the other standardized metadata.

This architecture allows large scale reuse of earlier code base that deals with the basic structure and composition capabilities of PICML/CQML. The developer producing QoS enhancements to the existing modeling capabilities of CQML need not be concerned with the other complexity of the framework and the format of the standardized descriptors, but

simply add/modify the metadata for the QoS dimension they are addressing. Our GRAFT model interpreter exploits these capabilities of the Deployment Plan Framework to "inject" three different kinds of metadata.

- **Replica component instances** of the primary protected component depending upon replication degree annotated in the model. For example, if replication degree of an FOU is 3, then two replicas of the primary FOU are created. Thus, two replicas of each component in the FOU are effectively added by the interpreter.
- **Component connection metadata** is injected based on the replication style and degree of replication. The incoming connections to the protected components are marked with special annotations so that the run-time system can use suitable implementations to realize them. One such possible annotation is IOGR, i.e. Interoperable Object Group Reference. IOGR is a part of the FT-CORBA [92] standard.
- **Deployment metadata** is the assignment of components to computing resources available in the system. This metadata includes information for all the primary protected components, their replicas and the monitoring infrastructure components (e.g. Heartbeat components).

Handling complex connections. As shown in Figure 14, shows the effect of the replication style and the degree of replication on the complexity of the connection establishment. In the original system, the Processor component and the Planner component have exactly one connection between them. The Figure 14 captures the multiplicative increase in the number of connections when both, the Processor component and the Planner component, are protected using active replication. Each Processor component, primary as well as its replica has to make three connections to each member of the Planner replica group because the degree of replication of the Planner fail over unit (FOU) is three. In general, if the source component of the connection is replicated M times and the destination component is replicated N times then the number of connections grow by a factor of M x N.

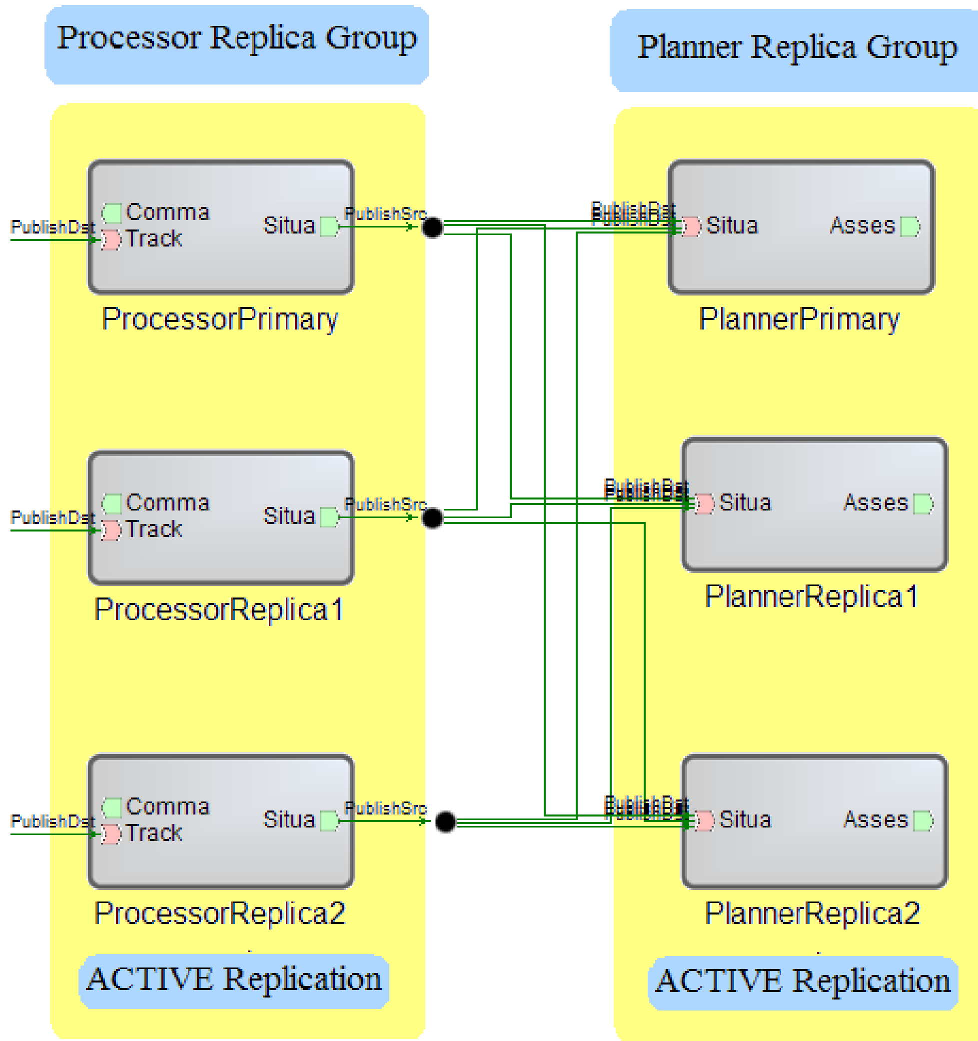

Figure 14: Complexity of connection generation

Note that the diagram only indicates the necessary number of connections the middleware has to establish when components are deployed. These connection may or may not actually be used to send requests across because it really depends upon where request/reply suppression is in place. Nevertheless, the component container has to prepare for any unforeseen failures and has to establish connections *apriori* in order to avoid the latency of connection establishment later when failures occur. The model interpreter that we have developed completely hides away the complexity of modeling the component replica instances and the connections between them.

III.4 Evaluation of GRAFT

In this section we evaluate the model-to-model, model-to-text transformation capabilities of GRAFT. First we present a representative case-study and later evaluate GRAFT by measuring the efforts saved to specialize middleware in the context of the case-study. Additionally we also qualitatively validate the runtime behavior of the specialized middleware in meeting the fault tolerance requirements of the MHS case study.

III.4.1 Case-study for GRAFT

To better present our GRAFT solution, we illustrate a case study that benefits from GRAFT to realize its fault tolerance requirements. Our case study is a warehouse *material handling system* (MHS). A MHS provides automated monitoring, management, control, and flow of warehouse goods and assets. A MHS represents a class of conveyor systems used by couriers (*e.g.,* UPS, DHL, and Fedex), airport baggage handling, retailers (*e.g.,* Walmart and Target), food processing and bottling.

Architecture. The software components in the MHS architecture can be classified as (1) *management* components, which make decisions such as where to store incoming goods, (2) *material flow control* (MFC) components, which provide support for warehouse management components by determining the routes the goods have to traverse, and (3) *hardware interface layer* (HIL) components, which control MHS hardware, such as conveyor belts and flippers.

Figure 15 shows a subset of the MHS operations, where a MFC component directs goods within the warehouse using the route BELT A→BELT B or the route BELT A→BELT C. Flippers F and F′ assist in directing goods from BELT A to BELT B and BELT C, respectively. Further, as shown in Figure 15, HIL components, such as Motor Controllers (MC1, MC2, MC1′, MC2′) and the Flipper Controller (FC, FC′), control the belt motors and flippers, respectively. The MFC component instructs the Flipper Controller component

to flip, which in turn instructs the Motor Controller components to start the motors and begin transporting goods.

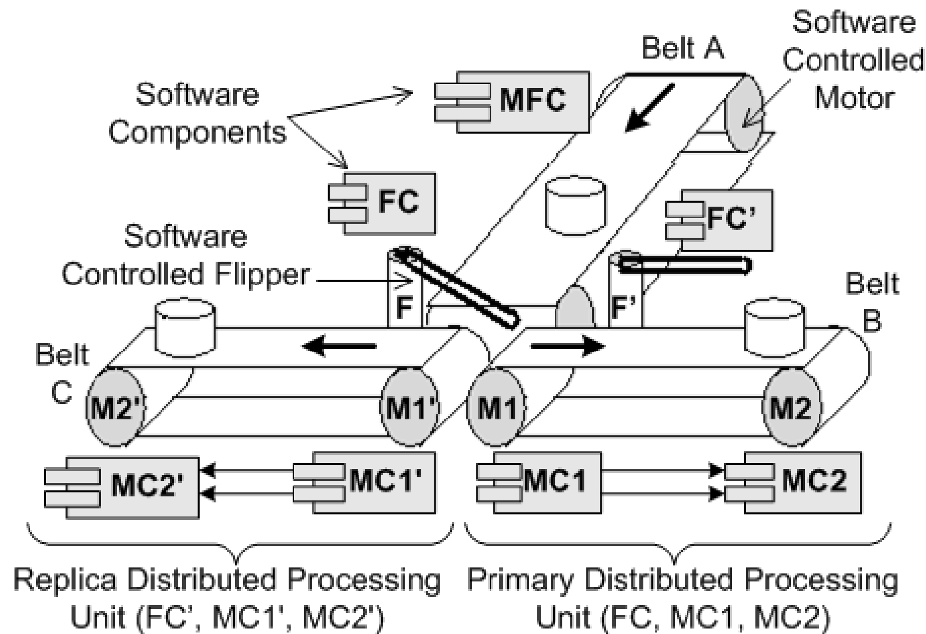

Figure 15: A Distributed Processing Unit Controlling Conveyor Belts

Domain-specific Fault Model. As goods are transported using different conveyor belts, faults could occur. Two broad kinds of faults are possible in the MHS system: (1) hardware faults, (*e.g.,* jamming of the flipper) and (2) software faults, (*e.g.,* MC or FC component crashes). Hardware faults in the MHS system are detected by their associated HIL components and communicated using application-specific software exceptions. Software faults, such as software component crashes, are detected by the clients of those components using system-level software exceptions generated by the underlying middleware. Both types of faults affect the reliable and correct operation of the MHS system, and are classified as *catastrophic* faults.

Domain-specific Failure Handling and Recovery Semantics. Failure recovery actions in MHS are based on *warm-passive* replication semantics. When catastrophic faults are

detected in a MHS, the desired system response is to shutdown the affected hardware assembly and activate a backup hardware assembly automatically. For example, when one of the motors of BELT B or flipper F fails, the MFC component should stop using the BELT B and route the packages via BELT C instead. The consequence of such a decision means that the HIL components associated with BELT B should be deactivated and those with BELT C as well as flipper F′ need to be activated.

The MHS thus imposes a *group*-based fault tolerance semantics on the software components controlling the physical hardware. If any one component of the group fails, the failure prevents the whole group from functioning and warrants a failover to another group. We call this group of components as a distributed processing unit (DPU) – in this case MC1, MC2 and FC for BELT B. Further, the clients of a DPU (*e.g.*, the MFC component) must failover to an alternative DPU if any of the components in the primary DPU fails.

	Fault-tolerance Modeling Efforts		
Component Name	**# of original connections**	**# of replica components**	**# of replica connections**
Material Flow Control	1 / 1	0 / 0	2 / 0
Flipper Controller	2 / 2	2 / 0	4 / 0
Motor Controller 1	1 / 1	2 / 0	2 / 0
Motor Controller 2	2 / 1	2 / 0	2 / 0

Table 3: Savings in Fault-tolerance Modeling Efforts in Developing MHS Casestudy Without/With GRAFT

III.4.2 Evaluating savings in effort to specialize middleware

Table 3 shows the manual efforts saved by adopting GRAFT's approach in designing and developing the MHS case study described in earlier. The table shows that there is reduction in the efforts of modeling replica components and connections for all the four

	Fault-tolerance Programming Efforts		
Component Name	**# of try blocks**	**# of catch blocks**	**Total # of lines**
Material Flow Control	1 / 0	3 / 0	45 / 0
Flipper Controller	2 / 0	6 / 0	90 / 0
Motor Controller 1	0 / 0	0 / 0	0 / 0
Motor Controller 2	0 / 0	0 / 0	0 / 0

Table 4: Savings in Fault-tolerance Programming Efforts in Developing MHS Casestudy Without/With GRAFT

components. The declarative nature of CQML's FailOverUnit annotations and the automated model-to-model transformation thereafter, obviates the need for modeling the replica components and connections explicitly, resulting in a modular design of the MHS system.

A significant reduction in programming efforts is achieved due to automatic generation of code that handles failure conditions at runtime in the MHS system. Listing 4 shows a sample generated AspectC++ code from the CAML model of our MHS case-study. The generated code for each component is different depending upon the number of remote interfaces used by a component, the number of methods in each remote interface, and the types of exceptions raised by the methods. The number of `try` blocks in Table 3 corresponds to the number of remote methods whereas the number of `catch` blocks correspond to the number of exceptions.

For example, when MFC component invokes a method of the FC component, 45 lines of aspect code is generated to handle group recovery semantics for that one function call alone. GRAFT's approach yields higher savings in modeling and programming efforts for larger, more complex systems, which may have hundreds of components with tens of them requiring fault-tolerance capabilities.

```
1:  aspect FailOverUnit_Client {
2:    // Auto-generated array of names of FailOverUnit participants.
3:    char * FOU_Participants[] = { "FlipperController",
4:                                  "MotorController1",
5:                                  "MotorController2",
6:                                  0 };
7:    size_t failure_count_; // Initialized to zero.
8:
9:    // Contains remote object reference of the replica.
10:    HIL::IFlipperController_var replica_ref_;
11:
12:    // Weave advice around local stub of the flip() method of MFC.
13:    advice execution ("void HIL::IFlipperController::flip()")
14:    : around () // The advice is applied around the flip method.
15:    {
16:      do {
17:        // Use the remote reference of the backup FlipperController
18:        // component only if the primary component has failed.
19:
20:        if (failure_count_ > 0)
21:        // "_that" is used to change "this" pointer before proceeding.
22:        // Use live object reference of the replica.
23:          tjp->action()._that = replica_ref_.in();
24:
25:        try {
26:          // Continue the flip() function call as usual.
27:          tjp->proceed ();
28:          break;
29:        }
30:        catch(HIL::FlipperJamException & e) {
31:          handle_exception(e); // deactivates FailOverUnit participants
32:        }
33:        catch(CORBA::COMM_FAILURE & e) {
34:          handle_exception(e); // deactivates FailOverUnit participants
35:        }
36:        catch(CORBA::TRANSIENT & e) {
37:          handle_exception(e); // deactivates FailOverUnit participants
38:        }
39:        // Application-specific non-catastrophic exceptions are passed.
40:      } while (replica_ref_.in() != NULL_POINTER);
41:    }
42:    // For other functions of the IFlipperController interface, similar
43:    // around advices are generated with corresponding execution
44:    // pointcuts and catastrophic exceptions.
45:  };
```

Listing 4: Generated AspectC++ code for Transparent Fault Masking

III.4.3 Qualitative validation of runtime behavior

Figure 16 shows how the specialized stubs generated by GRAFT react to failures at runtime and provide group recovery semantics. To control the lifecycle of the components, the aspect code communicates with domain application manager (DAM), which is a standard deployment and configuration infrastructure service defined in LwCCM. It provides high-level application programming interface (API) to manage lifecycle of application components. Below, we describe the steps taken by GRAFT when a catastrophic exception is raised.

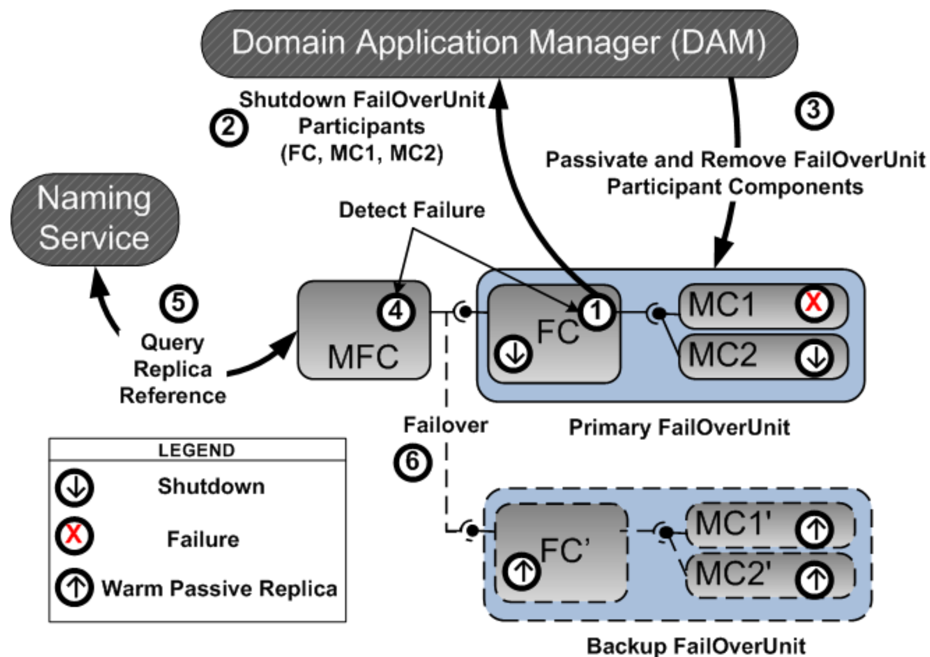

Figure 16: Runtime Steps Showing Group Recovery Using GRAFT

1. As shown in Figure 16, MFC component directly communicates with the FC component, which in turn communicates with MC1 and MC2 components. Consider a scenario where FC makes a call on MC1 and MC1 detects a motor failure and raises *MotorFailureException*. The exception is caught by the generated aspect code in FC indicated by (1) in Figure 16.

2. The specialized stubs in FC, initiate shutdown of the primary DPU by instructing the DAM to remove participating components of the primary DPU (FC, MC1, and MC2), including itself.
3. DAM instructs the containers hosting the primary DPU components (FC, MC1, and MC2) to passivate and remove the components.
4. Removal of FC component triggers a system-level exception at the MFC component, which is again caught by the specialized stub at MFC-side.
5. The specialized stubs for MFC fetch a reference of FC′ from the naming service. The naming service is assumed to be pre-configured at deployment-time with lookup information for all the components in the system.
6. MFC successfully fails over to the replica DPU (FC′, MC1′, and MC2′) and resumes the earlier incomplete remote function call. Finally, FC′ communicates with MC1′ and MC2′ to drive the belt motors of the backup BELT C and continues the operation of MHS system without interruption.

III.5 Deployment-time Network QoS Provisioning Framework

Network Resource Allocator Framework (NetRAF) is a resource allocator engine that allocates network resources for DRE systems using a variety of network QoS mechanisms, such as DiffServ and IntServ. As shown in Figure 17, the NetQoS modeling capabilities in CQML capture the modeled per-flow network QoS requirements in the form of a *deployment plan* that is input to NetRAF.

When using NetQoS, application developers only annotate the connection between the component instances. Since NetRAF operates on the *deployment plan* that captures this modeling effort, network QoS mechanisms are used only for the connection on which QoS attributes are added. NetRAF thus improves conventional approaches [117] that modify application source code to work with network QoS mechanisms, which can become complex when source code is reused in a wide range of deployment contexts.

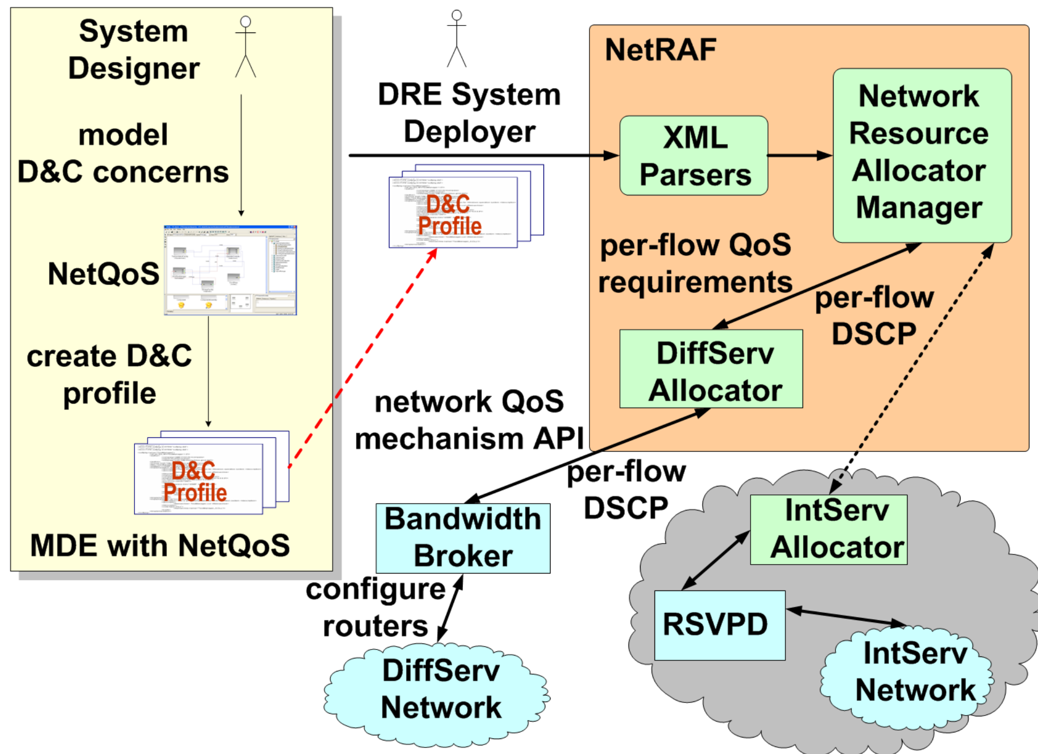

Figure 17: NetRAF's Network Resource Allocation Capabilities

NetRAF's *Network Resource Allocator Manager* accepts application QoS requests at pre-deployment-time and determines the network QoS mechanism (*e.g.*, DiffServ or IntServ) to use to serve the requests. As shown in Figure 17, NetRAF's Network Resource Allocator Manager works with QoS mechanism-specific allocators (*e.g.*, DiffServ Allocator), which shields it from interacting directly with complex APIs for network QoS mechanisms (*e.g.*, DiffServ Bandwidth Broker), thereby enhancing NetQoPE's flexibility.

Multiple allocators (*e.g.*, IntServ Allocator and DiffServ Allocator) can be used by NetRAF's Network Resource Allocator Manager to serve the needs of small-scale deployments (where IntServ and DiffServ are both suitable) and large-scale deployments (where DiffServ often provides better scalability). For example, the shaded cloud connected to the Network Resource Allocator Manager in Figure 17 shows how NetRAF can be extended to work with other network QoS mechanisms, such as IntServ.

NetRAF invokes the Bandwidth Broker's admission control capabilities [31] by feeding

it one application flow at a time. If all flows *cannot* be admitted, NetRAF allows developers an option to modify the deployment context since applications have not yet been deployed. Example modifications include changing component implementations to consume fewer resources or change the source and destination nodes. As demonstrated in later in this chapter, this capability helps NetRAF incur lower overhead than conventional approaches [117, 151] that perform validation decisions when applications are deployed and operated at runtime.

NetRAF's DiffServ Allocator instructs the Bandwidth Broker to reserve bi-directional resources in the specified classes. The Bandwidth Broker determines the bi-directional DSCPs and NetRAF encodes those values as connection attributes in the deployment plan. In addition, the Bandwidth Broker uses its *Flow Provisioner* [31] to configure the routers to provide appropriate per-hop behavior when they receive IP packets with the specified DSCP values. Component containers are auto-configured to add these DSCPs when applications invoke remote operations.

