PRINCIPLES FOR SAFE AND AUTOMATED MIDDLEWARE SPECIALIZATIONS FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

By

Akshay V. Dabholkar

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2012

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Douglas Schmidt

Dr. Gabor Karsai

Dr. Jeffrey Gray

Dr. Janos Sztipanovits

To Aai, Daddy and Poonam for their unconditional love, support and encouragement
And ofcourse Riya and Simba, the joys of our lives

ACKNOWLEDGMENTS

Having a tremendously inquisitive mind about everything I see and do, I was always sure about pursuing a research oriented career since my junior years of undergraduates studies. However, I didn't know what was required in order to achieve the coveted PhD degree and when I embarked on that journey, I was everything but certain. Treading through the highs and lows of being a PhD student wanting to fulfill his ambitions, there are many people I am thankful for believing in me at times when I was not feeling motivated and for guiding me down the right path. Without them, I would never have realized that its not the destination that is important but the journey is what you take away in terms of your attitude towards challenges you face in your career and life.

First and foremost, I would like to thank my advisor Dr. Aniruddha Gokhale for providing me with support and encouragement not only professionally as a mentor but also personally as a friend. It is the friendly atmosphere created by him that made the entire PhD experience bearable and provided me impetus to fight on and conquer the challenges posed along the way. Andy, as he is fondly called, spent many hours discussing research ideas, reviewing papers, presentations and ensuring a good foundation based on independent thinking was laid to achieve the goals. I am also thankful to him for having confidence in me and supporting me during difficult times while trying to forge research ideas and getting them published.

I would also like to thank Dr. Douglas Schmidt (Doug), for providing me the initial opportunity to be a part of the reputed Distributed Object Computing (DOC) group at Vanderbilt. Doug's pioneering work in distributed middleware has been the core source of inspiration for me since my undergraduate days that motivated me to pursue a doctoral degree in this area in the first place. Through his teaching, I learned the nuances of good software design and distributed middleware that will stay with me for the rest of my professional life.

I would also like to express my gratitude to the rest of my committee members, Dr. Janos Sztipanovits, Dr. Gabor Karsai, and Dr. Jeff Gray for agreeing to serve on my qualifier and dissertation committees. I am especially grateful to Jeff for reviewing my dissertation and managing to find time to remotely attend my qualifier and dissertation defenses.

I have been fortunate to interact with some very motivated and talented past and present colleagues: Krishnakumar Balasubramanian, Nishanth Shankaran, Amogh Kavimandan, Chetan Kulkarni, Abhishek Dubey. Moreover, the journey would not have been enjoyable and survivable without the company of Jaiganesh Balasubramanian (Jai), Nilabja Roy, and Sumant Tambe who were an integral part of our daily coffee sessions, that brewed up some of the most hilarious jokes on graduate student life. We concocted many that were worthy of deserving a DOC group version of the PhD Comics.

Finally, I would like to especially thank my mother and father, Minal and Vishwas, and my dear wife, Poonam, for their unconditional love, unwavering support and encouragement over the years throughout the uphill journey of my PhD. And of course our beautiful baby girl, Riya, the joy of our life. Also our lovely dog, Simba, our 1st child, for the great times everyday. To them, I dedicate this thesis.

Akshay V. Dabholkar

Vanderbilt University

April 2, 2012

ABSTRACT

Developing distributed applications, particularly those for distributed, real-time and embedded (DRE) systems, is a difficult and complex undertaking due to the need to address four major challenges: the complexity of programming interprocess communication, the need to support a wide range of services across heterogeneous platforms and promote reuse, the need to efficiently utilize resources, and the need to safely adapt to runtime conditions. The first two challenges are addressed to a large extent by standardized, general-purpose middleware. However, the need to support a large variety of applications in different domains has resulted in very feature-rich implementations of these standardized middleware. Consequently, this feature-richness acts counter productive to resolving the remaining two challenges; instead it incurs excessive memory footprint and performance overhead, as well as increased cost of testing and maintenance. Moreover, despite the richness in general-purpose features, middleware often lacks application-specific behavior that is needed to adapt to runtime conditions including faults.