III.6 Evaluation of Network QoS Provisioning Framework

III.6.1 Case-study for NetQoS

Figure 18 shows a representative DRE system in an office enterprise security and hazard sensing environment, which we use as a case study to demonstrate and evaluate NetQoPE's model-driven, middleware-guided network QoS provisioning capabilities.

Enterprises often transport network traffic using an IP network over high-speed Ethernet. Network traffic in an enterprise can be grouped into several classes, including (1) e-mail, videoconferencing, and normal business traffic, and (2) sensory and imagery traffic of the safety/security hardware (such as fire/smoke sensors) installed on office premises. Our case study makes the common assumption that safety/security traffic is more critical than other traffic, and thus focuses on model-driven, middleware-guided mechanisms to

assure the specified QoS for this type of traffic in the presence of other traffic that shares the same network.

As shown in Figure 18, our case study uses software controllers to manage hardware devices, such as sensors and monitors. Each sensor/camera software controller filters the sensory/imagery information and relays them to the monitor software controllers that display the information. These software controllers were developed using Lightweight CCM (LwCCM) [98] and the traffic between these software controllers uses a bandwidth broker [31] to manage network resources via DiffServ network QoS mechanisms. Although the case study in this chapter focuses on DiffServ and LwCCM, NetQoPE is designed for use with other network QoS mechanisms (*e.g.*, IntServ) and component middleware technologies (*e.g.*, J2EE).

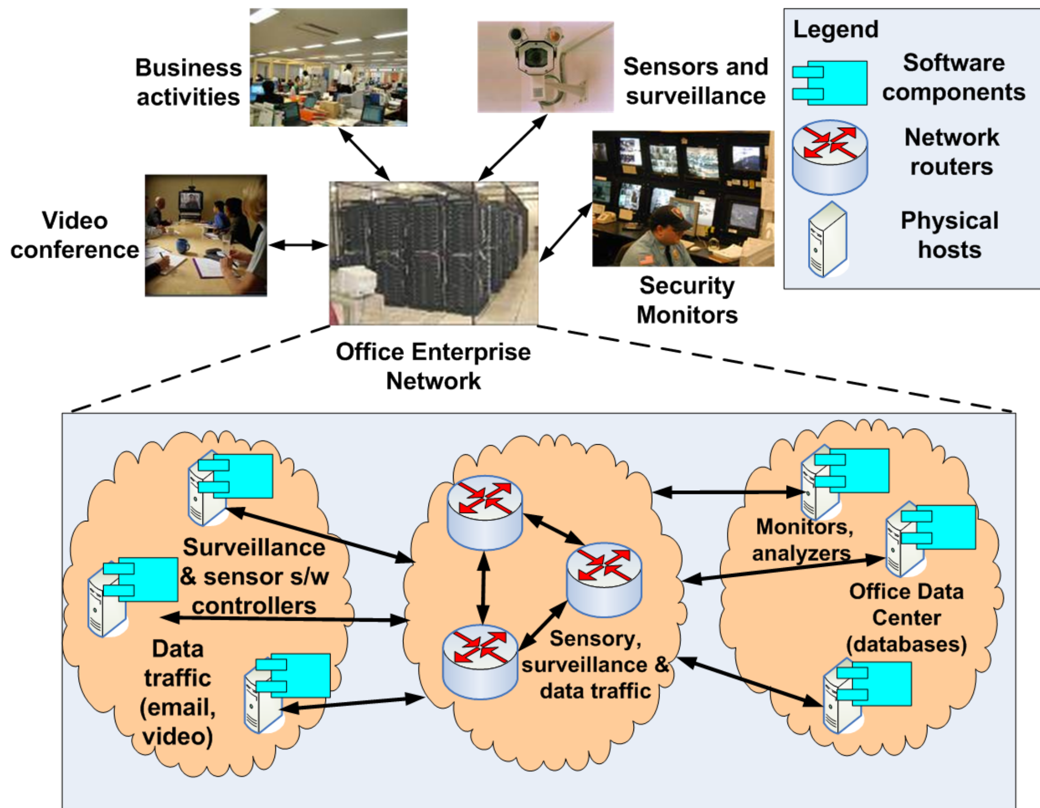

Figure 18: Network Configuration in an Enterprise Security and Hazard Sensing Environment

Component-based applications in our case study use bandwidth broker services via the following middleware-guided steps: (1) network QoS requirements are specified on each application flow, along with information on the source and destination IP and port addresses, (2) the bandwidth broker is invoked to reserve network resources along the network paths for each application flow, configure the corresponding network routers, and obtain per-flow DSCP values to help enforce network QoS, and (3) remote operations are invoked with appropriate DSCP values added to the IP packets so that configured routers can provide per-flow differentiated performance.

III.6.2 Evaluating Model-driven Network QoS Provisioning

Rationale. NetQoPE is designed to provide network QoS to applications in an extensible manner. This experiment evaluates NetQoPE's application-transparent network QoS provisioning capabilities.

Methodology. We first define a taxonomy for evaluating technologies that provide network QoS assurances to end-to-end DRE application flows. Conventional approaches can be classified as being (1) object-oriented [40, 113, 117, 151], (2) aspect-oriented [38], and (3) component middleware-based [33, 123]. We now describe how each approach provides the following functionality needed to leverage network QoS mechanism capabilities:

- **Requirements specification.** In conventional approaches applications use (1) middleware-based APIs [40, 151], (2) contract definition languages [113, 117], (3) runtime aspects [38], or (4) specialized component middleware container interfaces [33] to specify network QoS requirements. Whenever the deployment context and the associated QoS requirements change, however, application source code must also change, thereby limiting reusability. In contrast NetQoS provides domain-specific, declarative techniques that increase reusability across different deployment contexts and alleviate the need to programatically specify QoS requirements.

- **Network resource allocation.** Conventional approaches require the deployment of

applications before their per-flow network resource requirements can be provisioned by network QoS mechanisms. If the required resources cannot be allocated for those applications they must be stopped, new resource requirements must be specified, and the resource reservation process must be restarted. This approach is tedious since it involves deploying and re-deploying applications multiple times (potentially on different nodes). In contrast, NetRAF handles deployment changes through NetQoS models and allocates resources during pre-deployment before applications have been deployed. This reduces the effort needed to change deployment topology or application QoS requirements.

• **Network QoS enforcement.** Conventional approaches modify application source code [113] or programming model [33] to instruct the middleware to enforce runtime QoS for their remote invocations. Applications must therefore be designed to handle two different usecases—to enforce QoS and when no QoS is required—thereby limiting application reusability. In contrast NetCON uses a container programming model that transparently enforces runtime QoS for applications without changing their source code or programming model.

Using the conventional approaches and the NetQoPE approach, we now compare the manual effort required to provide network QoS to the 4 end-to-end application flows described in the case-study. We decompose the manual effort across the following general steps: (1) *implementation*, where software developers write code, (2) *deployment*, where system deployers map (or stop) application components on their target nodes, and (3) *modeling tool use*, where application developers use NetQoPE to model a DRE application structure and specify per-flow QoS requirements. In our evaluation, a complete QoS provisioning lifecycle consists of specifying requirements, allocating resources, deploying applications, and stopping applications when they are finished.

To compare NetQoPE with manual efforts, we devised a realistic scenario for the 4 end-to-end application flows described the case-study. In this scenario, three sets of experiments were conducted with the following deployment variants:

• In the first variant, all 4 end-to-end application flows were configured with the QoS requirements specified in the case-study.

• In the second variant, to demonstrate the effect of changes in QoS requirements on manual efforts we modified the bandwidth requirements from 20 Mbps to 12 Mbps for each end-to-end flow.

• In the third variant, we demonstrate the effect of changes in QoS requirements and resource (re)reservations taken together on manual efforts. We modified bandwidth requirements of all flows from 12 Mbps to 16 Mbps. We also changed temperature sensor controller component to use the high reliability (HR) class instead of the best effort BE class, as described the case-study. Finally, we increased the background HR class traffic across the blades so that the resource reservation request for the flow between temperature sensor and monitor controller components fails. In response, deployment contexts (*e.g.*, bandwidth requirements, source and destination nodes) were changed and resource re-reservation was performed.

For the first deployment, the manual effort required using conventional approaches involved 10 steps: (1) modify source code for each of the 4 components to specify their QoS requirements (8 implementation steps), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step). Conversely, the effort required using NetQoPE involved the following 4 steps: (1) model the DRE application structure of all 4 end-to-end application flows using NetQoS (1 modeling step), (2) annotate QoS specifications on each end-to-end application flow (1 modeling step), (3) deploy all components (1 deployment step), and (4) shutdown all components (1 deployment step).

For the second deployment, the effort required using a conventional approach is also 10 steps since source code modifications are needed as the deployment contexts changed (in this case, the bandwidth requirements changed across 4 different deployment contexts). In contrast, the effort required using NetQoPE involves 3: (1) annotate QoS specifications on

each end-to-end application flow (1 modeling step), (2) deploy all components (1 deployment step), and (3) shutdown all components (1 deployment step). Application developers also reused NetQoS's application structure model created for the initial deployment, which helped reduce the required efforts by a step.

For the third deployment, the effort required using a conventional approach is 13 steps: (1) modify source code of each of the 8 components to specify their QoS requirements (8 implementation steps), (2) deploy all components (1 deployment step), (3) shutdown the temperature sensor component (1 deployment step – resource allocation failed for the component), (4) modify source code of temperature sensor component back to use BE network QoS class (deployment context change) (1 implementation step), (5) redeploy the temperature sensor component (1 deployment step), and (6) shutdown all components (1 deployment step).

In contrast, the effort required using NetQoPE for the third deployment is 4 steps: (1) annotate QoS specifications on each end-to-end application flow (1 modeling step), (2) begin deployment of the all components, but NetRAF's pre-deployment-time allocation capabilities determined the resource allocation failure and prompted the NetQoPE application developer to change the QoS requirements (1 pre-deployment step) (3) re-annotate QoS requirements for the temperature sensor component flow (1 modeling step) (4) deploy all components successfully (1 deployment step), and (5) shutdown all components (1 deployment step).

Table 5 summarizes the step-by-step analysis described above. These results show that

Approaches	# Steps in Experiment Variants		
	First	Second	Third
NetQoPE	4	3	5
Conventional	10	10	13

Table 5: Comparison of Manual Efforts Incurred in Conventional and Model-driven NetQoS Approaches

conventional approaches incur roughly an order of magnitude more effort than NetQoPE to provide network QoS assurance for end-to-end application flows. Closer examination shows that in conventional approaches, application developers spend substantially more effort developing software that can work across different deployment contexts. Moreover, this process must be repeated when deployment contexts and their associated QoS requirements change. Moreover, implementations are complex since the requirements are specified using middleware [151] and/or network QoS mechanism APIs [72].

Moreover, application (re)deployments are required whenever reservation requests fail. In this experiment, only one flow required re-reservation and that incurred additional effort of 3 steps. If there are a large number of flows—and enterprise DRE systems like our case study often have dozens or hundreds of flows—the amount of effort required is significantly more than for conventional approaches.

CHAPTER IV

END-TO-END RELIABILITY OF NON-DETERMINISTIC STATEFUL COMPONENTS

The previous two chapters discussed the techniques for modeling fault-tolerance requirements and weaving them into system artifacts to transparently achieve highly available component-based DRE systems. Although the presented techniques significantly reduce the efforts needed to provision fault-tolerance, they do not adequately address the run-time state consistency issues that arise when operational strings of non-deterministic stateful components are replicated to achieve high availability. This chapter presents the challenges and a novel solution that ensures that the data in operational strings remains consistent and timely despite failures.

IV.1 Introduction

Operational strings are inherently multi-tier systems where server components act as clients of other components giving rise to nested invocations. The systems that combine replication and nested invocation must deal with the side-effects of replicated invocations. A *replicated invocation* is a (nested) request from a replicated server to another (possibly replicated) server [133].

In replicated invocation, care must be taken with stateful components so that the client's invocation of a service appears to have executed *exactly once* despite partial failures. Exactly-once semantics are highly desirable because it simplifies client-side programming. Moreover, it improves the perceived availability of the system in the client's view. Even if the request or its subcomponents are physically executed more than once due to failures, the modifications to the state and the reply to the client must be as if it was executed only once without failures.

The exact solution to provide exactly-once semantics in the presence of replicated invocations depends on whether the system admits non-determinism or not. For deterministic components, simple caching of request and replies is sufficient [32]. Duplicate invocations are suppressed and the cached reply from the previous computation is returned instead. Real-time systems, however, exhibit several [108] forms of non-determinism such as local information (*e.g.,* sensors and clocks), timers and timeouts, multi-threading (*e.g.,* dynamic scheduling, preemption), load-balancing, time-dependent sensor calibration, and non-deterministic program constructs such as true random number generators. Reusing cached results to avoid side-effects of replication is not acceptable because the result of the re-invocation after failure may not be identical to the earlier execution. If the nested invocations have one or more non-deterministic components in the call-chain, the problem of *orphan request* [69, 133] may arise if one of them fails.

An *orphan request* is a request received by a component that is no longer valid due to failure of the client component. Note that the client component itself acts like a server to another component forming a chain of nested invocations. If the failing component is non-deterministic, it is not guaranteed that the reinvocation of the request will lead to the same nested invocation. Thus leaving the earlier partially completed request an orphan.

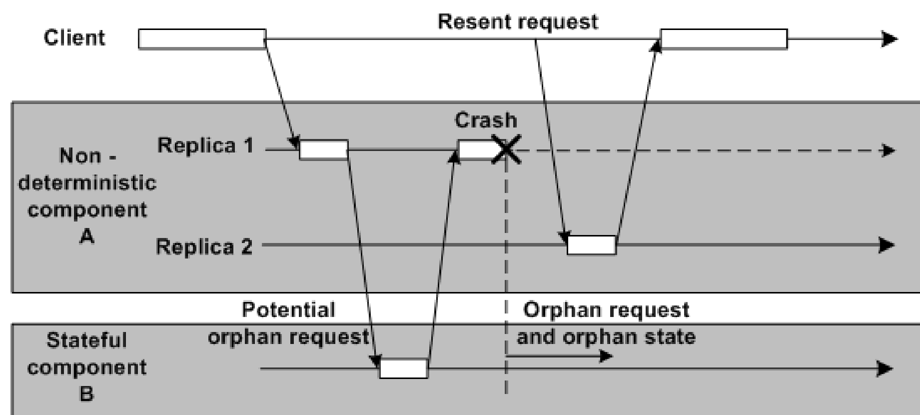

Figure 19: An orphan request caused by the failure of non-deterministic component A

Figure 19 shows how a non-deterministic component causes an orphan request in another component. The client invokes an operation on component A, which in turn calls another component B. Component A, however, crashes before returning the reply to the client. The client reinvokes the request on the replica of component A. As component A is a non-deterministic, there are three possible outcomes: (1) no invocation, (2) an invocation with different parameters, and (3) an identical invocation on component B. The lattermost, however, is not guaranteed all the time. If the replica 2 of component A makes no invocation on component B, the earlier request is considered an orphan. If an invocation with different parameters is made, then the state of component B may become inconsistent with respect to the rest of the system. Both of the above conditions violate the exactly-once semantics.

Note that orphan computations are an instantaneous consequence of a failure in the non-deterministic middle tier. Before the failure, the requests sent by the middle tier components are *potential* orphan requests (as shown in Figure 19). Upon failure of the sender, however, they instantaneously become orphans. Therefore, orphan computations can not be prevented but rather must be rectified because they waste resources, may hold locks delaying other computations, and may lead to system inconsistency.

In this chapter, we present a novel approach to the problem of orphan elimination for systems that must meet both real-time and fault-tolerance requirements. We present a *group-failover* protocol, which ensures that replicated data is both consistent and timely. Group-failover protocol supports exactly-once execution semantics in the presence of replicated invocations and non-deterministic stateful components. It ensures strong state consistency between the primary components and their replicas while avoiding the use of transactions. Moreover, programmers do not have to implement application-specific *prepare*, *commit*, and *rollback* methods. We present two variations of the protocol which have different overheads during fault-free and failure scenarios.

The work presented in this chapter differs from our earlier work on FLARe [12] and

CORFU [153]. FLARe provides real-time, fault-tolerance to client-server applications and not end-to-end task chains. Moreover, FLARe provides only weak state consistency. CORFU exposes FLARe's capabilities at the component abstraction level thereby allowing end-to-end task chains. However, CORFU does not handle the orphan request problem. Nevertheless, our work leverages the infrastructure provided in FLARe and CORFU, which themselves are part of the CIAO framework.

The chapter is organized as follows. Section IV.2 presents the related work in detail and describes why existing approaches are not suitable for real-time systems; Section IV.4 describes the system model; Section IV.5 presents the group-failover protocol and its two variations; Section IV.6 evaluates the protocol and shows its suitability for distributed real-time systems.

IV.2 Related Research

We categorize the existing research work on addressing the issues of non-determinism in replicated invocation in two categories. The first set of work admits non-determinism but compensates for the side effects of replicated invocation using transactions. The second set of work focuses on enforcing determinism and thereby avoiding the side effects.

IV.2.1 Integrated transaction and replication services

Replication and transactions are two separate techniques for achieving fault-tolerance in reliable systems. Replication represents *roll-forward* recovery where an incomplete request is re-executed at another replica. On the other hand, transactions represent *roll-back* recovery where a failed parent transaction forces undo of all the sub-transactions. The two reliability mechanisms are different in the sense that the prior protects processing whereas the later protects data to ensure system consistency. Note that transactions provide *all-or-nothing* (atomicity) guarantee whereas replication provides *at-least-once* guarantee as long as there are fewer failures than the available replicas. Neither provide *exactly-once*

guarantee, which is stronger than both. Therefore, the solutions that depend on both replication and transactions to provide *exactly-once* semantics must integrate the two services in a non-trivial way.

Felber *et al.* [44] reconcile CORBA's transaction service (OTS) [91] and the replication service (FT-CORBA) [103] to protect both data and processing to provide consistent end-to-end reliable operation. Their approach restarts execution of a failed sub-transaction on a backup and aborts sub-transactions where a parent has failed. This reconciliation, however, does not handle the intricate details of transaction completion in failure scenarios, which are handled in [133].

Pleisch *et al.* [133] describe two alternatives to handle non-determinism; one pessimistic and one optimistic, which is an improvement over [44]. The first forces the sub-transaction to wait for the commit of the parent, while the latter allows subtransactions to commit before its parent. Information about how to undo the changes is sent to the backups before making the nested invocation. In the pessimistic case, orphan subtransactions are aborted whereas in the optimistic case they are compensated by *undo* transactions.

Frølund *et al.* [46] present an approach to integrate replication and transactions for three-tier applications. However, their support is limited to stateless middle-tier servers, where all state is forced in the end-tier databases. ITRA [36] handles the side effects of replication by propagating the result of each non-deterministic operation to the backups. ITRA supports replicated transactions by replicating the start, join, prepare, commit, and abort operations.

Kolltveit *et al.* [69] present an approach where a passively replicated transaction manager is allowed to break replication transparency to abort orphan requests, thus handling non-determinism. The transaction manager is aware of the replication and able to see the individual replicas of the transaction participants instead of the set of replicas as an opaque group.

Although the prior work addresses the orphan request problem in the context of transactional enterprise systems, the use of transactions becomes a significant source of overhead in the critical path for most real-time systems. During failure recovery, orphan components are identified and rolled-back using undo information, which causes unpredictable increase in the client-response time. Moreover, transactions are not transparent because programmers are required to implement application-specific *prepare*, *commit*, and *rollback* operations.

IV.2.2 Enforcing determinism

Considerable research efforts have been expended in designing strategies to enforce replica determinism and to circumvent specific sources of nondeterminism. Slember *et al.* [128, 129] apply program analysis to discover the sources of non-determinism. They target the instances of nondeterminism that can be compensated automatically and highlight the remaining instances that must be manually rectified. Work on deterministic scheduling algorithms [18, 65] handles the non-determinism of multi-threading. A deterministic schedule is ensured either by multicasting the scheduling decisions to the replicas or by assuming shared state between all threads of the same replica.

The fault-tolerant real-time MARS system [109] requires deterministic behavior in highly responsive automotive applications which exhibit nondeterminism due to time-triggered event activation and preemptive scheduling. Replica determinism is enforced using a combination of timed messages and a communication protocol for agreement on external events.

Finally, timestamp-based orphan elimination techniques [62, 81] have been developed to remove crash and abort orphans from the system. Closely synchronized real-time clocks are required for timely elimination of orphans.

IV.3 Unresolved Challenges

Despite significant prior work, the orphan request problem for DRE systems remains unresolved due to the simultaneous QoS and stringent timeliness requirements.

Limitations of Transactional Approaches. Distributed transactions are unsuitable for real-time systems because they cause significant messaging overhead in the fault-free execution of the critical path: (1) A server must initiate a transaction by sending a *create* message to the transaction manager; (2) Every object that participates in the transaction must register itself with the transaction manager using a *join* message; (3) Non-deterministic components must transfer undo information to their replicas to either *abort* or *commit* the subtransactions in case of a failure; (4) The server must finish the transaction using a *commit* message; (5) While finishing a transaction, the transaction manager initiates a two phase commit (2PC) protocol that sends *prepare* messages to all the transaction participants in the first phase, which if acknowledged positively, sends *commit* messages to all the participants in the second phase; (6) Each participant object sends its vote to the replica before sending it to the transaction manager; Finally, (7) the reply is sent to the client only when the transaction manager indicates successful completion of the 2PC to the initiating server.

While the above steps are needed in every fault-free execution, in case of a failure, not only the orphans must be eliminated by sending the *abort* messages (or compensating transactions) to every orphan, all the above steps must be repeated to re-execute the aborted subtransactions. Note that with the increase in the tiers in the system, the number of orphans that must be eliminated increases. Clearly, the client has to block during this time and may miss the deadline during recovery. Empirical evaluations in [69] confirm that response time may suffer up to 200% increase in the failure-free case whereas client-perceived failover delay could vary from 200 to 400 ms. Contemporary real-time systems often have more stringent performance requirements.

Some optimizations are possible in the above protocols. For instance, in the optimistic

approach of [133], 2PC is not required whereas in the pessimistic case, only the second phase of 2PC is sufficient. In [69], there is no need to send undo information to the replicas, instead, votes must be synchronized with the replicas. The transaction manager must be extended to support join messages with *view-id* of the underlying group communication system thereby loosing the transparency of replication. Additionally, all the above approaches also require objects to implement *prepare*, *commit*, and *rollback* methods to participate in the 2PC.

Finally, ensuring end-to-end schedulabilty of the system in the presence of transactions is a complex problem. The number of orphans that must be rolled-back varies depending upon the tier where the failure occurs. Although the number of orphans are bounded by the number of tiers, the transaction services are often not designed with real-time applications in mind.

Limitations of Enforcing Determinism. The approaches based on enforcing determinism target only a small subset of the sources of non-determinism out of many that are possible in contemporary real-time systems. The approach provided by Slember *et al.* [128, 129] cannot be completely automated. Human intervention is needed to address the non-determinism that cannot be automatically rectified. The solutions for deterministic scheduling [18, 65] add significant overhead in the critical path due to the need to communicate the non-deterministic decisions.

Finally, timestamp-based orphan elimination techniques [62, 81] incur additional messaging overhead of periodic system-wide *refresh*. This overhead is similar to the periodic garbage collection phase in [133].

IV.4 System and Fault Models

System Model. We consider systems in which applications are composed of multiple tiers (*n-tier*) of components/processes that communicate over a network of computing nodes. All or a (non-null) subset of the components may be non-deterministic and may maintain

volatile state across mulitple client invocations (session). The services in the system are invoked by clients periodically via remote operation requests. Every service has a soft real-time deadline, which if missed, reduces the value to the client gradually down to zero. If a deadline is missed (say, due to failures), earlier completion of the request has higher value to the client than later completion. More formally, the nested invocation of components can be understood as end-to-end task chains [82].

One particular way of realizing end-to-end task chains is the *operational strings* [77] model. The operational string model is a form of component-based multi-tier distributed computing that is focused on end-to-end Quality-of-Service (QoS). To satisfy the end-to-end response time, the end-to-end schedulability [134] of computing resources is ensured based on the deployment of components.

We assume that we have access to the deployment and composition information of the components. Deployment and composition information is readily available in component technologies (*e.g.* CORBA Component Model (CCM) [106]) that make use of standards-based meta-data (*e.g., XML descriptors*) to describe the structure of the system. Standard-based deployment and configuration [100] services are used to deploy the components and their replicas, if any. The meta-data becomes the foci of configuring functionality, deployment, and QoS of an operational string. In Section IV.5 we show how group-failover is applicable for not only orphan elimination in operational strings, but also other higher-level availability requirements.

Fault Model. Processors and processes may experience fail-stop [118] failures. Passive replication [23] (primary-backup) approach is used for high-availability and roll-forward recovery because it tolerates non-determinism better than active replication and consumes less resources. The state updates in the primary are *transmitted* to the backups upon completion of each request. However, the state updates are *absorbed* in the backups as dictated by the group-failover protocol described in Section IV.5. If a replica crashes and restarts, it joins the group of backups.

We assume that networks provide bounded communication latency and do not fail or partition. This assumption is reasonable for many soft real-time systems, such as SCADA systems, where nodes are connected by highly redundant high-speed networks. We further assume that due to the synchronous environment, perfect failure detection in the sense of [25] is possible. It bounds the delay for a failure detection and eliminates false suspicions. A minimum of f + 1 replicas will be required in a given tier to tolerate f replica failures. We use an implementation of failure detectors based on per-node daemon and periodic *heart-beat* beacon.

IV.5 The Architecture of the Group-failover Protocol

In this section we present the group-failover protocol that provides exactly-once semantics to the clients while rectifying orphan components in case of the failures of non-deterministic server components. During a client session, non-determinism of one or more components in the nested invocation causes the state updates in the components unique to that particular execution. These state updates become orphan as a consequence of the failure involving a non-deterministic computation. Note that, in multi-tier systems, the orphan components often form a group. For instance, consider a nested invocation among stateful components A, B, C, and D as shown in Figure 20. Just before the non-deterministic component A returns the reply to the client, a failure in A renders components B, C, and D orphan, forming a group.

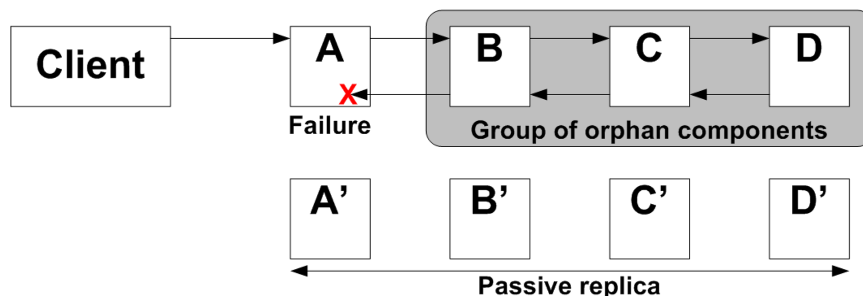

Figure 20: A group of orphan components

To ensure exactly-once semantics, three things must take place as part of the recovery:

1. **Transparent failover.** To ensure forward recovery, the client must transparently failover to the replica of component A (A') and must quickly resume execution of the original request to meet the deadline.

2. **Eliminate orphans.** The orphan state (and any unfinished orphan computations) must be eliminated to avoid inconsistencies in the system and conserve resources. In Figure 20 the orphan state is bounded in the group of primary components B, C and D.

3. **Ensure state consistency.** Exactly-once semantics require that the state of all the components that participate in the reexecution of the client's request (*e.g.,* A', B', C', and D') must be the same as the state of the respective primary components before the execution of the request that caused the failure. Ensuring state consistency is challenging due to lack of transactional semantics for components.

IV.5.1 Transparent failover

Several existing techniques can be employed to achieve transparent failover of the client. Like most contemporary middleware, CORBA supports a standard client-side request interceptor, which enables interception of the call-path at the client-side. Protocol-specific code can be executed in response to various events such as remote function call return or exceptional return (*e.g.,* standard CORBA exceptions such as OBJECT_NOT_EXIST, COMM_FAILURE). The code may also invoke other remote services to obtain a failover replica reference. A standard CORBA exception called LOCATION_FORWARD is raised locally, which then redirects the client to the failover target replica.

Techniques based on Aspect-oriented programming (AOP) are presented in Chapter III to achieve transparent client failover. A protocol-specific advice is weaved *around* client-side stubs, which masks the server failures from the client. An advantage of AOP-based

techniques over the client-side request interceptor is that the advice is invoked only while communicating with a remote object implementing a specific interface. The client-side request interceptors suffer from higher overhead because they are invoked for all the outgoing requests, which may be undesirable.

We use client-side request interceptors because they are sufficient for the group-failover protocol and can be portably implemented for standard compliant CORBA clients. Note that programmers do not need to implement the interceptors. Only requirement is to integrate them in the client build process.

IV.5.2 Eliminating orphans

Lack of well-defined transactional semantics prevents us from rolling the orphan components back to the state before the non-deterministic execution. Orphan state (and computations), however, must be eliminated from the system to prevent inconsistencies and to conserve resources. As noted earlier, the orphan state is bounded inside a group of orphan components. This group of orphan components is simply *discarded* to save time during recovery.

The components can be discarded by invoking life-cycle operations on them. Component technologies often require programmers to implement component life-cycle operations, which are invoked during system deployment, configuration, and startup. For instance, *activate* and *passivate* are two life-cycle operations supported by all CCM [106] session components. Passivating a component discards all its application-specific state as well as middleware state (it is no longer remotely addressable and can no longer initiate remote invocations).

Group-failover protocol uses passivation as the way of eliminating orphan state from the system, which depends on three steps. First, failure must be accurately detected. Second, the group of orphan components must be identified. Finally, the server processes hosting the orphan components must be notified to passivate the select hosted components.

IV.5.2.1 Accurate Failure Detection

To detect process or processor failures and to initiate recovery actions, we use a monitoring infrastructure consisting of dedicated *host-monitor* daemons in every node where the components are deployed. The daemons send a periodic heart-beat to a central *replication manager* (RM) process. In case of a server process failure, the daemons notify the RM about the failure whereas processor failures can be accurately detected by RM when periodic heart-beat from a host-monitor ceases. Upon detection of a failure, RM initiates the process of identifying orphans, which is described next.

IV.5.2.2 Identifying Orphans

Orphans are components to which a non-deterministic component has directly or indirectly communicated before its failure. The number of orphan components varies depending upon where the non-deterministic component lies in the nested invocation and how many components have executed non-deterministic computations in response to the client's request. As mentioned in Section IV.4, we assume that the RM has access to the system composition meta-data, which is cached in RM during deployment of the server components. Composition meta-data describe static connections between the provided and required interfaces implemented by the components. We assume that no additional connections are established dynamically. As a result, RM is aware of the complete topology (directed graph) of the components in the system. Using the meta-data and the failure notifications from the host-monitor daemons, RM can determine orphan components after the failure. For instance, if host-monitor informs RM that component A in Figure 20 has failed, the RM can determine that components B, C, and D are orphans and must be eliminated.

The lack of the run-time information about the stage of a nested invocation limits the accuracy of our static technique of identifying orphans. Due to the inherent asynchronous nature of failures, RM may falsely identify some components as orphans. For instance, if component A fails before invoking nested request to component B, then components B, C,

and D do not become orphan. In general, it is impossible to determine the stage of nested invocation without adding significant overhead[1] in the critical path.

In response to this inherent difficulty, we propose two strategies to identify the span of orphan components.

(A) The entire operational string. This strategy is the most pessimistic of all and designates the entire connected group of (server-side) components as orphan in the case of a failure. The strategy does not need component-specific knowledge of whether it is deterministic or non-deterministic. Pessimistically, it considers the entire group of components as an atomic *failover unit* of non-deterministic components. If any one of them fails, the whole unit is considered as failed.

These failure atomicity semantics are desirable in certain systems based on the operational strings model that employ N-version programming to achieve reliability though diversity. N-version programming [8] is an effective technique to avoid *Bohr-bugs*, which are predictably repeated when the same set of conditions reappear in the system. To avoid Bohr-bugs, operational strings often use dissimilar replicas of the fault-tolerant multi-tier applications. The replicas could be dissimilar in various ways, such as structure, implementation, deployment, resource and QoS requirements, end-to-end deadlines, and priorities. Due to their differences, such *non-isomorphic* operational strings often need granularity of failover larger than a single component, which can be provided using failover units comprising multiple components.

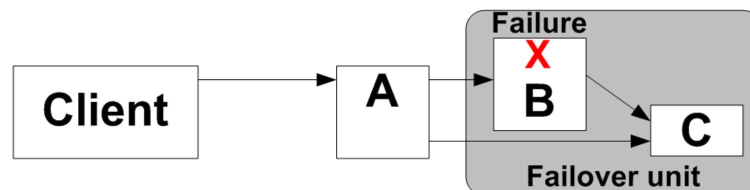

Figure 21: A failover unit spanning two components (B and C).

[1] Transaction allow us to achieve precisely this capability at the cost of significant overhead.

(B) Dataflow-aware component grouping. Accurate meta-information about system composition, such as dataflow and component behavior (deterministic/non-deterministic) could be used to optimize orphan elimination. For instance, consider Figure 21, which shows dataflow between components A, B, and C while component B is non-deterministic. Failure of component B may make component C an orphan but not component A. Therefore, the span of the failover unit consists of components B and C. The static meta-information can be obtained from annotated system composition models. Our earlier work [142] shows a technique based on domain-specific modeling languages to allow system developers to specify component properties using intuitive annotations. Ad-hoc grouping of components is supported using *failover units* (FOU). Such application-specific meta-information is embedded in standards-based deployment meta-data, which is used by RM to eliminate orphans at run-time.

IV.5.2.3 Passivating Orphan Components

Using the static meta-data RM can communicate with the host server processes to passivate the *suspected* orphan components and thereby eliminating them from the system. Passivated components can be activated again and may join the group of replicas to receive state updates from the then-active primary. A subsequent activation phase after recovery is conceivable to maintain the desired level of replication in the system.

IV.5.3 Ensuring state consistency

In the group-failover protocol, client's original request is re-executed by allowing the client to failover on a group of replica components, which must have the state client expects. As per the exactly-once semantics, the client expects the state updates from its previous successful execution to be available at the replicas where the session continues after the failure. The replica state synchronization must be managed carefully so that the state updates during the entire nested invocation ensure exactly-once semantics.

To ensure state consistency, we present two significant variations of the state synchronization strategies: *eager* and *lag-by-one*. The eager state synchronization strategy has overhead during fault-free execution but no overhead during failure recovery. On the other hand, lag-by-one state synchronization strategy has no overhead during fault-free execution but recovery takes longer than the eager strategy.

IV.5.3.1 The eager state synchronization strategy

In the eager strategy, the state of all the nested components is synchronized with their respective replicas only after the client-side interceptor has received the result of the non-deterministic computation. The state that builds up at the server-side during execution of the client's request is a *potential orphan* state because any subsequent failure in the *upstream* component renders these state changes orphaned. Therefore, it is only after the client-side interceptor has received the reply, the server-side state changes can be made permanent in the system.

In case of a failure, the orphan state is eliminated as described in Section IV.5.2. Therefore, a client's request must be reexecuted on replica components with the state of the last successful execution. Eager state synchronization strategy ensures that the state in the replicas is indeed from the last successful execution. Replication manager (RM) in Figure 22 has the responsibility of maintaining state consistency. State from the last successful execution can be maintained at the replicas if state synchronization completes in an atomic way. Either all the primary components synchronize their state with respective replicas or none at all. This atomicity is guaranteed by a variation of the two-phase commit (2PC) protocol.

Figure 22 shows how the eager strategy works. A client is requesting a service from a non-deterministic set of components A, B, and C. Consider that the kth execution has completed and the reply has arrived at the client-side. Although the execution has completed, replica components (A', B', and C') do not yet have the state updates resulting from

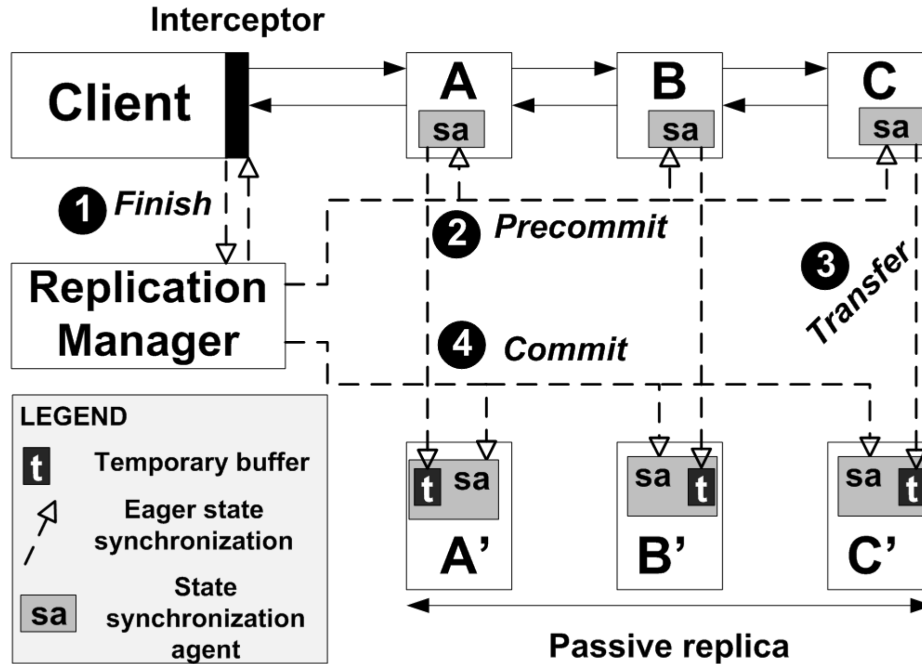

Figure 22: Eager state synchronization strategy

the kth execution. The client-side request interceptor detects successful execution and before returning to the client application, initiates the eager state synchronization strategy by sending a *finish* message to the RM.

In response to the *finish* message, RM initiates the two-phase process to synchronize the state atomically. It sends a *precommit* message to each *state synchronization agent* (sa) located in the server processes hosting the primary components. The intent of the precommit message is to persist the application-specific state by sending it to the replicas. The state is retrieved by using the predefined interface (*e.g., get_state*) implemented by the primary components. The state synchronization agents collocated with the primary components *transfer* the state to the state synchronization agents in the process hosting the replica components. The state is maintained in a temporary buffer untill the second phase.

RM initiates the second phase only if the first phase completes successfully. In the second phase, RM sends *commit* messages to the state synchronization agents collocated with the replicas. In response to the commit message, the state in the temporary buffers

is pushed in the replica components by means of a predefined interface such as *set_state*. When both the phases complete, RM returns an error value indicating success to the client-side interceptor allowing the client to process the reply.

If some primary component is unable to send the state to its replicas, RM detects the failure during the first phase, skips the second phase, eliminates the orphans as described in Section IV.5.2, and returns an error value indicating failure to the client-side interceptor. The client-side interceptor does a transparent failover to the replica group and reinvokes the same request to ensure that there is exactly-one non-deterministic execution of the request. The resulting state is persisted in the system in the form of volatile memory of the replica components. Note that a failure of the second phase (say, due to a failure of a replica) does not affect state consistency because primary components can serve the subsequent requests. Failed replicas may be restarted to maintain a desired level of replication degree.

IV.5.3.2 The lag-by-one state synchronization strategy

In this section we present an optimization of the eager strategy, which has no overhead during the fault-free execution but incurs recovery messaging overhead in the failure scenario. We call this technique *lag-by-one* state synchronization strategy because the state in the replicas is always lagging by exactly one state update than that of the primary components. A schematic of the lag-by-one strategy is shown in Figure 23.

The lag-by-one strategy eliminates the need for explicit two phases of state synchronization found in the eager approach. Instead, the potential orphan state is transferred to the temporary buffers of the state synchronization agents collocated with the replica components immediately after the completion of a request at every primary nested component. This is shown in Figure 23 by double-dashed lines. This state synchronization is initiated lazily after sending the reply back to the calling non-deterministic component eliminating overhead in the critical path. Note that the transferred state is considered a *potential orphan* until the client receives the reply. The transferred state is maintained in the temporary

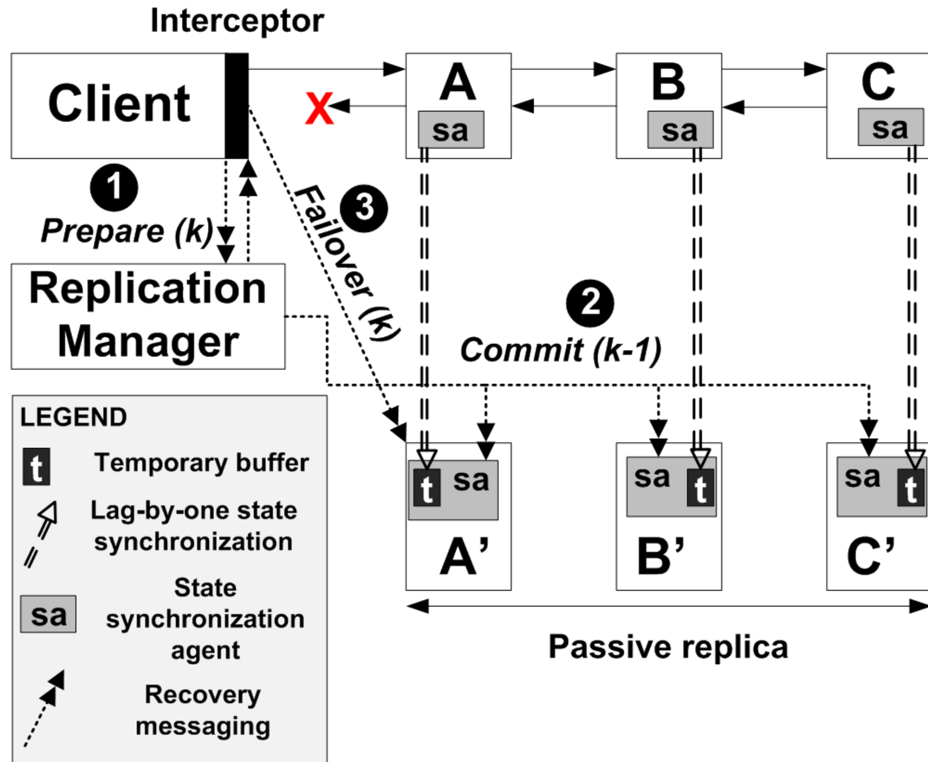

Figure 23: Lag-by-one state synchronization strategy

buffer at the replica side. Unlike the eager approach, however, no *finish* message is sent to the RM at the end of successful invocation. Instead, invocation and execution of the subsequent request from the client is interpreted as the successful completion of the earlier request. As a result, when the kth state update arrives at the replica state synchronization agent, the $(k-1)^{th}$ state update (stored in the temporary buffer) is pushed into the replica component. The explicit *commit* phase of the eager strategy is replaced by the implicit arrival of new potential orphan state from the primary components.

In case of a failure, the replica components are not ready for an instantaneous client failover because the state that client expects lies in the temporary buffers of the state synchronization agents. In other words, when the kth invocation completes successfully, the temporary buffers at the replica side contain the corresponding potential orphan state but the replicas themselves have the state at the end of the $(k-1)^{th}$ execution. If the kth request

fails due to a failure of one of the primary components, the replica components must be *prepared* to accept the client's reinvocation.

When the system is configured to follow the lag-by-one strategy, the client-side interceptor and the RM must also be configured likewise. Consider the client is invoking the kth request. Upon detection of failure the client-side interceptor sends a *prepare(k)* message to the RM requesting it to prepare the replicas for the kth invocation. In response to the prepare message, RM eliminates the orphans because prepare message is possible only after a failure. Subsequently, it sends *commit(k-1)* message to the state synchronization agents collocated in the replicas, which push the state in the temporary buffers into the replicas. RM returns to the client-side interceptor indicating success by an appropriate return value. The client-side interceptor does transparent failover to the replica group and reinvokes the kth request.

IV.5.3.3 Middleware implementation of the two strategies

We extended the client-side interceptor logic and the RM from our prior work on FLARe [12]. In particular, the RM is now strategizable. When using the eager strategy, the RM is equipped with the messaging logic for the two phases. When using the lag-by-one strategy, the RM gets involved only during failure recovery and orphan elimination. Similarly, the client-side interceptor is also strategizable. For the eager state synchronization strategy, it is refined to include the additional logic for ensuring atomicity. In the lag-by-one strategy, the client interceptor performs additional work during failure recovery.

The state synchronization agents are leveraged from our prior work on CORFU [153]. For this work, we introduce a new temporary buffer with the agents. Additionally we introduce new logic into the agents on when to permanently accept the state into the backup replicas.

Our implementation of the eager synchronization strategy uses CORBA asynchronous messaging interface (AMI) in both phases to exploit parallelism during state transfer (Step

#2 and #4 in Figure 22). RM uses AMI to send the *precommit* to all the primary state synchronization agents in parallel and waits for all of them to complete state transfer in parallel. The second phase also uses AMI. The use of AMI allows the eager state synchronization strategy to be (nearly) insensitive to the number of components involved in the group. For lag-by-one, AMI is used to lazily transfer state (see double dashed lines in Figure 23) thereby eliminating the overhead of state transfer in the critical path.

Finally, all the deployment and configuration activities, including activating and passivating the components is seamlessly handled by the existing CIAO middleware.