To address the four challenges all at once while leveraging the benefits of general-purpose middleware, this dissertation describes a scientific approach to specializing the middleware. To enable better comprehension, easier validation and to promote reuse, the dissertation presents a three dimensional taxonomy to document recurring specializations, and assess the strengths and weaknesses of the documented techniques. The principles of separation of concerns are used in the context of this taxonomy to define six stages of a middleware specialization process lifecycle. Finally, to overcome the accidental complexities stemming from the manual use of specialization techniques, such as aspect-oriented programming (AOP), feature-oriented programming (FOP), and reflection, the six-stage specialization process has been codified resulting in concrete tool artifacts that automate the specialization process for different requirements.

The tooling resulting from this dissertation includes (1) FORMS (Feature Oriented

Reverse Engineering based Middleware Specializations), which provides coarse-grained middleware feature pruning through a decision tree based reasoning of desired middleware features and a novel reverse-engineering algorithm, (2) GeMS (Generative Middleware Specializations), which provides fine-grained middleware feature pruning through an automated process that deduces the context for specializations through application invariant properties and subsequently optimizes the middleware design patterns and frameworks through generative source-to-source transformations, (3) GrAFT (Generative Aspects for Fault-Tolerance), which provides fine-grained middleware feature augmentation by weaving application-specific reliability concerns in system artifacts through model-to-text, model-to-code transformations, and (4) SafeMAT (Safe Middleware Adaptation for Real-Time Fault-Tolerance), which enables safe middleware adaptation to runtime failures while improving predictability and resource utilization within the hard real-time constraints.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Emerging Trends and Technologies

A large variety of applications and application product lines such as those found in avionics [116], telecommunication call processing, multimedia streaming video, industrial automation, multi-satellite missions [122], shipboard computing [113] and mission-critical computing environments, have varied requirements such as transparent distribution, interoperability, real-timeliness, predictability, fault tolerance, fast recovery, high throughput, high availability, etc. As a result these distributed, real-time and embedded (DRE) systems are leveraging general-purpose middleware in their design and implementation due to many benefits such as lowering the time to market and hiding accidental complexities [18] associated with a particular domain. Traditionally, middleware hides the underlying details of interprocess communication and heterogeneous technologies from the application developers using a *"black-box"* paradigm such as encapsulation in object-oriented programming and through the use of elegant design abstractions such as design patterns and frameworks as shown in figure 1.

Middleware like Real-time Common Object Request Broker Architecture (RT-CORBA) [88] and the Real-Time Specification for Java (RTSJ) [17] enables these systems to realize long shelf lives by shielding these systems from the constant evolution in the underlying operating systems and hardware resources. Research in middleware over the past decade [16, 114, 143] has significantly advanced the quality and feature-richness of general-purpose middleware, such as J2EE, .NET, CORBA, and DDS. The economic benefits of middleware are significant with up to 50% decrease reported in software development time and costs [100].

Despite these benefits, general-purpose middleware poses numerous challenges when