Implementing new capabilities and adding extensions to entities of the existing middleware code base without making invasive changes is easily feasible through the use of elegant design patterns, such as Strategy and Factory.

IV.6 Evaluating the Merits of the Group-failover Protocol

We implemented the eager and lag-by-one variations of the group-failover protocol using the open-source Component Integrated ACE ORB (CIAO) [64] middleware, which is an implementation of Lightweight CORBA Component Model [95] (LwCCM). We leveraged a number of infrastructure elements from our prior work on FLARe and CORFU, however, a number of extensions were implemented for this research. The state synchronization agent was refined to include a temporary buffer, while the replication manager design was refined to support the two state synchronization strategies. Host monitors were used directly from prior work. All these elements were implemented as CORBA objects and integrated into the deployment process of components.

We empirically evaluated our implementation of the group-failover protocol at ISISlab (www.isislab.vanderbilt.edu) on a testbed of up to 6 blades. Each blade has two 2.8 GHz CPUs, 1GB memory and they are connected by a Gigabit LAN.

Methodology and Rationale. We evaluated the overhead and the client-perceived failover latency in fault-free and failure scenarios, respectively. In every experiment we varied the

number of nested components from 2 to 5 to show that both state synchronization strategies have bounded overhead independent of the task chain size. Dummy computations, calibrated to consume 200ms (+/- 3%) of CPU time, were performed on every component in response to the client request. As a result, the server-side execution time varies from 400ms to 1000ms (+/- 3%) with the number of components varied from 2 to 5. In both fault-free and failure scenarios, we measured the client-perceived execution time, which includes the actual server-side execution time, overhead of the state synchronization phases (if any), and the communication latency. Per node host-monitors and a single instance of replication manager were also deployed.[2]

As discussed earlier, we used asynchronous method invocation (AMI) to exploit concurrency during state transfer and the various phases of the state synchronization strategies. With every asynchronous call a callback handler is registered, which is invoked upon completion of the remote request. It is also possible to abort waiting for a callback based on a timeout.

IV.6.1 Overhead measurements in fault-free scenarios

Nesting level	2	3	4	5
Actual execution time (ms)	412.9	624.9	831.2	1040.1
Client-perceived execution time (ms)	423.3	637	843.1	1056.6
Overhead (ms)	8.2	12.1	11.9	10.6
Overhead without AMI (ms)	13	23	26.1	32.7

Table 6: Overhead of the eager strategy (fault-free) (jitter +/- 3%)

Table 6 shows the overhead of the eager strategy during fault-free executions, which is

[2]Fault tolerance of the replication manager can also be achieved through replication.

Nesting level	2	3	4	5
Client-perceived execution time (ms)	419.1	624.9	840.9	1048.3
Actual execution time (ms)	411.5	620.1	831.2	1040.1
Difference (ms)	7.6	4.8	9.7	8.2

Table 7: Difference in the actual and perceived execution times in the lag-by-one strategy (fault-free) (jitter +/- 3%)

computed using a high resolution timer around the *finish* method invocation from the client-side request interceptor to the replication manager. The overhead was bounded between 8.2 ms and 12.1 ms (with low jitter) for the two phases of the eager strategy. The client-perceived execution time is more than the summation of actual execution time and the overhead of the phases due to the unavoidable communication latency. To better understand the benefits of AMI, we implemented the two phases of eager strategy without AMI. From the results in Table 6, it is evident that without AMI the overhead is sensitive to the number of nested components.

Table 7 shows the difference between the actual execution time and the client-perceived execution time for the lag-by-one strategy during fault-free executions. Due to the lack of any explicit phases in the lag-by-one strategy, we did not use high resolution timers. The difference, computed by subtracting the actual execution time from the client-perceived execution time, includes a small overhead of initiating the lazy state transfer at every component and the unavoidable communication latency.

These results indicate that the overhead of both the strategies is extremely low compared to the protocols that integrate transactions and replication, and that the additional work performed is bounded irrespective of the size of the task chain.

IV.6.2 Client-perceived failover latency in failure scenarios

The objective of this experiment is to find the failover latency that a client will experience when the server component that the client is communicating with (*i.e.*, head of the chain) fails just before returning the reply to the client. This is the worst possible case for the client in terms of meeting its deadline because the client needs to wait for the entire execution not only on the primary components but also on the backup replica components.

Nesting level	2	3	4	5
Primary execution time (ms)	411.5	620.1	831.2	1040.1
Replica execution time (ms)	419.1	623.3	840.6	1060.2
Client-perceived failover latency (ms)	856.8	1310.2	1710.1	2134.5
Lag-by-one's prepare phase latency (ms)	4.1	7.2	6.2	5.3
Eager protocol's failover latency (ms)	0	0	0	0

Table 8: Client-perceived failover latency of the state synchronization strategies

Table 8 shows the client-perceived failover latency for different nesting levels (2 to 5). We observe that the execution times on the replica components are larger than that of the primary components because of the higher cache misses during the first execution after failover. We also measured the overhead of the *prepare* phase of the lag-by-one protocol using a high resolution timer in the client-side interceptor. During the prepare phase, the replication manager exploits concurrency by sending the *commit* messages using AMI. As a result, we see that the overhead of the prepare phase is not dependent on the nesting level. These results indicate that the performance of group-failover protocols is acceptable in failure scenarios, and that the latencies are bounded irrespective of the task chain size (*i.e.*, nesting level).

CHAPTER V

RESOLVING SOLUTION DOMAIN CHALLENGES: OBJECT STRUCTURE TRAVERSAL

Model-driven engineering (MDE) has been used extensively in this dissertation to address the systemic issues of distributed real-time and embedded systems. The discipline of MDE is still maturing and limitations of the contemporary tools and techniques for MDE were identified during the course of the research work on model-driven fault-tolerance provisioning described in chapters II to IV. This chapter is dedicated to the novel solutions addressing the deficiencies of the existing tools for a key step in MDE: object structure traversal.

V.1 Introduction

Compound data processing is commonly required in applications, such as program transformation, XML document processing, model interpretation and transformation. The data to be processed is often represented in memory as a heterogeneously typed hierarchical object structure in the form of either a tree (*e.g.,* XML document) or a graph (*e.g.,* models). The necessary type information that governs such object structures is encoded in a schema. For example, XML schema [145] specifications are used to capture the vocabulary of an XML document. Similarly, metamodels [137] serve as schema for domain-specific models. We categorize such applications as *schema-first* applications because at the core of their development lie one or more schemas.

The most widespread technique in contemporary object-oriented languages to organize these schema-first applications is a combination of the Composite and Visitor [48] design patterns where the composites represent the object structure and visitors traverse it. Along with traversals, *iteration*, *selection*, *accumulation*, *sorting*, and *transformation* are other

common operations performed on these object structures. In this chapter, we deal with the most general form of object structures, *i.e.,* object graphs, unless stated otherwise.

Existing techniques [39] to write traversals often use language-specific data binding [70, 83] tools to generate an object-oriented API for manipulating the models. Such object structure traversals are often verbose due to schema-specificity of the API. Intuitive and succinct traversal notations, such as XPath [154], can not be used without sacrificing the type-safety of the generated object-oriented API. Traversal programming idioms (*e.g.,* XPath child/parent axes and wildcards) are not natively supported in general-purpose programming languages.

Moreover, in many programming paradigms, object structure traversals are often implemented in a way such that the traversal logic and type-specific computations are entangled. Tangling in the functional programming paradigm has been identified in [75]. In object-oriented programming, when different traversals are needed for different visitors, the responsibility of traversal is imposed on the visitor class coupled with the type-specific computations. Such a tight coupling of traversal and type-specific computations adversely affects the reusability of the visitors and traversals equally.

V.2 Related Research

To overcome the pitfalls of the Visitor [48] pattern, domain-specific languages (DSL) that are specialized for the traversals of object structures have been proposed [51, 107]. These DSLs separate traversals from the type-specific computations using *external* representations of traversal rules and use a separate code generator to transform these rules into a conventional imperative language program. This two step process of obtaining executable traversals from external traversal specifications, however, has not enjoyed widespread use. Among the most important reasons [17, 29, 84, 121] hindering its adoption are (1) high upfront cost of the language and tool development, (2) their extension and maintenance overhead, and (3) the difficulty in integrating them with existing code-bases. For example,

development of language tools such as a code generator requires the development of at least a lexical analyzer, parser, back-end code synthesizer and a pretty printer. Moreover, Mernik et al. [84] claim that language extension is hard to realize because most language processors are not designed with extension in mind. Finally, smooth integration with existing code-bases requires an ability of not only choosing a subset of available features but also incremental addition of those features in the existing code-base. External traversal DSLs, however, lack support for incremental addition as they tend to generate code in bulk rather than small segments that can be integrated at a finer granularity. Therefore, programmers often face an *all-or-nothing* predicament, which limits their adoption. *Pure embedding* is a promising approach to address these limitations of external DSLs.

Other prominent research on traversal DSLs have focused on Strategic Programming (SP) [74, 75, 148] and Adaptive Programming (AP) [79, 80] paradigms, which support advanced separation of traversal concerns from type-specific actions. SP is a language-independent generic programming technique that provides a design method for programmer-definable, reusable, generic traversal schemes. AP, on the other hand, uses static meta-information to optimize traversals and check their conformance with the schema. Both the paradigms allow *structure-shy* programming to support traversal specifications that are loosely coupled to the object structure. We believe that the benefits of SP and AP are critical to the success of a traversal DSL. Therefore, an approach that combines them in the context of a pure embedded DSL while addressing the *integration challenge* will have the highest potential for widespread adoption.

A more through survey of related research is presented in Section V.8.

V.3 Unresolved Challenges and Overview of the Solution Approach

To address the limitations in the current state-of-the-art, this chapter presents a multi-paradigm programming [27, 150] approach to develop a domain-specific language (DSL) for specifying traversals over object graphs governed by a schema. An expression-based [29]

pure embedded (in C++) DSL called Language for Embedded quEry and traverSAl (LEESA) is presented. LEESA combines generic programming, static metaprogramming [3], generative programming [28], and strategic programming [75, 147] paradigms together with C++ operator overloading to provide an intuitive notation for writing type-safe traversals.

- It provides a notation for traversal along several object structure axes, such as *child*, *parent*, *sibling*, *descendant*, and *ancestor*, which are akin to the XML programming idioms in XPath [154] – an XML query language. LEESA additionally allows composition of type-specific behavior over the axes-oriented traversals without tangling them together.

- It is a novel incarnation of SP using C++ templates, which provides a combinator style to develop programmer-definable, reusable, generic traversals akin to the classic SP language Stratego [147]. The novelty of LEESA's incarnation of SP stems from its use of static meta-information to implement not only the regular behavior of (some) primitive SP combinators but also their customizations to prevent traversals into unnecessary substructures. As a result, efficient *descendant* axis traversals are possible while simultaneously maintaining the schema-conformance aspect of AP.

- One of the most vexing issues in embedded implementations of DSLs is the lack of mechanisms for intuitive *domain-specific error reporting*. LEESA addresses this issue by combining C++ template metaprogramming [3] with concept checking [57, 126] in novel ways to provide intuitive error messages in terms of *concept violations* when incompatible traversals are composed at compile-time.

- Finally, its embedded approach allows incremental integration of the above capabilities into the existing code-base. During our evaluation of LEESA's capabilities, small segments of verbose traversal code were replaced by succinct LEESA expressions in a step by step fashion. We are not aware of any external C++ code generator that allows integration at comparable granularity and ease.

V.4 Language for Embedded Query and Traversal (LEESA)

In this section we formally describe the syntax of LEESA and its underlying semantic model in terms of axes traversals. To better explain its syntax and semantics, we use a running example of a domain-specific modeling language for a hierarchical finite state machine (HFSM) described next.

V.4.1 Hierarchical Finite State Machine (HFSM) Language: A Running Example

Figure 24 shows a metamodel of a HFSM language using a UML-like notation. Our HFSM metamodel consists of `StateMachines` with zero or more `States` having directional edges between them called `Transitions`. States can be marked as a "start state" using a boolean attribute. States may contain other states, transitions, and optionally a `Time` element. A `Transition` represents an association between two states, where the source state is in the `srcTransition` role and the destination state is in the `dstTransition` role with respect to a `Transition` as shown in Figure 24. `Time` is an indivisible modeling element (hence the stereotype ≪Atom≫), which represents a user-definable delay in seconds. If it is absent, a default delay of 1 second is assumed. `Delay` represents the composition role of a `Time` object within a `State` object. All other composition relationships do not have any user-defined composition roles but rather a default role is assumed. The `Root` is a singleton that represents the root level model.

To manipulate the instances of the HFSM language, C++ language bindings were obtained using a code generator. The generated code consists of five C++ classes: `Root`, `StateMachine`, `State`, `Transition`, and `Time` that capture the vocabulary and the relationships shown in the above metamodel. We use these classes throughout the chapter to specify traversals listings.

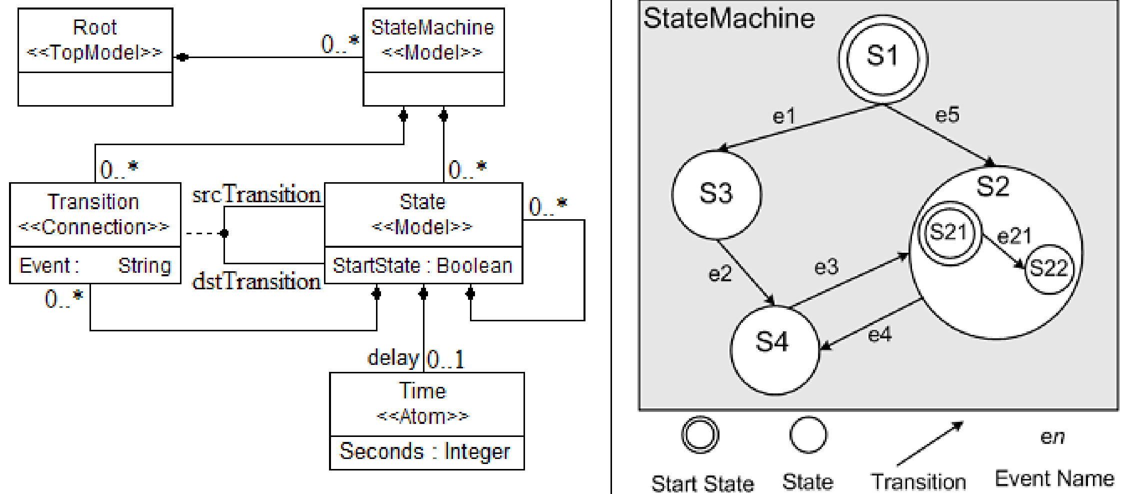

Figure 24: Meta-model of Hierarchical Finite State Machine (HFSM) language (left) and a simple HFSM model (right)

V.4.2 An Axes-Oriented Notation for Object Structure Traversal

Designing an intuitive domain-specific notation for a DSL is central to achieving productivity improvements as domain-specific notations are closer to the problem domain in question than the notation offered by general-purpose programming languages (GPLs). The notation should be able to express the key abstractions and operations in the domain succinctly so that the DSL programs become more readable and maintainable than the programs written in GPLs. For object structure traversal, the key abstractions are the objects and their typed collections while the basic operations performed are the navigation of associations and execution of type-specific actions.

When designing a notation for an embedded DSL, an important constraint imposed by the host language is to remain within the limits of the programmer-definable operator syntax offered by the host language. Quite often, trade-offs must be made to seek a balance between the expressiveness of the embedded notation against what is possible in a host language.

Listing 5 shows LEESA's syntax represented in the form of a grammar. *Statement* marks the beginning of a LEESA expression, which usually contains a series of *type*s, *action*s, and visitor objects separated by *operator*s. The first *type* in a LEESA statement

```
Statement   : Type { (Operator    Type)  |  (LRShift  visitor-object) |
                     (LRShift   Action)  |  (>>  Members)             |
                     (>> | >>=  Association)  }+
Type        : class-name '('  ')'
Operator    : LRShift |  >>=  |  <<=
LRShift     : >>  |  <<
Action      : "Select" | "Sort" | "Unique" | "ForEach" | (and more ...)
Association : "Association" '(' class-name :: role-name ')'
Members     : "MembersOf" '(' Type {  ','  Statement }+ ')'
```

Listing 5: Grammar of LEESA expressions

determines the type of object where the traversal should begin. The four operators ($\gg$, $\ll$, $\gg$=, $\ll$=) are used to choose between *children* and *parent* axes and variations thereof. This traversal notation of LEESA resembles XPath's query syntax, however, unlike XPath, the LEESA expressions can be decorated with visitor objects, which modularize the type-specific actions away from the traversals.

The *association* production in Listing 5 represents traversal along user-defined association roles (captured in the metamodel) whereas *members* represent traversal along *sibling* axis. Actions are generic functions used to process the results of intermediate traversals. The parameters accepted by these actions, which are implemented as C++ function templates, are not shown. Instead, only the string literals sufficient for illustration are shown. Finally, instances of programmer-defined visitor classes can be added in the place of *visitor-object* that simply dispatch the type-specific actions. It conveniently allows accumulation of information during traversal without tangling the type-specific computations and the traversal specifications.

We now present concrete examples of LEESA expressions with their semantics in the context of the HFSM language case-study in Section V.4.1.

Axes	LEESA expressions and their semantics
(A) Child (breadth first)	**Root() ≫ StateMachine() ≫ v ≫ State() ≫ v** Visit *all* state machines followed by all their immediate children states.
(B) Child (depth first)	**Root() ≫= StateMachine() ≫ v ≫= State() ≫ v** Visit *a* state machine and all its immediate children states. Repeat this for the remaining state machines.
(C) Parent (breadth first)	**Time() ≪ v ≪ State() ≪ v ≪ StateMachine() ≪ v** Visit a given set of time objects followed by their immediate parent states followed by their immediate parent state machines.
(D) Parent (depth first)	**Time() ≪ v ≪= State() ≪ v ≪= StateMachine() ≪ v** For a given set of time objects, visit *a* Time object followed by visit its parent state followed by visit its parent state machine. Repeat this for the remaining time objects.

Table 9: Child and parent axes traversal using LEESA (v can be replaced by an instance of a programmer-defined visitor class.)

V.4.2.1 Child and Parent Axes.

Child and parent axes traversals are one of the most common operations performed on object structures. LEESA provides a succinct and expressive syntax in terms of "≫" and "≪" operators for child and parent axes traversals, respectively. Two variations, *breadth-first* and *depth-first*, of both the axes are also supported. Presence of the "=" operator after the above operators turns a breadth-first strategy into a depth-first.[1] Table 9 shows four LEESA traversal expressions using child and parent axes notations. Figure 25 illustrates the graphical outlines corresponding to the examples shown in Table 9.

Breadth-first and depth-first variations of the axes traversal strategies are of particular interest here because of the ease of control over traversal provided by them. The breadth-first strategy, if applied successively (as in examples (a) and (c) in Table 9), visits all the instances of the specified type in a *group* before moving on to the next group of objects along an axis. Essentially, this strategy simulates multiple looping constructs in a sequence. The depth-first strategy, on the other hand, selects a single object of the specified type at a time, descends into it, executes the remaining traversal expression in the context of that single object, and repeats the same with the next object, if any. Therefore, successive

[1] In C++, "≪=" and "≫=" are bitwise shift left & assign and shift right & assign operators, respectively.

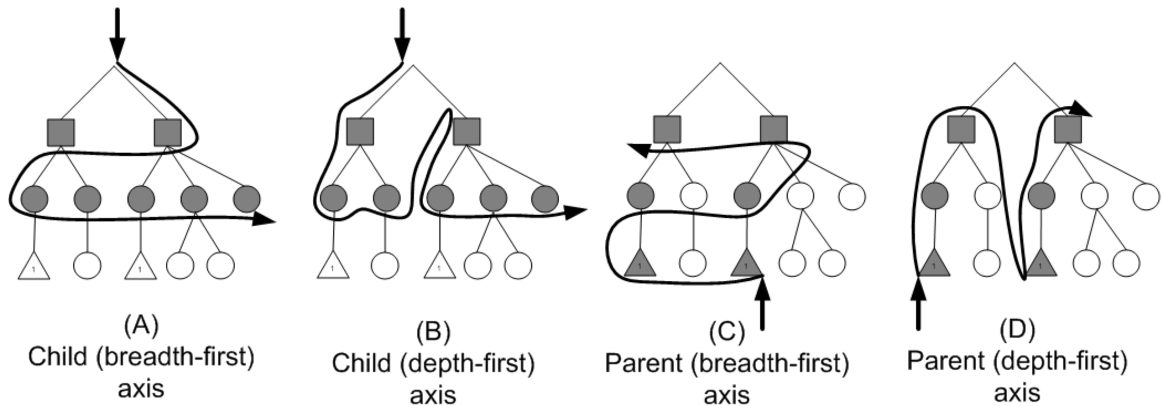

Figure 25: Outlines of child/parent axes traversals (Squares are statemachines, circles are states, triangles are time objects, and shaded shapes are visited.)

application of the depth-first strategy (as in examples (b) and (d) in Table 9), traverses the edges of the object tree unlike the breadth-first strategy. Essentially, the depth-first strategy simulates nested looping constructs.

LEESA uses the Visitor [48] design pattern to organize the type-specific behavior while restricting traversals to the LEESA expressions only. To invoke type-specific computations, LEESA expressions can be decorated with instances of programmer-defined visitor classes as shown in Table 9. If a visitor object `v` is written after type `T`, LEESA invokes `v.Visit(t)` function on every collected object `t` of type `T`. LEESA expressions can be used not only for visitor dispatch but also for obtaining a collection of the objects of the type that appears last in the expression. Such a collection of objects can be processed using conventional C++. For instance, example (a) in Table 9 returns a set of `States` whereas example (c) returns a set of `StateMachines`.

V.4.2.2 Descendant and Ancestor Axes.

LEESA supports *descendant* and *ancestor* axes traversal seamlessly in conjunction with child/parent axes traversals. For instance, Listing 6 shows a LEESA expression to obtain

a set of `Time` objects that are recursively contained inside a `StateMachine`. This expression supports a form of structure-shy traversal in the sense that it does not explicitly specify the intermediate structural elements between the `StateMachine` and `Time`.

```
Root() >> StateMachine() >> DescendantsOf(StateMachine(), Time())
```

Listing 6: A LEESA expression showing descendant axis traversal

Two important issues arise in the design and implementation of the structure-shy traversal support described above. First, how are the objects of the target type located *efficiently* in a hierarchical object structure? and second, at what stage of development is the programmer notified of impossible traversal specifications? In the first case, for instance, it is inefficient to search for objects of the target type in composites that do not contain them. Whereas in the second case, it is erroneous to specify a target type that is not reachable from the start type. Section V.5.3 and Section V.6 present solutions to the efficiency and the error reporting issues, respectively.

V.4.2.3 Sibling Axis.

Composition of multiple types of objects in a composite object is commonly observed in practice. For example, the HFSM language has a composite called `StateMachine` that consists of two types of children that are siblings of each other: `State` and `Transition`. Support for object structure traversal in LEESA would not be complete unless support is provided for visiting multiple types of siblings in a programmer-defined order.

Listing 7 shows an example of how LEESA supports sibling traversal. The sample expression visits all the `States` in a `StateMachine` before all the `Transitions`. The types of visited siblings and their order is programmer-definable. The `MembersOf` notation is designed to improve readability as its first parameter is the common parent type

```
ProgrammerDefinedVisitor v;
Root()>> StateMachine() >> MembersOf(StateMachine(), State()      >> v,
                                                    Transition() >> v)
```

Listing 7: A LEESA expression for traversing siblings: `States` and `Transitions`

(*i.e.,* `StateMachine`) followed by a comma separated list of LEESA subexpressions for visiting the children in the given order. It effectively replaces multiple *for* loops written in a sequence where each for loop corresponds to a type of sibling.

V.4.2.4 Association Axis.

LEESA supports traversals along two different kinds of user-defined associations. First, *named composition roles*, which use user-defined roles while traversing composition instead of the default composition role. For instance, in our HFSM modeling language, `Time` objects are composed using the `delay` composition role inside `States`. Second, *named associations* between different types of objects turn tree-like object structures into graphs. For example, `Transition` is a user-defined association possible between any two `States` in the HFSM language described in Section V.4.1. Moreover, `srcTransition` and `dstTransition` are two possible roles a `State` can be in with respect to a `Transition`.

LEESA provides a notation to traverse an association using the name of the association class (*i.e., class-name* in Listing 5) and the desired role (*i.e., role-name* in Listing 5). Listing 8 shows two independent LEESA expressions that traverse two different user-defined associations. The first expression returns a set of `Time` objects that are composed immediately inside the top-level `States`. The expression traverses the `delay` composition role defined between states and time objects. This feature allows differentiation (and selection) of children objects that are of the same type but associated with their parent with different composition roles.

```
Root()>> StateMachine() >> State() >> Association(State::delay)...(1)
Root()>> StateMachine() >> Transition()
                         >> Association(Transition::dstTransition)...(2)
```

Listing 8: Traversing user-defined associations using LEESA.

The second expression returns all the top-level states that have at least one incoming transition. Such a set can be conceptually visualized as a set of states that are at the *destination* end of a transition. The second expression in Listing 8 up to `Transition()` yields a set of transitions that are the immediate children of `StateMachines`. The remaining expression to the right of it traverses the user-defined association `dstTransition` and returns `States` that are in the *destination* role with respect to every `Transition` in the previously obtained set.

In the above association-based traversals, the operator "≫" does not imply child axis traversal but instead represents continuation of the LEESA expression in a breadth-first manner. As described before, breadth-first strategy simulates loops in sequence. Use of "≫=" turns the breadth-first strategy over association axis into a depth-first strategy, which simulates nested loops. Expressions with associations can also be combined with visitor objects if role-specific actions are to be dispatched.

V.4.3 Programmer-defined Processing of Intermediate Results Using Actions

Writing traversals over object structures often requires processing the intermediate results before the rest of the traversal is executed (*e.g.,* filtering objects that do not satisfy a programmer-defined predicate, or sorting objects using programmer-defined comparison functions). LEESA provides a set of *actions* that process the intermediate results produced by the earlier part of the traversal expression. These actions are in fact higher-order functions that take programmer-defined predicates or comparison functions as parameters and apply them on a collection of objects.

```
int comparator (State, State) { ... } // A C++ comparator function
bool predicate (Time) { ... }         // A C++ predicate function
Root() >> StateMachine() >> State() >> Sort(State(), comparator)
                         >> Time()  >> Select(Time(), predicate)
```

Listing 9: A LEESA expression with actions to process intermediate results

Listing 9 shows a LEESA expression that uses two predefined actions: `Sort` and `Select`. The `Sort` function, as the name suggests, sorts a collection using a programmer-defined comparator. `Select` filters out objects that do not satisfy the programmer-defined predicate. The result of the traversal in Listing 9 is a set of `Time` objects, however, the intermediate results are processed by the actions before traversing composition relationships further. `Sort` and `Select` are examples of higher-order functions that accept conventional functions as parameters as well as stateful objects that behave like functions, commonly known as *functors*.

LEESA supports about a dozen different actions (*e.g.,* Unique, ForEach) and more actions can be defined by the programmers and incorporated into LEESA expressions if needed. The efforts needed to add a new action are proportional to adding a new class template and a global overloaded operator function template.

V.4.4 Generic, Recursive, and Reusable Traversals Using Strategic Programming

Although LEESA's axes traversal operators ($\gg$, $\ll$, $\gg=$, $\ll=$) are reusable for writing traversals across different schemas, they force the programmers to commit to the vocabulary of the schema and therefore the traversal expressions (as whole) cannot be reused. Moreover, LEESA's axes traversal notation discussed so far lacked support for *recursive* traversal, which is important for a wide spectrum of domain-specific modeling languages that support *hierarchical* constructs. For example, our case study of HFSM requires recursive traversal to visit deeply nested states.

A desirable solution should not only support recursive traversals but also enable higher-level reuse of *traversal schemes* while providing complete control over a traversal. Traversal schemes are higher level control patterns (*e.g.,* top-down, bottom-up, depth-first, etc.) for traversal over heterogeneously typed object structures. Strategic Programming (SP) [74, 75, 148] is a well-known generic programming idiom based on programmer-definable (recursive or otherwise) traversal abstractions that allow separation of type-specific actions from reusable traversal schemes. SP also provides a *design method* for developing reusable traversal functionality based on so called *strategies*. Therefore, based on the observation that LEESA shares this goal with that of SP, we adopted the SP design method and created a new incarnation of SP on top of LEESA's axes traversal notation. Next, we describe how LEESA leverages the SP design method to meet its goal of supporting generic, recursive, and reusable traversals. For a detailed description of the foundations of SP, we suggest [74, 75, 148].

Primitive combinators	**Description**
Identity	Returns its input datum without change.
Fail	Always throws an exception indicating a failure.
Sequence<S1,S2>	Invokes strategies S1 and S2 in sequence on its input datum.
Choice<S1,S2>	Invokes strategy S2 on its input datum only if the invocation of S1 fails.
All<S>	Invokes strategy S on all the immediate children of its input datum.
One<S>	Stops invocation of strategy S after its first success on one of the children of its input datum.

Table 10: The set of basic class template combinators

LEESA's incarnation of the SP design method is based on a small set of *combinators* that can be used to construct new combinators from the given ones. By combinators we mean reusable C++ class templates capturing basic functionality that can be composed in different ways to obtain new functionality. The basic combinators supported in LEESA are

summarized in Table 10. This set of combinators is inspired by the *strategy* primitives of the term rewriting language Stratego [147].

```
FullTD<Strategy> = Sequence<Strategy, All<FullTD> >
FullBU<Strategy> = Sequence<All<FullBU>, Strategy>
```

Listing 10: Pseudo-definitions of the class templates of the predefined traversal schemes (Strategy = Any primitive combinator or combination thereof, TD = top-down, BU = bottom-up)

All and One are *one-layer traversal* combinators, which can be used to obtain full traversal control, including recursion. Although none of the basic combinators are recursive, higher-level traversal schemes built using the basic combinators can be recursive. For instance, Listing 10 shows a subset of predefined higher-level traversal schemes in LEESA that are recursive. The (pseudo-) definition of FullTD (full top-down) means that the parameter Strategy is applied at the root of the incoming datum and then it applies itself recursively to all the immediate children of the root, which can be of heterogeneous types. Figure 26 shows a graphical illustration of FullTD and FullBU (full bottom-up) traversal schemes. Section V.5.3 describes the actual C++ implementation of the primitives and the recursive schemes in detail.

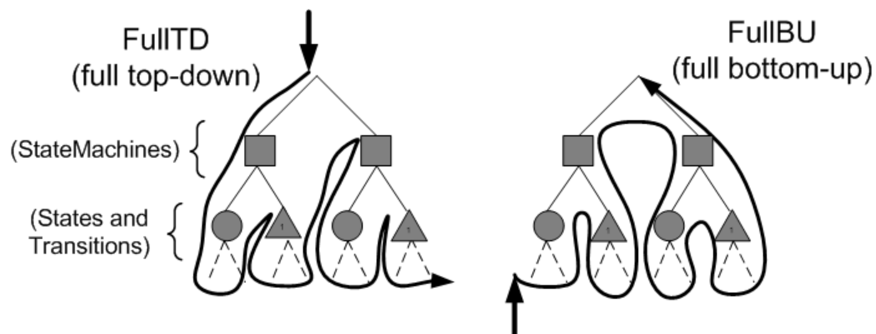

Figure 26: Graphical illustration of FullTD and FullBU traversal schemes. (Squares, circles, and triangles represent objects of different types)

```
Root() >> StateMachine()
       >> FullTD(StateMachine(), VisitStrategy(v), LeaveStrategy(v));
```

Listing 11: Combining axes-oriented traversal, strategic programming, and hierarchical visitor in LEESA. (v can be replaced by a programmer-defined visitor.)

Listing 11 shows how the `FullTD` recursive traversal scheme can be used to perform full top-down traversal starting from a `StateMachine`. Note that the heterogeneously typed substructures (`State`, `Transition`, and `Time`) of the `StateMachine` are not mentioned in the expression. However, they are incorporated in the traversal automatically using the static meta-information in the metamodel. This is achieved by *externalizing* the static meta-information in a form that is understood by the C++ compiler and in turn the LEESA expressions. Later in Section V.5.2 we describe a process of externalizing the static meta-information from the metamodel (schema) and in Section V.5.3 we show how it is used for substructure traversal.

Finally, the `VisitStrategy` and `LeaveStrategy` in Listing 11 are predefined LEESA strategies that can not only be configured with programmer-defined visitor objects, but can also be replaced by other programmer-defined strategies. We envision that the combination of `VisitStrategy` and `LeaveStrategy` will be used predominantly because it supports the *hierarchical visitor* [111] pattern to keep track of depth during traversal. This pattern is based on a pair of type-specific actions: `Visit` and `Leave`. The `Visit` action is invoked while *entering* a non-leaf node and the `Leave` action is invoked while *leaving* it. To keep track of depth, the visitor typically maintains an internal stack where the `Visit` function does a "push" operation and `Leave` function does a "pop."

V.4.5 Schema Compatibility Checking

Every syntactically correct traversal expression in LEESA is statically checked against the schema for type errors and any violations are reported back to the programmer. Broadly,

LEESA supports four kinds of checks based on the types and actions participating in the expression. First, only the types representing the vocabulary of the schema are allowed in a LEESA expression. The visitor instances are an exception to this rule. Second, impossible traversal specifications are rejected where there is no way of reaching the elements of a specified type along the axis used in the expression. For example, the child-axis operators ($\gg$, $\gg=$) require (immediate) parent/child relationship between the participating types whereas `DescendantsOf` requires a transitive closure of the child relationship. Third, the argument type of the intermediate results processing actions must match that of the result returned by the previous expression. Finally, the result type of the action must be a type from the schema if the expression is continued further. Table 11 summarizes the LEESA's assertions for checking schema conformance of child and parent axes expressions. Section V.6 describes in detail how we have implemented schema compatibility checking using C++ Concepts.

Notation
A, B : Intermediate result processing actions
L, R : Compound LEESA expressions (*e.g.,* L $\gg$ R)
Result(x) = The right-most type in the expression when x is a LEESA expression. = The result type of the action when x is an action.
Arg(x) = The left-most type in the expression when x is a LEESA expression. = The argument type of the action when x is an action.
Assertions
For L $\gg$ R and L $\gg=$ R, Result(L) must be *parent of* Arg(R)
For L $\ll$ R and L $\ll=$ R, Result(L) must be *child of* Arg(R)
For L $\gg$ A and L $\ll$ A, Result(L) must be *same as* Arg(A)
For A $\gg$ R and A $\ll$ R, Result(A) must be *same as* Arg(R)
For A $\gg$ B and A $\ll$ B, Result(A) must be *same as* Arg(B)

Table 11: Assertions in LEESA for checking schema compatibility

V.5 The Implementation of LEESA

In this section we present LEESA's layered software architecture, the software process of obtaining the static meta-information from the schema, and how we have implemented the strategic traversal combinators in LEESA.

V.5.1 The Layered Architecture of LEESA

Figure 27 shows LEESA's layered architecture. At the bottom is the in-memory *object structure*, which could be a tree or a graph. An *object-oriented data access layer* is a layer of abstraction over the object structure, which provides schema-specific, type-safe interfaces for iteratively accessing the elements in the object structure. Often, a code generator is used to generate language bindings (usually a set of classes) that model the vocabulary. Several different types of code generators such as XML schema compilers [127] and domain-specific modeling tool-suites [39] are available that generate a schema-specific object-oriented data access layer from the static meta-information.

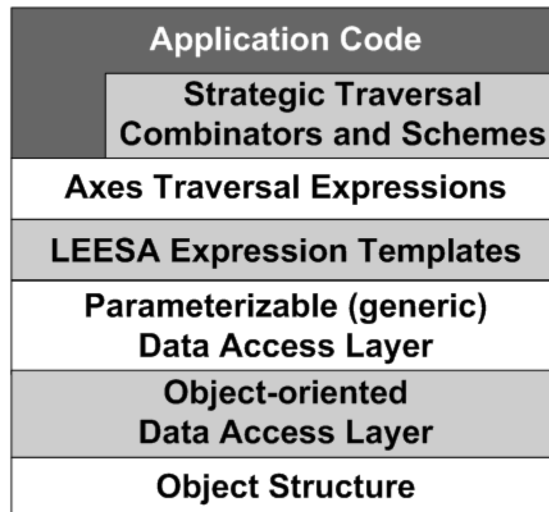

Figure 27: Layered View of LEESA's Architecture (Shading of blocks shown for aesthetic reasons only.)

To support generic traversals, the schema-specific object-oriented data access layer

must be adapted to make it suitable to work with the generic implementation of LEESA's C++ templates. The *parameterizable data access layer* is a thin generic wrapper that achieves this. It treats the classes that model the vocabulary as type parameters and hides the schema-specific interfaces of the classes. This layer exposes a small generic interface, say, `getChildren`, to obtain the children of a specific type from a composite object and say, `getParent`, to obtain the parent of an object. For example, using C++ templates, obtaining children of type `T` of an object of type `U` could be implemented[2] as `U.getChildren<T>()`, where `U` and `T` could be any two classes modeling the vocabulary that have a parent/child relationship. This layer can also be generated automatically from the object structure schema.

Expression Templates [146] is the key idea behind embedding LEESA's traversal expressions in C++. Using operator overloading, expression templates enable *lazy evaluation* of C++ expressions, which is otherwise not supported natively in C++. Lazy evaluation allows expressions – rather than their results – to be passed as arguments to functions to extract results lazily when needed. LEESA overloads the $\gg$, $\ll$, $\gg=$, and $\ll=$ operators using the design method of expression templates to give embedded traversal expressions a look and feel of XPath's axes-oriented traversal specifications. Moreover, LEESA expressions can be passed to other generic functions as arguments to extract results lazily. LEESA's expression templates map the traversal expressions embedded in a C++ program onto the parameterizable data access layer. They raise the level of abstraction by hiding away the iterative process of accessing objects and instead focus only on the *relevant* types in the vocabulary and different strategies (breadth-first and depth-first) of traversal. LEESA's expression templates are independent of the underlying vocabulary. Schema-specific traversals are obtained by instantiating them with schema-specific classes. For more details on LEESA's expression templates, including an example, the readers are directed to our previous work [140].

[2] A widely supported, standard C++ feature called "template explicit specialization" could be used.

Finally, LEESA programmers use the *axes traversal expressions* and *strategic traversal combinators and schemes* to write their traversals. The axes traversal expressions are based on LEESA's expression templates. The strategic traversal combinators use an externalized representation of the static meta-information for their generic implementation. Below we describe the process of externalizing the static meta-information.

V.5.2 Externalizing Static Meta-information.

Figure 28 shows the software process of developing a schema-first application using LEESA. The object-oriented data access layer, parameterizable data access layer, and the static meta-information are generated from the schema using a code generator. Conventional [39, 127] code generators for language-specific bindings generate the object-oriented data access layer only, but for this work we extended the Universal Data Model (UDM) [39] – a tool-suite for developing domain-specific modeling languages (DSML) – to generate the parameterizable data access layer and the static meta-information. The cost of extending UDM is amortized over the number of schema-first applications developed using LEESA. While the static meta-information is used for generic implementations of the primitive strategic combinators, C++ Concepts [57, 126] shown in Figure 28 are used to check the compatibility of LEESA expressions with the schema and report the errors back to the programmer at compile-time. C++ Concepts allow the error messages to be succinct and intuitive. Such a diagnosis capability is of high practical importance as it catches programmer mistakes much earlier in the development lifecycle by providing an additional layer of safety.

The Boost C++ template metaprogramming library (MPL) [3] has been used as a vehicle to represent the static meta-information in LEESA. It provides easy to use, readable, and portable mechanisms for implementing metaprograms in C++. MPL has become a defacto standard for metaprogramming in C++ with a collection of extensible compile-time algorithms, typelists, and metafunctions. Typelists encapsulate zero or more C++ types

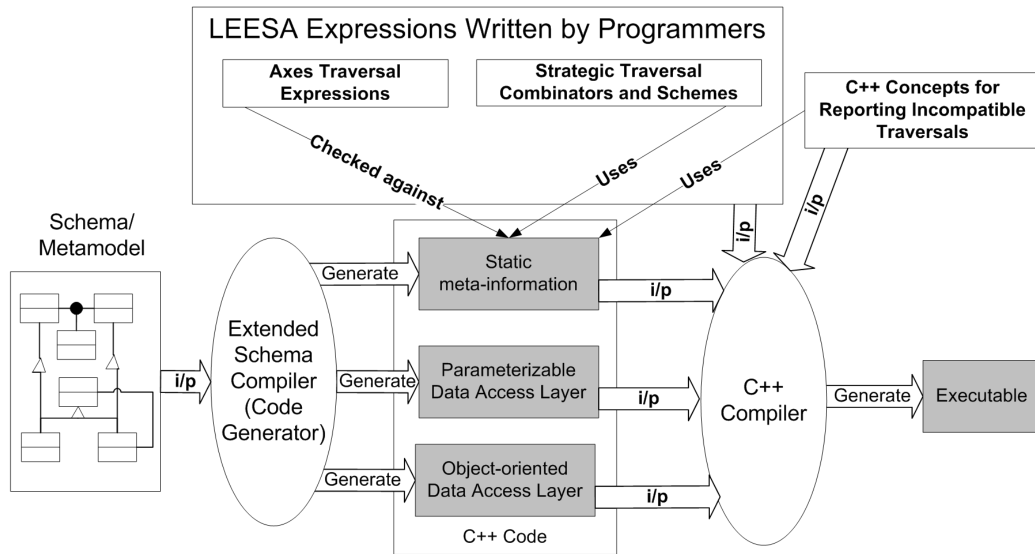

Figure 28: The software process of developing a schema-first application using LEESA. (Ovals are tools whereas shaded rectangular blocks represent generated code)

(programmer-defined or otherwise) in a way that can be manipulated at compile-time using MPL metafunctions.

V.5.2.1 Using Boost MPL to Externalize the Static Meta-information.

The static meta-information (partial) of the HFSM metamodel (Section V.4.1) captured using Boost MPL typelists is shown below.

```
class StateMachine {
   typedef mpl::vector < State, Transition > Children;
};
class State {
   typedef mpl::vector < State, Transition, Time > Children;
};
class Transition {                           // Same as class Time
   typedef mpl::vector < > Children          // empty
};
mpl::contains <StateMachine::Children,State>::value //...(1) true
mpl::front    <State::Children>::type               //...(2) class State
mpl::pop_front<State::Children>::type               //...(3) mpl::vector<Transition,Time>
```

Each class has an associated type called `Children`, which is a MPL typelist implemented using `mpl::vector`. The typelist contains a list of types that are children of its host type. A MPL metafunction called `mpl::contains` has been used to check existence of a type in a MPL typelist. For example, the statement indicated by (1) above checks whether typelist `StateMachine::Children` contains type `State`. It results in a compile-time constant *true* value. Metafunctions `mpl::front` and `mpl::pop_front`, indicated by (2) and (3), are semantically equivalent to "car" and "cdr" list manipulation functions in Lisp. While `mpl::front` returns the first type in the typelist, `mpl::pop_front` removes the first type and returns the remaining typelist.

We leverage this metaprogramming support provided by MPL to represent children, parent, and descendant axes meta-information in C++. We have extended the UDM toolsuite to generate Boost MPL typelists that capture the static meta-information of these axes.

V.5.3 The Implementation of Strategic Traversal Schemes.

In LEESA's implementation of SP, `All` and `One` are *generative* one-layer combinators because their use requires mentioning the type of only the start element where the strategy application begins. The children and descendant (in case of recursive traversal schemes) types of the start type are automatically incorporated into the traversal using the externalized static meta-information and the LEESA's metaprograms that iterate over it.

Listing 12 shows the C++ implementation of the `All` and `Sequence` primitive combinators and the `FullTD` recursive traversal scheme in LEESA. `All` is a class template that accepts `Strategy` as a type parameter, which could be instantiations of other combinators or other instantiations of `All` itself. Execution of `All` begins at the `apply` function, which delegates execution to another member template function called `children`. `All::children` is instantiated as many times as there are children of type `T`. From the `T::Children` typelist, repeated instantiation of the `children` member template

```
template <class Strategy>
class All {
      Strategy strategy_;
   public:
      All (Strategy s) : strategy_(s) { }    // Constructor
      template <class T>
      void apply (T arg) {    // Every strategy implements this member template function.
          // If T::Children typelist is empty, calls (B) otherwise calls (A)
          children(arg, typename T::Children());
      }
   private:
      template <class T, class Children>
      void children(T arg, Children)   {                          // ........... (A)
         typedef typename mpl::front<Children>::type Head;        // ...(1)
         typedef typename mpl::pop_front<Children>::type Tail;    // ...(2)
         for_each c in arg.getChildren<Head>()                     // ...(3)
             strategy_.apply(c);
         children(arg, Tail());                                    // ...(4)
      }
      template <class T>
      void children(T, mpl::vector<> /* empty typelist */) { }    // ........... (B)
};
-----------------------------------------------------------------------------------------
template <class S1, class S2>           template <class Strategy>
class SEQ {                             class FullTD {
   S1 s1_; S2 s2_;                          Strategy st_;
public:                                 public:
   SEQ(S1 s1, S2 s2)                        FullTD(Strategy s) : st_(s) {}
      : s1_(s1), s2_(s2) {}                 template <class T>
   template <class T>                       void apply (T arg) {
   void apply (T arg) {                        All<FullTD> all(*this);
      s1_.apply(arg);                          SEQ<Strategy, All<FullTD> > seq(st_, all);
      s2_.apply(arg);                          seq.apply(arg);
   }                                        }
};                                      };
```

Listing 12: C++ implementations of `All` and `SEQ` (Sequence) primitive combinators and the `FullTD` recursive traversal scheme

function are obtained using the metaprogram indicated by statements (1), (2), and (4) in Listing 12.