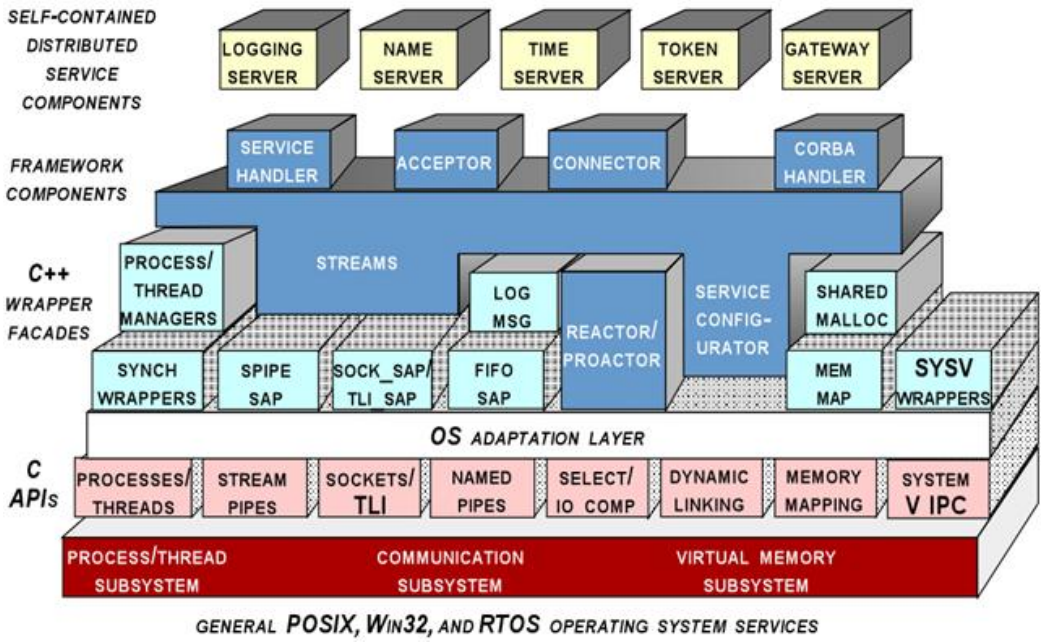


Figure 1: Middleware Features (ACE [110])

developing Distributed Realtime and Embedded Systems (DRE). First, owing to the stringent demands of DRE on quality of service (QoS) (*e.g.* real-time response in industrial automation) and/or constraints on resources (*e.g.* memory footprint of embedded medical devices monitoring a patient), the feature-richness and flexibility of general-purpose middleware becomes a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains and/or energy savings. Second, general-purpose middleware lack *out of the box* support for modular extensibility of both domain-specific and domain-independent features within the middleware without unduly expending extensive manual efforts at retrofitting these capabilities. For example, DRE in two different domains as in industrial automation and automotive may require different forms of domain-specific fault tolerance and failover support. Arguably, it is not feasible for general-purpose middleware developers to have accounted for these domain-specific requirements ahead-of-time in their design. Doing so would in fact contradict the design

goals of middleware, which aim to make them broadly applicable to a wide range of domains, i.e., general-purpose. The figure 2 shows the antagonistic design forces between domain-specific applications and general-purpose middleware.