Similar to list processing in functional languages, statement (1) yields the first type (`Head`) in the typelist whereas statement (2) yields the remaining typelist (`Tail`). Statement (4) is a compile-time recursive call to itself but with `Tail` as its second parameter. This compile-time recursion terminates only when `Tail` becomes empty after successive application of `mpl::pop_front` metafunction. When `Tail` is an empty typelist, the `children` function marked by (B) is invoked terminating the compile-time recursion.

Figure 29 shows a graphical illustration of this recursive instantiation process. Multiple recursive instantiations of function `children` are shown in the order they are created with a progressively smaller typelist as its second parameter. Finally, the statement marked as (3) is using the parameterizable data access interface `T::getChildren`, which returns all the `Head` type children of `arg`.

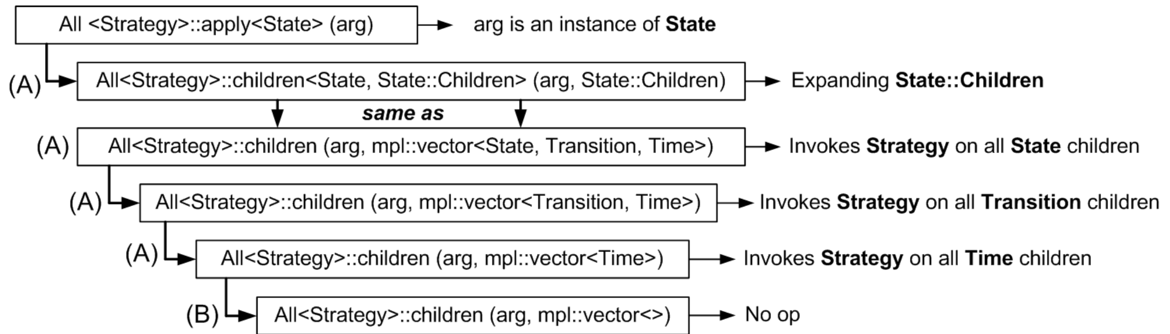

Figure 29: Compile-time recursive instantiations of the `children` function starting at `All<Strategy>::apply<State>(arg)` when arg is of type `State`.

We apply a sophistication of C++ template metaprogramming technique called *loop unrolling* [28] in the above generative one-layer combinators to alleviate the complexity of mentioning all the children and descendant types of the start type. Note that immediate children types of a given type `T` in the schema are obtained at compile-time using the associated type `T::Children` as described in Section V.5.2. The primitive combinator templates that encapsulate the traversal strategies are instantiated repeatedly using LEESA's template metaprograms iterating over `T::Children` typelists. These typelists are finite in length and therefore repeated instantiation is guaranteed to terminate. Finally, these automatically instantiated templates are responsible for run-time traversal.

V.5.3.1 Efficient Descendant Axis Traversal.

Compile-time customizations of the primitive combinator `All` and in turn `FullTD` traversal scheme are used for efficient implementation of the *descendant* axis traversal.

LEESA can prevent traversals into unnecessary substructures by controlling the types that are visited during recursive traversal of `FullTD` and `FullBU` schemes. LEESA customizes the behavior of the `All` primitive combinator using the descendant types information that is obtained from the schema. The `T::Children` typelist in the `All::apply` function is manipulated using C++ template metaprogramming such that the schema types that have no way of reaching the objects of the target type are eliminated before invoking the `All::children` function. This is achieved using compile-time boolean queries over the list of descendant types implemented using MPL's metafunction `mpl::contains` as described in Section V.5.2. All these metaprograms are completely encapsulated inside the C++ class templates that implement recursive traversal schemes and are not exposed to the programmers.

While static meta-information can be used for efficient traversal, the same meta-information can be used to check the LEESA expressions for their compatibility with the schema. We describe that next.

V.6 Domain-specific Error Reporting using C++ Concepts

In DSL literature [29, 84], embedded DSLs have been criticized for their lack of support for domain-specific error reporting. The importance of intuitive error messages should not be underestimated as it directly affects the programmer's effectiveness in locating and correcting errors in a DSL program. This issue is all the more important for embedded DSLs since their compiler is the same as the host language compiler, and hence the error reports are often in terms of the host language artifacts instead of domain-specific artifacts that are relevant to the problem. Moreover, for embedded DSLs in C++ that are implemented using templates, the problem is further exacerbated because templates lack early modular (separate) type-checking. As a result, extremely long error messages pointing deep into the implementation details are produced even for minor syntactical inconsistencies. We

leverage C++ Concepts [57] to produce significantly shorter and intuitive error messages when a LEESA expression violating the schema is identified at compile-time.

V.6.1 Early Type-checking of C++ Templates using Concepts

C++ Concepts [57] have been developed to address the problem of late type-checking of templates during compilation. Concepts express the syntactic and semantic behavior of types and constrain the type parameters in a C++ template, which are otherwise unconstrained. Concepts allow separate type-checking of template definitions from their uses, which makes templates easier to use and easier to compile. The set of constraints on one or more types are referred to as *Concepts*. Concepts describe not only the functions and operators that the types must support but also other accessible types called *associated types*. The types that satisfy the requirements of a concept are said to *model* that concept. When a concept constrained C++ template is instantiated with a type that does not model the concept, an error message indicating the failure of the concept and the type that violates it are shown at the call site in the source code. An experimental support for C++ Concepts has been implemented in the ConceptGCC [55] compiler.

V.6.2 Schema Compatibility Checking Using Concepts and Metaprogramming

We have defined several C++ Concepts in LEESA that must be satisfied by different types participating in a LEESA expression. These Concepts are related primarily to child, parent, descendant, and ancestor axes traversals and the invocation of actions for intermediate results processing. For example, each type in a child axis traversal expression must model a `ParentChildConcept` with respect to its preceding type. An implementation of the `ParentChildConcept` is shown below.

```
concept ParentChildConcept <typename Parent, typename Child> {
  typename Children = typename Parent::Children;
  typename IsChild = typename mpl::contains<Children, Child>::type;
```

```
  requires std::SameType<IsChild, true_type>;
};
```

The concept is parameterized with two types and essentially requires that the Child type be present in the list of children of the Parent type. This is achieved by (1) obtaining the result of the application of MPL metafunction contains on Parent's associated type Children and (2) enforcing the type of the result to be the same as true_type, which signifies success. If the Concept does not hold, a short error message is produced stating the failure of the Concept and the types that violate it. The error is reported at the first occurrence of the type that violates it regardless of the length of the expression. As the length of the erroneous LEESA expression grows, the error output grows linearly due to increasing size of the recursively constructed type using expression templates. However, the reason and location are always stated distinctly in the form of concept violation.

For example, consider a LEESA expression, "StateMachine() $\gg$ Time()", which is incorrect with respect to the metamodel of the HFSM modeling language because Time is not an immediate child of StateMachine and therefore, does not satisfy the Parent-ChildConcept described before. Below, we have shown the actual error message produced by the ConceptGCC [55] compiler, which is only four lines long, and clearly states the reason and the location (both on the fourth line) of the error. The line numbers in the error message map to the programmer-written traversal expressions and not in the implementation of LEESA.

```
t.cc: In function 'int main()':
t.cc:99: error: no match for 'operator>>' in 'StateMachine() >> Time()'
t.cc:85: note: candidates are: R LEESA::operator>>(const L&, const R&)
               [with L = StateMachine, R = Time] <requirements>
t.cc:99: note: no concept map for requirement
         'LEESA::ParentChildConcept<StateMachine, Time>'
```

Similar to the `ParentChildConcept`, `DescendantConcept` has also been defined, which depends on the automatically generated list of descendant types.

V.7 Evaluation of LEESA

In this section, we present two case-studies evaluating the capabilities of LEESA. In the first case study, we present quantitative results on LEESA's effectiveness in reducing efforts while programming traversals compared to the third generation object-oriented languages. The second case-study measures the compile-time and run-time performance of LEESA on contemporary C++ compilers and platfroms.

V.7.1 Case-study 1: Evaluating Programmer Productivity

Experimental setup. We conducted the experiments using our open-source domain-specific modeling tool-suite: CoSMIC.[3] CoSMIC is a collection of domain-specific modeling languages (DSML), interpreters, code generators, and model-to-model transformations developed using Generic Modeling Environment (GME) [78], which is a meta-programmable tool for developing DSMLs. CoSMIC's DSMLs are used for developing distributed applications based on component-based middleware. For instance, the Platform Independent Component Modeling Language (PICML) [15] is one of the largest DSMLs in CoSMIC for modeling key artifacts in all the life-cycle phases of a component-based application, such as interface specification, component implementation, hierarchical composition of components, and deployment. A PICML model may contain up to 300 different types of objects. Also, PICML has over a dozen model interpreters that generate XML descriptors pertaining to different application life-cycle stages. All these interpreters are implemented in C++ using UDM as the underlying object-oriented data access layer.

Methodology. The objective of our evaluation methodology is to show the reduction

[3] http://wwww.dre.vanderbilt.edu/cosmic

Traversal Pattern	Axis	Occurrences	Original #lines (average)	#Lines using LEESA (average)
A single loop iterating over a list of objects	Child	11	8.45	1.45
	Association	6	7.50	1.33
5 sequential loops iterating over siblings	Sibling	3	41.33	6
2 Nested loops	Child	2	16	1
Traversal-only visit functions	Child	3	11	0
Leaf-node accumulation using depth-first	Descendant	2	43.5	4.5
Total traversal code	-	All	414 (absolute)	53 (absolute)

Table 12: Reduction in code size (# of lines) due to the replacement of common traversal patterns by LEESA expressions.

in programming efforts needed to implement commonly observed traversal patterns using LEESA over traditional iterative constructs.

To enable this comparison, we refactored and reimplemented the traversal related parts of PICML's deployment descriptor generator using LEESA. This generator exercises the widest variety of traversal patterns applicable to PICML. It amounts to little over 2,000 lines[4] of C++ source code (LOC) out of which 414 (about 21%) LOC perform traversals. It is organized using the Visitor [48] pattern where the accept methods in the object structure classes are non-iterating and the entire traversal logic along with the type-specific actions are encapsulated inside a monolithic visitor class. Table 12 shows the traversal patterns we identified in the generator. We replaced these patterns with their equivalent constructs in LEESA. This procedure required some refactoring of the original code.

Analysis of results. Table 12 shows a significant reduction in the code size due to LEESA's succinct traversal notation. As expected, the highest reduction (by ratio) in code size was observed when two ad-hoc implementations of depth-first search (*e.g.,* searching nested components in a hierarchical component assembly) were replaced by LEESA's

[4]The number of lines of source code is measured excluding comments and blank lines.

adaptive expressions traversing the descendant axis. However, the highest number of reduction in terms of the absolute LOC (114 lines) was observed in the frequently occurring traversal pattern of a *single loop*. Cumulatively, leveraging LEESA resulted in 87.2% reduction in traversal code in the deployment descriptor generator. We expect similar results in other applications of LEESA.

V.7.1.1 Incremental Adoption of LEESA.

It is worth noting here that due to its pure embedded approach, applying LEESA in the existing model traversal programs is considerably simpler than external DSLs that generate code in bulk. Incremental refactoring of the original code-base was possible by replacing one traversal pattern at a time while being confident that the replacement is not changing the behavior in any unexpected ways. Such incremental refactoring using *external* traversal DSLs that use a code generator would be extremely hard, if not impossible. Our pure embedded DSL approach in LEESA allows us to distance ourselves from such *all-or-nothing* predicament, which could potentially be a serious practical limitation. We expect that a large number of existing C++ applications that use XML data-binding [127] can start benefiting from LEESA using this incremental approach provided their XML schema compilers are extended to generate the parameterizable data access layer and the meta-information.

V.7.2 Case-study 2: Evaluating Compile- and Run-time Performance

Rationale. LEESA relies heavily on C++ meta-programming but C++ compilers are often not optimized for it. This motivates us to evaluate the effect of LEESA's meta-programs on the *edit-compile-test* cycle, which directly affects the productivity of professional programmers using the contemporary development environments for compiled languages. We also evaluate the effect of generative programming on the object code size and the *abstraction penalty* incurred by LEESA in terms of its run-time performance.

Experimental setup. We experimented with an open-source XML data binding [114]

```
<xs:element name="catalog">
 <xs:complexType> <xs:sequence>
  <xs:element name="book" maxOccurs="unbounded">
   <xs:complexType> <xs:sequence>
    <xs:element name="name" type="xs:string" />
     <xs:element name="author" maxOccurs="unbounded">
      <xs:complexType> <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="born" type="xs:date" />
        <xs:element name="died" type="xs:date" minOccurs="0"/>
       </xs:sequence> </xs:complexType>
     </xs:element>
   </xs:sequence> </xs:complexType>
  </xs:element>
 </xs:sequence> </xs:complexType>
</xs:element>
```

Listing 13: XML Schema Definition (XSD) of the catalog XML

```
<catalog>
  <book>
    <title>Hamlet</title>
    <price>9.99</price>
    <author>
      <name>William Shakespeare</name>
      <country>England</country>
    </author>
  </book>
  <book>...</book>
    ...
</catalog>
```

Listing 14: An XML document containing a book catalog.

tool for C++ called XSD [1]. XSD generates vocabulary-specific C++ classes from an XML schema, which describes the structure of domain-specific XML document instances. We developed a Python script to generate the generic data access layer to hide the schema-specific C++ interfaces. Moreover, the script is also used to externalize the meta-information and embed it in the generated C++ classes. For experimentation we used the XML schema shown in Listing 13 and the corresponding XML instance is shown in Listing 14. The C++ classes generated by XSD are shown in Listing 15 whereas the generic data access layer generated by our Python script is shown in Listing 16.

```
class title   {...};
class price   {...};
class name    {...};
class country {...};
class author  { // Constructors are not shown.
  private: name name_;
           country country_;
  public:  name get_name() const;
           void set_name(name const &);
           country get_country() const;
           void set_country(country const &);
};
class book { // Constructors are not shown.
  private: title title_;
           price price_;
           std::vector<author> author_sequence_;
  public:  title get_title() const;
           void set_title(title const &);
           price get_price() const;
           void set_price(price const &);
           std::vector<author> get_author() const;
           void set_author(std::vector<author> const &);
};
class catalog {...};  // Contains a std::vector of books.
```

Listing 15: C++ classes generated by a typical XML data binding tool for the catalog object-model

Analysis of Results. Table 13 shows the comparison of code sizes for one small (10 types) and one large (300 types) schema. We evaluated a single LEESA expression of each query type shown in the table against equivalent programs written using object-oriented abstractions only.

```
name children (author a, name const *) {
  return a.get_name();
}
country children (author a, country const *) {
  return a.get_country();
}
title children (book b, title const *) {
  return b.get_title();
}
price children (book b, price const *) {
  return b.get_price();
}
std::vector<author> children (book b, author const *) {
  return b.get_author();
}
std::vector<book> children (catalog c, book const *) {
  return c.get_book();
}
```

Listing 16: Automatically generated generic data access layer

Schema size	**Query type**	**Lines of code**		**Object code** (Megabytes)	
		(A)	(B)	(A)	(B)
Small	Child-axis, AllDescendants, LevelDescendants	3	13	0.38	0.35
Large	Child-axis	3	39	7.42	7.15
	AllDescendants	3	136	7.46	7.19
	LevelDescendants	4	88	7.49	7.18

Table 13: Comparison of the static metrics. (A) = LEESA and (B) = Object-oriented solution

The difference in the lines of code (LOC) in Table 13 clearly shows that LEESA expressions are highly expressive and succinct compared to the OO-centric solution. Data for the object code sizes reveals that LEESA's generative programming approach does not result in object-level code bloat.

Comparisons of the compilation times using the gcc 4.5 compiler and the test programs based on the large schema are shown in Figure 30. LEESA-based programs consistently require more time to compile than pure OO solutions because contemporary C++ compilers are not optimized for heavy meta-programming. The increasing compilation-times may

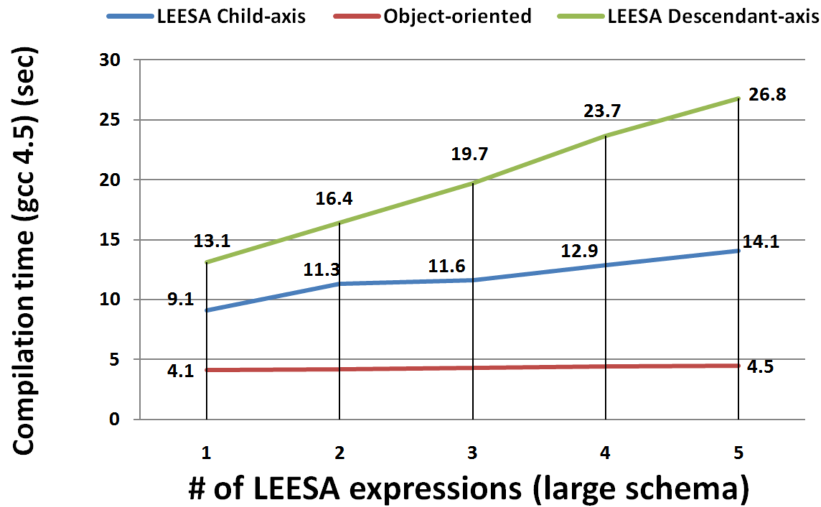

Figure 30: Comparisons of compilation times with LEESA and the pure object-oriented solution

lengthen the *edit-compile-test* cycles. However, we believe that the succinctness and intuitiveness of LEESA not only requires fewer key-strokes but also fewer compilations than the equivalent object-oriented programs.

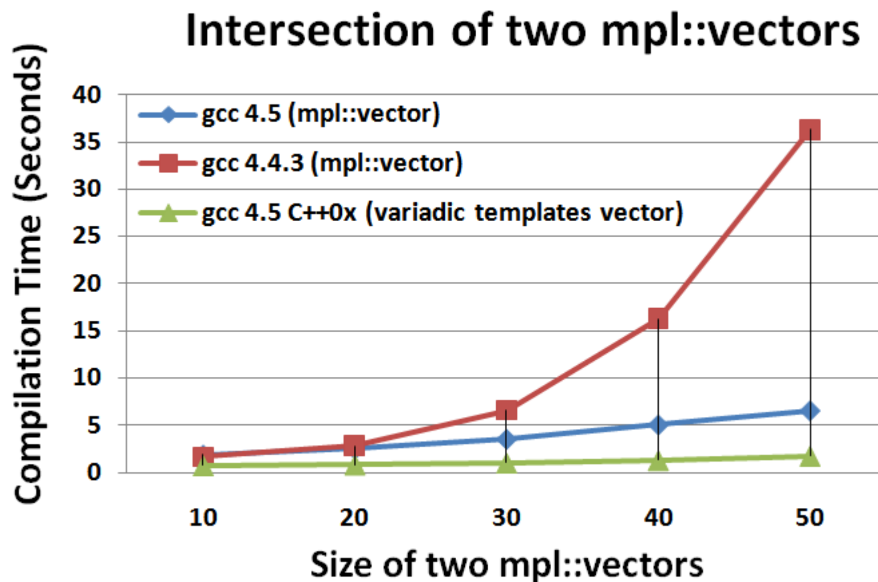

Figure 31: Comparison of meta-programming performance of different C++ compilers

Furthermore, *variadic templates* [56], an upcoming standard C++ language (C++0x [19]) feature that allows arbitrary number of template parameters is a promising solution to reduce the compilation overhead dramatically. Figure 31 shows the difference between the compilation times of three versions of the gcc compiler. The compilers were tested on a C++ meta-program that computes an intersection of two MPL typelists at compile-time, which is a frequently executed meta-program in LEESA. With the increase in the size of the typelist, compilation times of gcc 4.4.3 increases super-linearly whereas compilation times of gcc 4.5 increases linearly. However, with the variadic template support, size of the typelist has little or no effect on the compilation time of gcc 4.5 in C++0x mode. These results indicate that with the advent of C++0x, heavy typelist-oriented meta-programming may become practical.

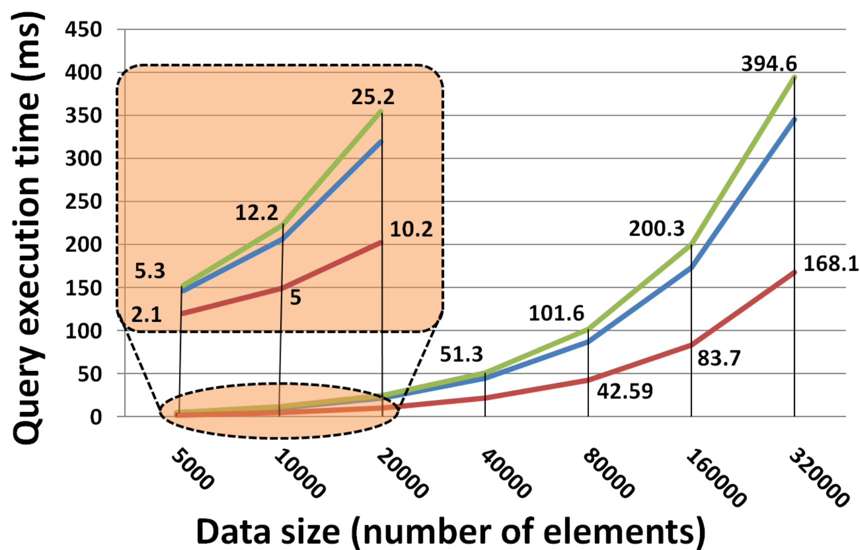

Figure 32: Run-time performance comparisons of LEESA and the pure object-oriented solution

Figure 32 compares the run-time performance of a LEESA query with an optimized

hand-written object-oriented solution. We compared the time needed to construct a standard C++ container (*e.g.,* std::vector<T>) of `name` objects from a set of large book catalogs. LEESA's descendant axis has consistently higher overhead by a factor of 2.5 compared to the hand-written solution. This *abstraction penalty* stems from the construction, copying, and destruction of the internal dynamic data structures LEESA maintains.

In practice, however, query execution amounts to a small fraction of the overall XML processing, which involves I/O, parsing, XML validation, construction of the in-memory object model, and the execution of business logic. For instance, our 320,000 elements test took over 36 seconds for XML parsing, validation, and object model construction, which is nearly two orders of magnitude higher than the query execution time.

V.8 Comparing LEESA with Related Research

In this section we place LEESA in the context of a sampling of the most relevant research efforts in the area of object structure traversal.

XPath 2.0 [154] is a structure-shy XML query language that allows node selection in a XML document using downward (children, descendant), upward (parent, ancestor), and sideways (sibling) axes. In general, XPath supports more powerful node selection expressions than LEESA using its untyped unconstrained (*i.e. axis::**) axes. XPath's formal semantics [155] describe how XML schema could be used for *static type analysis* to detect certain type errors and to perform optimizations. However, contemporary XPath programming APIs for C++ use string encoded expressions, which are not checked against the schema at compile-time. Moreover, unlike XPath, type-specific behavior can be composed over the axes-oriented traversals using LEESA.

Adaptive Programming (AP) [79, 80] specifies structure-shy traversals in terms of milestone classes composed using predicates, such as *from*, *to*, *through*, and *bypass*. It uses static meta-information to optimize traversals as well as to check their compatibility against

the schema. While LEESA focuses on accumulation of nodes using its axes-oriented notation and programmability of traversals using its strategic combinator style, AP focuses on a collection of unique paths in the object graph specified using the above mentioned predicates. The use of visitors in LEESA to modularize type-specific actions is similar in spirit to the code wrappers in AP.

Strategic Programming (SP) [75, 147], which began as a term rewriting [147] language has evolved into a language-interparadigmatic style of programming traversals and has been incarnated in several other contexts, such as functional [74], object-oriented [148], and embedded [17]. The strategic traversal expressions in LEESA are based on a new embedded incarnation of SP in an *imperative* language, C++. Unlike [17], however, no compiler extension is necessary. Also, all the expressions are *statically* checked against the schema, unlike visitor combinators [148].

Scrap++ [88] presents a C++ templates-based approach for implementing Haskell's "Scrap Your Boilerplate" (SYB) design pattern, which is remarkably similar to SP. Scrap++'s approach depends on recursive traversal combinators, a one-layer traversal, and a type extension of the basic computations. However, LEESA's approach is different in many significant ways. First, unlike LEESA, the objective of Scrap++ is to mimic Haskell's SYB and therefore does not provide an intuitive axes traversal notation. Second, LEESA presents a software process for generating schema-specific meta-information that is used during compilation for generating traversals as well as compatibility checking. Third, SYB lacks parental and sibling contexts. Finally, no technique is provided in Scrap++ to produce intuitive error messages.

Lämmel et al. [76] present a way of realizing adaptive programming predicates (*e.g., from*, *to*, *through*, and *bypass*) by composing SP primitive combinators and traversal schemes.

Due to a lack of static type information, their simulation of AP in terms of SP lacks important aspects of AP, such as static checking and avoiding unnecessary traversal into substructures. LEESA, on the other hand, uses the externalized meta-information to not only statically check the traversals but also makes them efficient.

Static meta-information has also been exploited by Cunha et al. [5] in the transformation of structure-shy XPath and SP programs for statically optimizing them. Both approaches eliminate unnecessary traversals into substructures, however, no transformation is necessary in the case of LEESA. Instead, the behaviors of `All` and `One` primitive combinators are customized at compile-time to improve efficiency. Moreover, LEESA's structure-shy traversals support mutually recursive types, unlike [5].

Lämmel [73] sketches an encoding of XPath-like axes (downward, upward, and sideways) using strategic function combinators in the SYB style. LEESA is similar to this work because both the approaches suggest an improvement of XPath-like set of axes with support for strategic, recursive traversal abstractions and provide a way of performing schema-conformance checking. The key differences are the improved efficiency of the *descendant* axis traversal in case of LEESA, its domain-specific error reporting capability, and its use of an imperative, object-oriented language as opposed to Haskell, which is a pure functional language.

Gray et al. [51] and Ovlinger et al. [107] present an approach in which traversal specifications are written in a specialized language separate from the basic computations. A code generator is used to transform the traversal specifications into imperative code based on the Visitor pattern. This approach is, however, heavyweight compared to the embedded approach because it incurs high cost of the development and maintenance of the language processor.

Language Integrated Query (LINQ) [7] is a Microsoft .NET technology that supports SQL-like queries natively in a program to search, project and filter data in arrays, XML, relational databases, and other third-party data sources. "LINQ to XSD" promises to add

much needed typed XML programming support over its predecessor "LINQ to XML." LINQ, however, does not support strategic combinator style like LEESA. The Object Constraint Language (OCL) [101] is a declarative language for describing well-formedness constraints and traversals over object structures represented using UML class graphs. OCL, however, does not support *side-effects* (*i.e.*, object structure transformations are not possible).

Czarnecki et al. [29] compare staged interpreter techniques in MetaOCaml with the template-based techniques in Template Haskell and C++ to implement embedded DSLs. Two approaches – *type-driven* and *expression-driven* – of implementing an embedded DSL in C++ are presented. Within this context, our previous work [140] presents LEESA's expression-driven pure embedding approach. Spirit[5] and Blitz++[6] are two other prominent examples of expression-driven embedded DSLs in C++ for recursive descent parsing and scientific computing, respectively. Although LEESA shares the implementation technique of expression templates with them, strategic and XPath-like axes-oriented traversals cannot be developed using Spirit or Blitz++.

[5]http://spirit.sourceforge.net
[6]http://www.oonumerics.org/blitz

CHAPTER VI

CONCLUDING REMARKS

Distributed real-time and embedded (DRE) systems are the backbone of many mission-critical domains, such as avionic mission computing, air traffic control, multi-satellite missions, and shipboard computing. The criticality of these domains require the DRE systems to be highly available and predictable (low latency and jitter) while operating in environments where network and CPU resources are often fluctuating. To meet the stringent quality-of-service (QoS) requirements of the DRE systems, effective techniques are required in all phases (specification to run-time) of the development lifecycle. This dissertation presents the research challenges in developing highly-available DRE systems and resolves them using novel solutions based on contemporary software development paradigms: component-based software engineering (CBSE) and model-driven engineering (MDE).

First, it describes the difficulties in reasoning about the DRE systems due to the tangling of availability concerns with composition, deployment, and timeliness concerns. To simplify reasoning of multiple QoS for DRE systems, it presents a domain-specific modeling language named Component QoS Modeling Language (CQML), that modularizes timeliness, fault-tolerance, and network bandwidth allocation decisions of component-based systems away from composition and deployment concerns. Second, this dissertation identifies the need for a coherent approach for capturing fault-tolerance requirements through all phases of the system lifecycle including specification, design, composition, deployment, configuration, and run-time. This challenge is resolved using a coherent model-driven process that synthesizes source code and system configuration to deploy a fault-tolerant component-based system with minimum human intervention. The process is implemented in GeneRative Aspects for Fault-Tolerance (GRAFT), which is a multi-stage

aspect-oriented model transformation process based on model-to-model, model-to-text, model-to-code transformations.

Although CQML and GRAFT address the design- and deployment-time issues in fault-tolerance provisioning satisfactorily, dealing with the side-effects of replication in multi-tier DRE systems requires run-time solutions. This dissertation shows how the orphan request problem manifests itself in multi-tier DRE systems due to a variety of sources of non-determinism in such systems. To address the run-time issues, a *group-failover* protocol is presented that ensures consistent and timely data for multi-tier real-time systems with non-deterministic stateful components. The experimental evaluations of the group-failover protocol highlight its suitability for real-time systems.

Finally, this dissertation presents the deficiencies in using the object-oriented paradigm for a key step in model-driven engineering: object structure traversal. The deficiencies stem from the inability of synthesizing domain-specific abstractions for programming traversals using object-oriented paradigm alone. To resolve these challenges, a multi-paradigm design approach that combines generic-, generative-, strategic-, and meta-programming paradigms is presented. The approach is implemented in Language for Embedded quEry and traverSAl (LEESA), which is an embedded domain-specific language for programming reusable, generic traversals using the multi-paradigm programming capabilities of C++.

APPENDIX A

UNDERLYING TECHNOLOGIES

This appendix summarizes the various technologies that are used to build the domain-specific languages and the fault-tolerant middleware solutions that are described in this thesis.

A.1 Overview of Lightweight CCM

The OMG Lightweight CCM (LwCCM) [98] specification standardizes the development, configuration, and deployment of component-based applications. LwCCM uses CORBA's distributed object computing (DOC) model as its underlying architecture, so applications are not tied to any particular language or platform for their implementations. *Components* in LwCCM are the implementation entities that export a set of interfaces usable by conventional middleware clients as well as other components. Components can also express their intent to collaborate with other components by defining *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components. *Homes* are factories that shield clients from the details of component creation strategies and subsequent queries to locate component instances.

Figure 33 illustrates the layered architecture of LwCCM, which includes the following entities:

- LwCCM sits atop an **object request broker** (ORB) and provides **containers** that encapsulate and enhance the CORBA portable object adapter (POA) demultiplexing mechanisms. Containers support various pre-defined hooks and strategies, such as

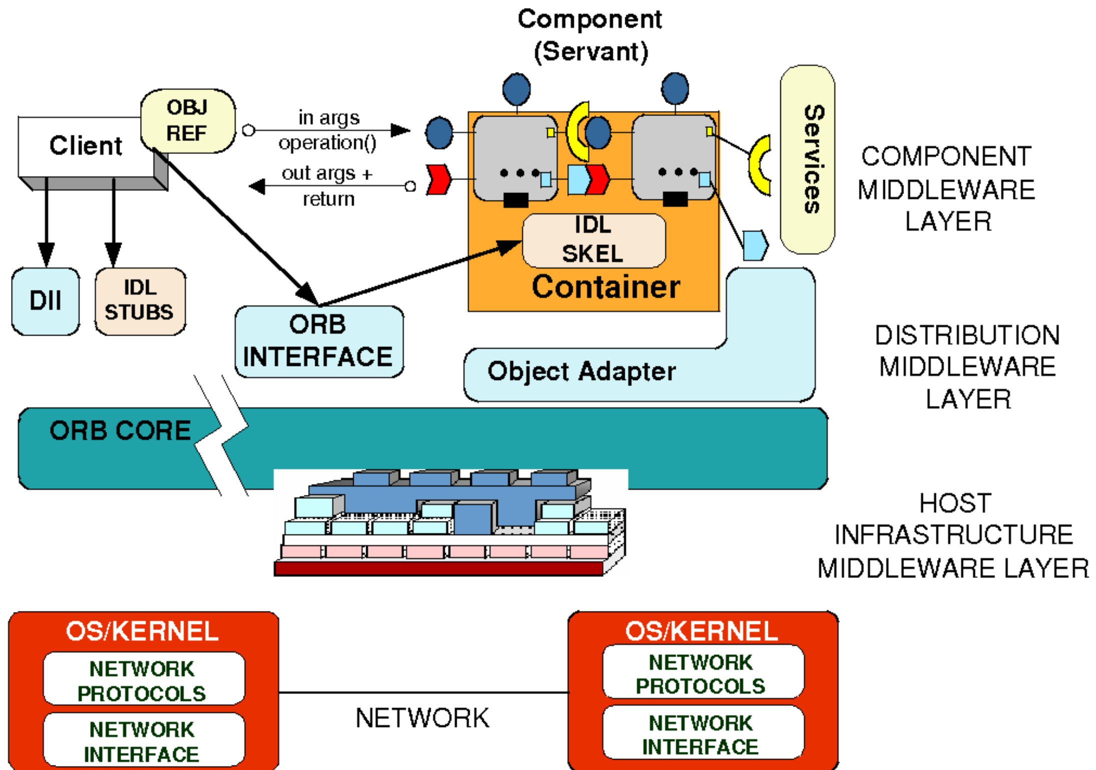

Figure 33: Layered LwCCM Architecture

persistence, event notification, transaction, and security, to the components it manages.

- A *component server* plays the role of a process that manages the homes, containers, and components.

- Each container manages one type of component and is responsible for initializing instances of this component type and connecting them to other components and common middleware services.

- The **component implementation framework** (CIF) consists of patterns, languages and tools that simplify and automate the development of component implementations which are called as **executors**. Executors actually provide the component's business logic.

- Component Implementation Definition Language (CIDL) is a text-based declarative language that defines the behavior of the components. In order to shield the component application developers from many complexities associated with programming POAs like servant activation and deactivation, a CIDL compiler generates infrastructure glue code called *servants*. Servants (1) activate components within the container's POA, (2) manage the interconnection of a component's ports to the ports of other components, (3) provide implementations for operations that allow navigation of component facets, and (4) intercept invocations on executors to transparently enact various policies, such as component activation, security, transactions, load balancing, and persistence.

- To initialize an instance of a component type, a container creates a component home. The component home creates instances of servants and executors and combines them to export component implementations to the external world.

- Executors use servants to communicate with the underlying middleware and servants delegate business logic requests to executors. Client invocations made on the component are intercepted by the servants, which then delegate the invocations to the executors. Moreover, the containers can configure the underlying middleware to add more specialized services, such as integrating an event channel to allow components to communicate and add Portable Interceptors to intercept component requests.

A.2 Overview of Component Middleware Deployment and Configuration

After components are developed and component assemblies are defined, they must be deployed and configured properly by deployment and configuration (D&C) services. The D&C process of component-based systems usually involves a number of service objects that must collaborate with each other. Figure 34 gives an overview of the OMG D&C model, which is standardized by OMG through the Deployment and Configuration

(D&C) [100] specification to promote component reuse and allow complex applications to be built by assembling existing components. As shown in the figure, since a component-based system often consists of many components that are distributed across multiple nodes, in order to automate the D&C process, these service objects must be distributed across the targeted infrastructure and collaborate remotely.

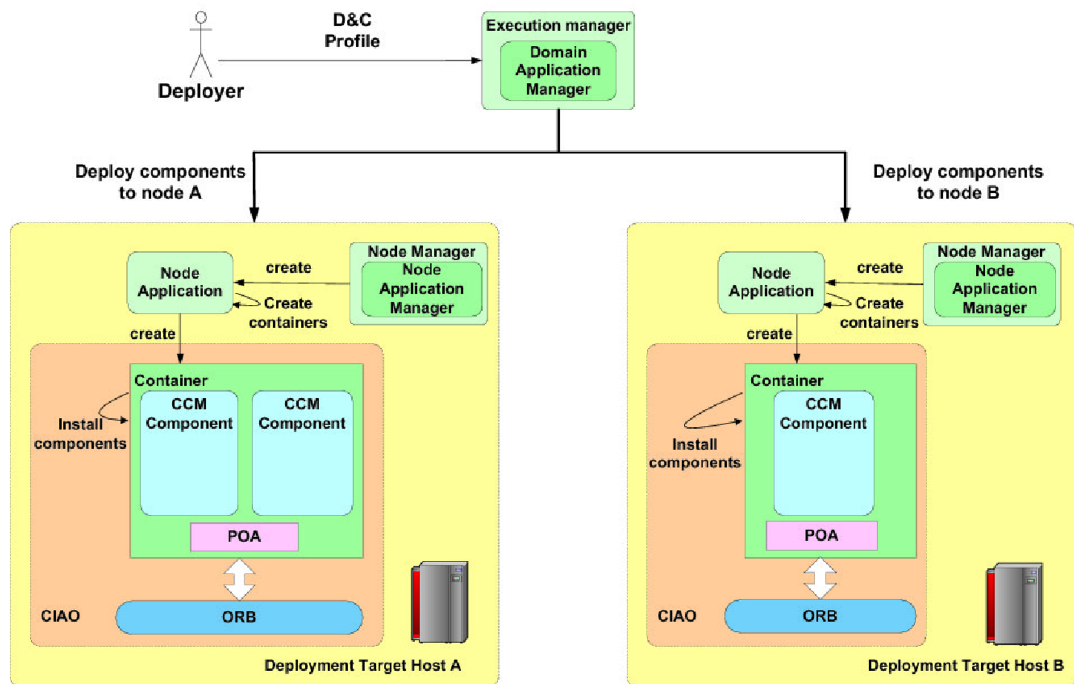

Figure 34: An Overview of OMG Deployment and Configuration Model

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

A.3 Overview of Generic Modeling Environment (GME)

GME is a configurable toolkit for creating DSMLs and program synthesis environments. Third-generation programming languages, such as C++, Java, and C#, employ imperative techniques for development, deployment, and configuration of systems. For example, real-time QoS provisioning with object request brokers is conventionally done using imperative techniques that specify the QoS policies at the same level of abstraction as the mechanisms that implement those policies [112].

In contrast, GME-based DSMLs use a declarative approach that clearly separates the specification of policies from the mechanisms used to enforce the policies. Policy specification is done at a higher level of abstraction (and in less amount of detail), *e.g.*, using *models* and declarative configuration languages. Declarative techniques help relieve users from the intricacies of how the policies are mapped onto the underlying mechanisms implementing them, thereby simplifying policy modifications.

GME-based DSMLs are described using *metamodels*, which specify the modeling paradigm or language of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain, *e.g.*, which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the family of models that can be created using the resultant modeling environment.

For example, a DSML might represent the different hardware elements of a radar system and the relationships between them in a component middleware technology like LwCCM. Likewise, it might represent the different elements, such as *EJBComponent*, *EJBHome*, *EJBContainer* and *ApplicationServer*, that are present in a component middleware technology like EJB. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

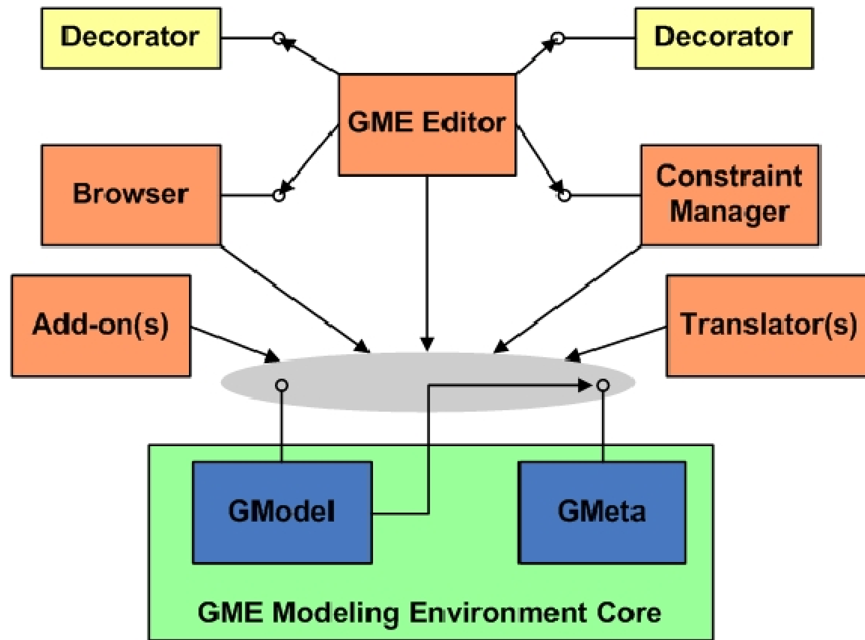

Figure 35: Overview of GME

To create metamodels and their associated DSMLs, GME uses a modular and component-based architecture as shown in Figure 35 (see [78] for a detailed overview of the GME architecture). Application developers create new DSMLs using the following core components of GME: (1) *GME Editor*, (2) *Browser*, (3) *Constraint Manager*, (4) *Translator*, and *Add-ons*. To support building large-scale and complex systems, GME's Editor and the Browser provide basic building blocks to model different entities of the system and express the relationships between those different entities. GME's Constraint Manager catches errors when models are constructed with incorrect relationships or associations. GME's Add-ons provide capabilities to extend the GME Editor, and its Translators support the analysis of models and synthesize various types of artifacts, such as source code, deployment descriptors, or simulator input.

A.4 Overview of Constraint-Specification Aspect Weaver (C-SAW)

The Constraint-Specification Aspect Weaver [130] (C-SAW) weaver is a generalized model-to-model transformation engine for manipulating domain-specific models, which is

implemented as a plug-in for the Generic Modeling Environment (GME) [78]. It can also be used to instrument structural changes within the model according to some higher-level requirement that represents a crosscutting concern. In C-SAW, transformations are specified at the modeling level using the Embedded Constraint Language (ECL). The *aspect* specification defines a set of locations in the domain models; and *strategy* specifications describe the transformation behaviors, which can be recursive. ECL provides support for composing complex transformation behaviors using a library of commonly used operations such as addition, deletion, selection, searching, and cloning of modeling elements and basic logical, numerical, and string manipulation operations.

A.5 Overview of C++ Template Metaprogramming

A *metaprogram* is defined as a program that manipulates/generates another program. A variation of metaprogramming is available in C++ where the metaprograms are expressed using C++ templates. Templates make C++ a two-level language where there are static constructs for metaprogramming and dynamic constructs for conventional run-time of the program. The language for the static constructs is a low-level, pure functional language which is interpreted by the C++ compiler at compile-time. Naturally, programming in the pure functional sub-language of C++ is a shift in paradigm from its conventional imperative sub-language. Moreover, development of ad-hoc and extremely clever metaprogramming techniques (hacks) that exploit subtleties of the C++ compilation process have increased the complexity of C++ metaprogramming over the years.

The Boost C++ template metaprogramming library (MPL) [3] provides easy to use, readable, and portable mechanisms for implementing metaprograms on top of low-level pure functional static constructs. MPL is a widely use library for metaprogramming in C++ with a collection of extensible compile-time algorithms, typelists, and metafunctions. Typelists encapsulate zero or more C++ types (programmer-defined or otherwise) in a way that can be manipulated at compile-time using MPL metafunctions.

A.6 Overview of C++ Concepts

C++ Concepts [57] have been developed to address the problem of late type-checking of templates during compilation. Concepts express the syntactic and semantic behavior of types and constrain the type parameters in a C++ template, which are otherwise unconstrained. Concepts allow separate type-checking of template definitions from their uses, which makes templates easier to use and easier to compile. The set of constraints on one or more types are referred to as *Concepts*. Concepts describe not only the functions and operators that the types must support but also other accessible types called *associated types*. The types that satisfy the requirements of a concept are said to *model* that concept. When a concept constrained C++ template is instantiated with a type that does not model the concept, an error message indicating the failure of the concept and the type that violates it are shown at the call site in the source code. An experimental support for C++ Concepts has been implemented in the ConceptGCC [55] compiler.

For example, the following concept called `LessThanComparable` defines a constraint that the "less than" operator $<$ be defined for objects of type T.

```
concept LessThanComparable <typename T> {
  bool operator < (T, T);
};
template <typename T>
  requires LessThanComparable<T>
void sort (T * begin, T * end) { ... }
```

A generic function called `sort` *requires* this concept to be satisfied by the range of objects pointed to by the `begin` and `end` pointers. If the `sort` function is instantiated with a type that does not model the `LessThanComparable` concept (*e.g.*, the native **`void`** type), a short error message indicating the failure of the concept along with the type that causes it is shown at the call site in the source code.

APPENDIX B

LIST OF PUBLICATIONS

Research on CQML, GRAFT, LEESA, and the group-failover protocol has led to the following journal, conference, and workshop publications.

B.1 Refereed Journal Publications

1. Friedhelm Wolf, Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt, "Supporting Component-based Failover Units in Middleware for Distributed Real-time and Embedded Systems," *Elsevier Journal of Software Architectures: Embedded Software Design, Special Issue on Embedded and Real-time Systems, 2010.*

2. Jules White, James Hill, Jeff Gray, Sumant Tambe, Douglas C. Schmidt, Anirrudha Gokhale, "Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques," *IEEE Software Special Issue on Domain-Specific Languages and Modeling, July-August 2009.*

B.2 Refereed Conference Publications

1. Sumant Tambe and Aniruddha Gokhale, "LEESA: Embedding Strategic and XPath-like Object Structure Traversals in C++," *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL WC), 2009.*

2. Sumant Tambe, Akshay Dabholkar, Aniruddha Gokhale, "CQML: Aspect-oriented Modeling for Modularizing and Weaving QoS Concerns in Component-based Systems," *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), 2009.*

3. Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale, “Fault-tolerance for Component-based Systems - An Automated Middleware Specialization Approach’,” *Proceedings of the International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), 2009.*

4. Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill, and Douglas C. Schmidt, “Adaptive Failover for Real-time Middleware with Passive Replication,” *Proceedings of the 15th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2009.*

5. 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,” *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), 2008.*

6. Sumant Tambe, Jaiganesh Balasubramanian, Aniruddha Gokhale, “MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems,” *Proceedings of the International Service Availability Symposium (ISAS), 2007.*

7. James Hill, Sumant Tambe, Aniruddha Gokhale, “Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems,” *Proceedings of Engineering of Computer Based Systems Conference (ECBS), 2007.*

B.3 Refereed Workshop Publications

1. Aniruddha Gokhale, Akshay Dabholkar, and Sumant Tambe, “Towards a Holistic Approach for Integrating Middleware with Software Product Lines Research,” *Proceedings of the GPCE/OOPSLA workshop on Modularization, Composition and Generative Techniques in Product Line Engineering (McGPLE), 2008.*

2. Sumant Tambe, Aniriddha Gokhale, “An Embedded Declarative Language for Hierarchical Object Structure Traversal,” *2nd International Workshop on Domain-Specific Program Development (DSPD), GPCE 2008.*

3. Sumant Tambe, Akshay Dabholkar, Aniruddha Gokhale, Amogh Kavimandan, “Towards A QoS Modeling and Modularization Framework for Component-based Systems,” *EDOC workshop on Advances in Quality of Service Management (AQuSerM) 2008.*

4. Aniruddha Gokhale, Sumant Tambe, Larry Dowdy and Gautam Biswas, “Towards High Confidence Cyberphysical Systems for Intelligent Transportation Systems,” *Position paper in National Workshop on High-Confidence Automotive Cyber-Physical Systems, April 2008.*

5. Sumant Tambe, Jaiganesh Balasubramanian , Aniruddha Gokhale, “Model-Driven Engineering of Fault Tolerance in Enterprise Distributed Real-time and Embedded Systems,” *Proceedings of OMG Real-time Systems Workshop (RTWS), 2006.*

B.4 Technical Reports

1. Sumant Tambe, Aniruddha Gokhale, “Toward Native XML Processing Using Multi-paradigm Design in C++,” *Technical Report ISIS-10-105, Institute for Software Integrated Systems, Vanderbilt University, April 2010.*

B.5 Submitted for Publication

1. Sumant Tambe, Aniruddha Gokhale, “Rectifying Orphan Components using Group-failover in Distributed Real-time Embedded Systems,” *Submitted to the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011.*

REFERENCES

[1] XML Data Binding for C++. http://www.codesynthesis.com/products/xsd. URL http://www.codesynthesis.com/products/xsd.

[2] Jan Oyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, Oslo, March 2001.

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321227255.

[4] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-Oriented Fault Tolerance for Real-Time Embedded Systems. In *ACP4IS '08: Proceedings of the 7th workshop on Aspects, components, and patterns for infrastructure software*, 2008. doi: http://doi.acm.org/10.1145/1233901.1233908.

[5] Cunha Alcino and Visser Joost. Transformation of Structure-Shy Programs: Applied to XPath Queries and Strategic Functions. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 11–20, 2007.

[6] Ruben Alexandersson and Peter Ohman. Implementing Fault Tolerance Using Aspect Oriented Programming. In *Latin American Symposium on Dependable Computing (LADC)*, volume 4746, pages 57–74. Springer, 2007.

[7] Anders Hejlsberg, Don Box et al. Language Integrated Query (LINQ). URL http://en.wikipedia.org/wiki/Language_Integrated_Query.

[8] Algirdas Avizienis and Chen Liming. On the Implementation of N-Version Programming for Software Fault-Tolerance During Prograrnm Execution. *Compsac*, pages 149–155, 1977.

[9] Markus Bajohr and Tiziana Margaria. High Service Availability in MaTRICS for the OCS. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008*, volume 17 of *Communications in Computer and Information Science*. Springer, 2008. ISBN 978-3-540-88478-1.

[10] Markus Bajohr and Tiziana Margaria. Model-Driven Self-Reconfiguration for Highly Available SOAs. In *EASE '09: Proceedings of the 2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 13–22, 2009. ISBN 978-0-7695-3623-1.

[11] 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.

[12] Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill, and Douglas C. Schmidt. Adaptive Failover for Real-time Middleware with Passive Replication. In *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS '09)*, pages 118–127, San Francisco, CA, April 2009.

[13] Jaiganesh Balasubramanian, Aniruddha Gokhale, Friedhelm Wolf, Abhishek Dubey, Chenyang Lu, Chris Gill, and Dougas C. Schmidt. Resource-Aware Deployment and Configuration of Fault-tolerant Real-time Systems. In *Proceedings of the 16th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS '10)*, Stockholm, Sweden, April 2010.

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

[15] 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*, pages 190–199, 2005.

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

[17] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, pages 36–47, 2007.

[18] Claudio Basile, Zbigniew Kalbarczyk, and Ravi Iyer. A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. *International Conference on Dependable Systems and Networks*, 0:149, 2003.

[19] Pete Becker. Standard for Programming Language C++. Working Draft, N3126=10-0116, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Aug. 2010.

[20] Andrey Berlizev, Alfredo Capozucca, Barbara Gallina, Nicolas Guelfi, Patrizio Pelliccione, and Alexander. CORRECT Project Annual Activity Report 2005. Technical report, Faculty of Science, Technology and Communication, Luxembourg-Kirchberg, 2006.

[21] Simona Bernardi, Jose Merseguer, and Dorina Petriu. A Dependability Profile within MARTE. *Journal of Software and Systems Modeling*, pages 872–923, 2009.

[22] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-time Specification for Java*. Addison-Wesley, 2000.

[23] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-backup Approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.

[24] Alfredo Capozucca, Barbara Gallina, Nicolas Guelfi, Patrizio Pelliccione, and Alexander Romanovsky. CORRECT - Developing Fault-Tolerant Distributed Systems. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 64(1), 2006. URL www.ercim.org/publication/Ercim_News/enw64/guelfi.html.

[25] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43:225–267, 1995.

[26] Denis Conan, Erik Putrycz, Nicolas Farcet, and Miguel DeMiguel. Integration of Non-Functional Properties in Containers. *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[27] James Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.

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

[29] Krzysztof Czarnecki, John O'Donnell, Jorg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain Specific Program Generation*, pages 51–72, 2004.

[30] B. Dasarathy, S. Gadgil, R. Vaidhyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot. Network QoS Assurance in a Multi-Layer Adaptive Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework. In *RTAS 2005*, San Francisco, CA, March 2005.

[31] B. Dasarathy, S. Gadgil, R. Vaidyanathan, K. Parmeswaran, B. Coan, M. Conarty,

and V. Bhanot. Network qos assurance in a multi-layer adaptive resource management scheme for mission-critical applications using the corba middleware framework. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 246–255, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.34.

[32] Roger Barga David, David Lomet, Stelios Paparizos, Haifeng Yu, and Sirish Ch. Persistent Applications via Automatic Recovery. In *7th International Database Engineering and Applications Symposium (IDEAS 2002)*, pages 258–267, 2002.

[33] Miguel A. de Miguel. Integration of QoS Facilities into Component Container Architectures. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 394, Washington, DC, USA, 2002.

[34] 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.

[35] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-based Development of Embedded Systems: The SysWeaver Approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 231–242, San Jose, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2516-4. doi: http://dx.doi.org/10.1109/RTAS.2006.30.

[36] Eliezer Dekel and Gera Goft. ITRA: Inter-Tier Relationship Architecture for End-to-end QoS. *Journal of Supercomputing*, 28(1):43–70, 2004.

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

[38] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building Adaptive Distributed Applications with Middleware and Aspects. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development (AOSD)*, pages 66–73, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3.

[39] E. Magyari and A. Bakay and A. Lang and T. Paka and A. Vizhanyo and A. Agrawal and G. Karsai. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling*, October 2003.

[40] M. El-Gendy, A. Bose, S.-T. Park, and K.G. Shin. Paving the first mile for qos-dependent applications and appliances. In *IWQoS '04: Proceedings of the 12th International Workshop on Quality of Service*, pages 245–254, Washington, DC,

USA, June 2004. IEEE Computer Society. doi: 10.1109/IWQOS.2004.1309390.

[41] M.A. El-Gendy, A. Bose, and K.G. Shin. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE*, 91(7):1086–1104, July 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2003.814615.

[42] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Pearson Education, Upper Saddle River, NJ, USA, 2005.

[43] Huascar Espinoza, Hubert Dubois, Sebastien Gerard, Julio Medina, Dorina Petriu, and Murray Woodside. Annotating UML Models with Non-functional Properties for Quantitative Analysis. In *Model Driven Engineering Languages and Systems, 9th International Conference (MoDELS 2006)*, volume 3844, pages 79–90, 2006.

[44] Pascal Felber and Priya Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 737–754, 2002.

[45] Pascal Felber and Priya Narasimhan. Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. *IEEE Transactions on Computers*, 54(5): 497–511, May 2004.

[46] Svend Frolund and Rachid Guerraoui. Transactional Exactly-Once. *Technical report, Hewlett-Packard Laboratories*, 1999.

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

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

[49] Laszlo Gonczy, Daniel Varro, and Gonczy Varro. Modeling of Reliable Messaging in Service Oriented Architectures. In *Proc. Int. Workshop on Web Service Modeling and Testing (WS-MATE 2006)*, 2006.

[50] Sathish Gopalakrishnan and Marco Caccamo. Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 199–207, San Jose, CA, USA, April 2006. IEEE Computer Society. ISBN 0-7695-2516-4. doi: http://dx.doi.org/10.1109/RTAS.2006.43.

[51] Jeff Gray and Gabor Karsai. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In *36th Hawaiian International Conference*

on System Sciences (HICSS), pages 325–334, 2003.

[52] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, 2003.

[53] 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.

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

[55] Douglas Gregor. ConceptGCC: Concept Extensions for C++. http://www.generic-programming.org/software/ConceptGCC, Aug. 2008.

[56] Douglas Gregor and Jaakko Järvi. Variadic Templates for C++0x. *Journal of Object Technology, Special Issue OOPS Track at SAC 2007*, 7:31–51, 2008.

[57] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, 2006.

[58] G.Rodrigues. A Model Driven Approach for Software Systems Reliability. In *In the proceedings of the 26th ICSE/Doctoral Symposium*. ACM Press, May 2004.

[59] 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.

[60] Rachid Guerraoui and André Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

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

[62] Maurice Herlihy and Martin McKendry. Timestamp-Based Orphan Elimination. *IEEE Transaction on Software Engineering*, 15(7):825–831, 1989. ISSN 0098-5589.

[63] J. Herrero, F. Sanchez, and M. Toro. Fault tolerance AOP approach. In *Workshop on Aspect-Oriented Programming and Separation of Concerns*, 2001.

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

[65] Ricardo Jimenez-peris, Marta Patino-Martinez, Sergio Arevalo, and Juan Carlos. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, pages 164–173, 2000.

[66] Jingwen Jin and Klara Nahrstedt. On Exploring Performance Optimizations in Web Service Composition. In *Middleware*, pages 115–134, 2004.

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

[68] 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.

[69] Heine Kolltveit and Svein olaf Hvasshovd. Preventing orphan requests by integrating replication and transactions. In *11th East-European Conference on Advances in Databases and Information Systems, ADBIS*. Springer, 2007.

[70] Boris Kolpackov. An Introduction to XML Data Binding in C++. *The C++ Source*, March 2007.

[71] Fabio Kon, Tomonori Yamane, Christopher Hess, Roy Campbell, and M. Dennis Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, pages 15–30, San Antonio, Texas, February 2001.

[72] L. Zhang and S. Berson and S. Herzog and S. Jamin. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification, September 1997.

[73] Ralf Lämmel. Scrap Your Boilerplate with XPath-like Combinators. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 137–142, 2007.

[74] Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *Proceedings of Practical Aspects of Declarative Languages*, pages 137–154. Springer-Verlag, 2002.

[75] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. Draft; Available at `http://homepages.cwi.nl/~ralf/eosp`, October15 2002.

[76] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive

programming. In *Proceedings of Aspect-Oriented Software Development (AOSD)*, pages 168–177, 2003.

[77] 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.

[78] 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.

[79] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[80] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of Object Structures: Specification and Efficient Implementation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(2):370–412, 2004. ISSN 0164-0925.

[81] Barbara Liskov, R Scheifler, E Walker, and W Weihl. Orphan Detection. *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 2–7, 1987.

[82] Jane W. S. Liu. *Real-time Systems*. Prentice Hall, New Jersey, 2000.

[83] Brett McLaughlin. *Java and XML data binding*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. ISBN 0-596-00278-5.

[84] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[85] Lydia Michotte, Robert France, and Franck Fleurey. Modeling and Integrating Aspects into Component Architectures. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 181, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2891-0.

[86] Shivajit Mohapatra, Radu Cornea, Hyunok Oh, Kyoungwoo Lee, Minyoung Kim, Nikil D. Dutt, Rajesh Gupta, Alexandru Nicolau, Sandeep K. Shukla, and Nalini Venkatasubramanian. A Cross-Layer Approach for Power-Performance Optimization in Distributed Mobile Systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2005.

[87] J. E. Moreira and V. K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. *IBM Journal of Research and Development*, 41(3):303–330, 1997.

[88] Gustav Munkby, Andreas Priesnitz, Sibylle Schupp, and Marcin Zalewski. Scrap++: Scrap Your Boilerplate in C++. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 66–75, 2006.

[89] Priya Narasimhan. MEAD: Support for Real-time Fault-Tolerant Middleware. In *OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems*, Washington, DC, July 2003. Object Management Group.

[90] Priya Narasimhan, Tudor Dumitras, Aaron M. Paulos, Soila M. Pertet, Charlie F. Reverte, Joseph G. Slember, and Deepti Srivastava. MEAD: Support for Real-Time Fault-Tolerant CORBA. *Concurrency - Practice and Experience*, 17(12):1527–1545, 2005.

[91] Object Management Group. *Transaction Services Specification*. Object Management Group, OMG Document formal/97-12-17 edition, December 1997.

[92] Object Management Group. *Fault Tolerant CORBA Specification*. Object Management Group, OMG Document orbos/99-12-08 edition, December 1999.

[93] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Version 3.0*. Object Management Group, 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. *Lightweight CORBA Component Model RFP*. Object Management Group, realtime/02-11-27 edition, November 2002.

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

[97] *UML Profile for Schedulability, Performance, and Time Specification*. Object Management Group, Final Adopted Specification ptc/02-03-02 edition, March 2002.

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

[99] *Object Transaction Service formal/03-09-02*. Object Management Group, 1.4 edition, 2003.

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

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

[102] *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission*. Object Management Group, OMG Document realtime/03-05-02 edition, May 2003.

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

[104] *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*. Object Management Group, OMG Document realtime/05-02-06 edition, May 2005.

[105] *Unified Modeling Language Infrastructure, v2.1.2*. Object Management Group, OMG Document formal/2007-11-04 edition, November 2007.

[106] 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.

[107] Johan Ovlinger and Mitchell Wand. A Language for Specifying Recursive Traversals of Object Structures. *SIGPLAN Notes*, 34(10):70–81, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/320385.320391.

[108] Stefan Poledna. Replica Determinism in Distributed Real-time Systems: A Bbrief Survey. *Real-Time Syst.*, 6(3):289–316, 1994.

[109] Stefan Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, Technical University of Vienna, Vienna, Austria, 1994.

[110] Andreas Polze, Janek Schwarz, and Miroslaw Malek. Automatic Generation of Fault-Tolerant CORBA-Services. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS)*, 2000.

[111] Portland Pattern Repository, Cunningham and Cunningham, Inc. Hierarchical Visitor Pattern, 2005. URL http://c2.com/cgi/wiki?HierarchicalVisitorPattern.

[112] I. Pyarali, D.C. Schmidt, and R.K. Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *Proceedings of the IEEE, Special Issue on Real-time System*, 91(7):1070–1085, July 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2003.814616.

[113] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali. Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In *Proc. of Middleware'03*, Rio de Janeiro, Brazil, June

2003. IFIP/ACM/USENIX.

[114] Ronald Bourret. XML Data Binding Resources. URL http://www.rpbourret.com/xml/XMLDataBinding.htm.

[115] Paul Rubel, Joseph Loyall, Richard Schantz, and Matthew Gillen. Adding Fault-Tolerance to a Hierarchical DRE System. In *Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference (DAIS)*, pages 303–308, 2006.

[116] Juan Carlos Ruiz, Marc-Olivier Killijian, Jean-Charles Fabre, and Pascale Thévenod-Fosse. Reflective Fault-Tolerant Systems: From Experience to Challenges. *IEEE Transaction on Computers*, 52(2):237–254, 2003. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.2003.1176989.

[117] Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, and Joseph Loyall. An Object-level Gateway Supporting Integrated-Property Quality of Service. *International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, 00:223, 1999. doi: doi.ieeecomputersociety.org/10.1109/ISORC.1999.776381.

[118] Richard D. Schlichting and Fred B. Schneider. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Transaction on Computer Systems*, 1(3):222–238, 1983. ISSN 0734-2071. doi: doi.acm.org/10.1145/357369.357371.

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

[120] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. In *CrossTalk - The Journal of Defense Software Engineering*, pages 10–16, Hill AFB, Utah, USA, nov 2001. Software Technology Support Center.

[121] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. *Draft Proceedings of Implementation of Functional Languages*, pages 186–205, 2003.

[122] Diego Sevilla, Jose Garcia, and Antonio Gomez. Aspect-Oriented Programing Techniques to support Distribution, Fault Tolerance, and Load Balancing in the CORBA(LC) Component Model. *International Symposium on Network Computing and Applications (NCA 2007)*, 00:195–204, 2007.

[123] Praveen Kaushik Sharma, Joseph P. Loyall, George T. Heineman, Richard E. Schantz, Richard Shapiro, and Gary Duzan. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In *CoopIS/DOA/ODBASE (2)*,

pages 1208–1224, Agia Napa, Cyprus, 2004. Springer.

[124] 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.

[125] David C. Sharp, Edward Pla, Kenn R. Luecke, and Ricardo J. Hassan II. Evaluating Real-time Java for Mission-Critical Large-Scale Embedded Systems. In *IEEE Real-time and Embedded Technology and Applications Symposium*, Washington, DC, May 2003. IEEE Computer Society.

[126] Jeremy G. Siek and Andrew Lumsdaine. C++ Concept Checking. *Dr. Dobb's Journal*, 26(6):64–70, 2001. ISSN 1044-789X.

[127] Fabio Simeoni, David Lievens, Richard Connor, and Paolo Manghi. Language Bindings to XML. *IEEE Internet Computing*, pages 19–27, 2003.

[128] Joseph Slember and Priya Narasimhan. Using Program Analysis to Identify and Compensate for Nondeterminism in Fault-Tolerant, Replicated Systems. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 251–263, 2004.

[129] Joseph Slember and Priya Narasimhan. Living with Nondeterminism in Replicated Middleware Applications. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 81–100, 2006.

[130] Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). `http://www.cs.ua.edu/~gray/Research/C-SAW`, University of Alabama, Tuscaloosa, AL.

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

[132] 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.

[133] Stefan Pleisch and Arnas Kupsys and Andre Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of 22nd Symposium on Reliable Distributed Systems*, pages 119–129, 2003.

[134] Jun Sun. *Fixed-Priority End-to-End Scheduling in Distribured Real-time Systems.*

PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champagne, 1997.

[135] 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.

[136] Diana Szentivany and Simin Nadjm-Tehrani. Aspects for improvement of performance in fault-tolerant software. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 283–291. IEEE Computer Society, 2004.

[137] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[138] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997. ISBN 0201178885.

[139] Francois Taiani and Jean-Charles Fabre. A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 270–279, 2005.

[140] Sumant Tambe and Aniruddha Gokhale. An Embedded Declarative Language for Hierarchical Object Structure Traversal. In *Proceedings of the 2nd International ACM GPCE Workshop on Domain-Specific Program Development (DSPD '08)*, Nashville, TN, October 2008.

[141] Sumant Tambe, Jaiganesh Balasubramanian, Aniruddha Gokhale, and Thomas Damiano. MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems. In *Proceedings of the International Service Availability Symposium (ISAS)*, volume 4526, pages 127–144, Durham, New Hampshire, USA, 2007. Springer.

[142] Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale. Generative Techniques to Specialize Middleware for Fault Tolerance. In *Proceedings of the* 12th *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[143] *Architecture Analysis and Design Language (AS5506)*. The Society for Automotive Engineers (SAE), 2004.

[144] *Architecture Analysis and Design Language (AADL) Annex vol. 1, Annex E: Error Model Annex, International Society of Automotive Engineers,*. The Society for Automotive Engineers (SAE), 2006.

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

[146] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[147] Eelco Visser, Zineelabidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, 1998.

[148] Joost Visser. Visitor Combination and Traversal Control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 270–282, 2001.

[149] Hiroshi Wada, Junichi Suzuki, and Katsuva Oba. A Model-Driven Development Framework for Non-Functional Aspects in Service Oriented Architecture . In *International Journal of Web Services Research*, volume 5, pages 1–31, 2008.

[150] Dean Wampler and Tony Clark. Guest Editors' Introduction: Multiparadigm Programming. *IEEE Software*, 27:20–24, 2010. ISSN 0740-7459.

[151] P. Wang, Y. Yemini, D. Florissi, and J. Zinky. A distributed resource controller for qos applications. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP*, pages 143–156, Los Alamitos, CA, USA, 2000. IEEE Computer Society. doi: 10.1109/NOMS.2000.830381.

[152] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Engineering and Simulation. *Journal of Software and System Modeling*, 7(1):3–23, 2008.

[153] Friedhelm Wolf, Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt. Supporting Component-based Failover Units in Middleware for Distributed Real-time and Embedded Systems. *Journal of Software Architectures: Embedded Software Design, Special Issue on Embedded and Real-time*, November 2010.

[154] World Wide Web Consontium (W3C). XML Path Language (XPath), Version 2.0, W3C Recommendation. http://www.w3.org/TR/xpath20, Jan. 2007.

[155] World Wide Web Consontium (W3C). XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Recommendation. http://www.w3.org/TR/xquery-semantics, Jan. 2007.

[156] Yair Amir and Claudiu Danilov and Jonathan Robert Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 327–336, June 2000.

[157] A. Zarras, P. Vassiliadis, and V. Issarny. Model-Driven Dependability Analysis of Web Services. In *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, October 2004.

[158] 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.