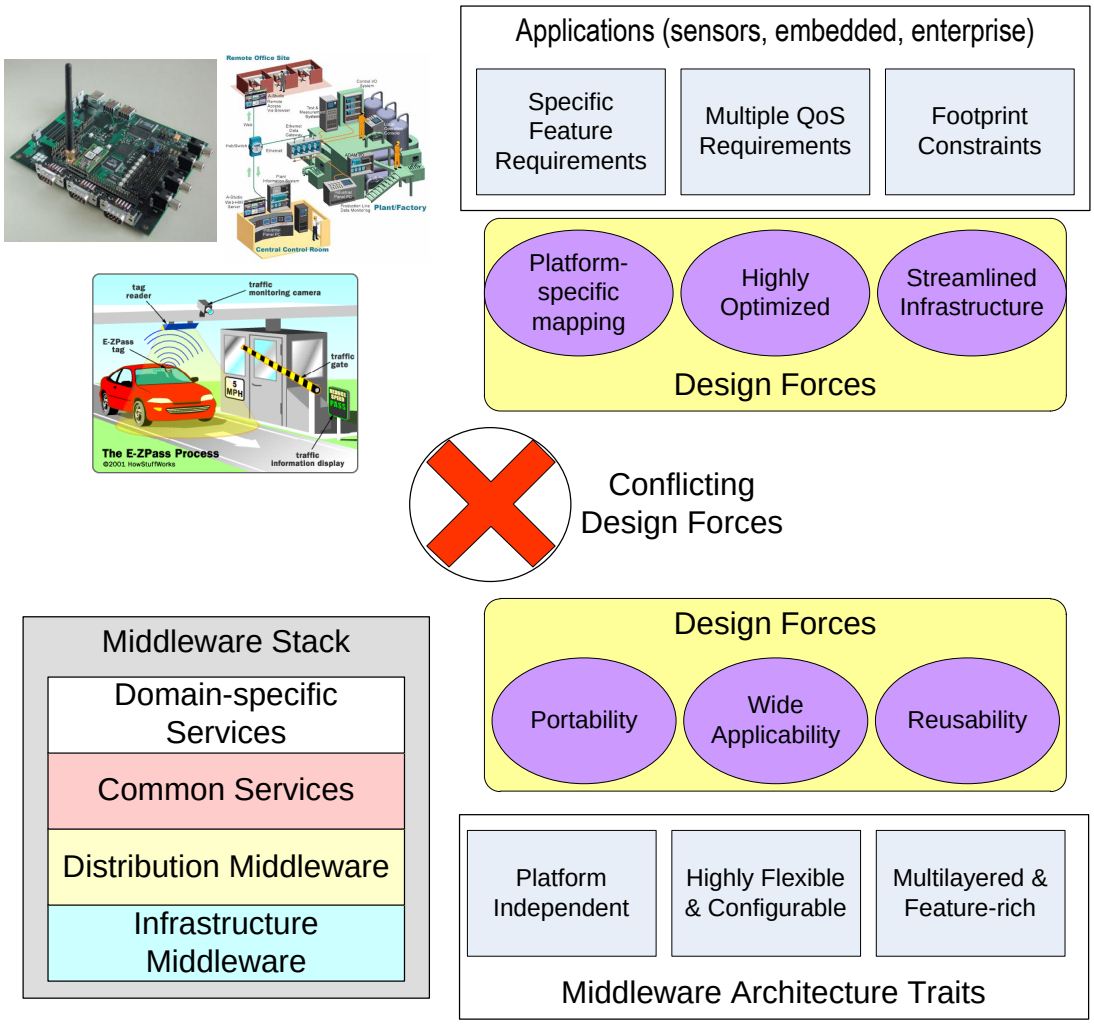


Figure 2: Antagonistic Design Forces

Developing proprietary and customized middleware solutions for individual DRE is not a feasible alternative due to the excessive costs of development, maintenance and testing.

Moreover, such solutions often tend to reinvent many solutions that already exist in general-purpose middleware. Due to the large number of application domains addressed by DOC middleware, the frameworks that are provided are feature rich. Any particular application, though, is likely to only use a fraction of these features. This problem is especially relevant for embedded applications, where memory and other resources are already at a premium which results into simultaneous stringent QoS requirements. As a result, developers are often faced with either reinventing pieces of an middleware, custom tailored to their needs, or they are faced with the daunting task of refactoring an existing middleware to obtain an appropriate subset of that application's functionality. In either case, subsequent development, maintenance and testing of the application becomes more complex, due to the effects of future revisions on all of the derived subsets. Current trends and economies of scale in software development actually call for extensive reuse and rapid assembly of application functionality from off-the-shelf infrastructure and application components. Third, legacy applications that are deployed in safety-critical domains such as avionics, automotive, industrial automation have a long shelf life. Due to their safety-critical nature, these applications are over-provisioned in terms of the resources to handle failures even in the worst-case scenarios, and are closed in nature with precisely specified hard real-time Quality of Service (QoS) requirements of schedulability, timeliness, processor and memory allocation, and reliability. Due to their closed nature, they are severely constrained in terms of their ability to support changing requirements without the right tools and techniques.

Addressing this dilemma requires an approach that will enable DRE developers to derive the benefits of general-purpose middleware, however, without incurring the overhead of unwanted features while seamlessly allowing domain-specific extensions. Such an approach must be rooted in scientific principles, which is particularly important for DRE applications due to the need to formally verify the correctness of different systemic properties of DRE. We call such an approach as *Middleware Specialization*. Although traditional

middleware solves these problems to some extent, it is limited in its ability to support specializations and adaptations. Middleware adaptation is a separate issue from specialization and focus is on middleware specialization techniques in this dissertation.

I.2 Overview of Research Challenges

DRE systems based on the .NET, J2EE, and CORBA standards are often subjected to both stringent certification and cost issues. Therefore, it is important that any modification to the middleware sources be retrofitted with minimal to no changes to the middleware portability, standard APIs interfaces, application software implementations, while preserving interoperability wherever possible. Otherwise such specialization approaches obviate the benefits accrued from using standards-based middleware. Additionally the accidental complexity from manually applying such approaches to mature middleware implementations renders the specializations tedious and error prone to implement.

Most prior efforts at specializing middleware (and other system artifacts) [22, 51, 70, 96, 136, 142] often require manual efforts in identifying opportunities for specialization and realizing them on the software artifacts. At first glance it may appear that these manual efforts are expended towards addressing problems that are purely accidental in nature. A close scrutiny, however, reveals that system developers face a number of inherent complexities as well, which stem from the following reasons:

I.2.1 Spatial disparity between horizontally decomposed middleware and vertical domain-level concerns

Middleware is traditionally designed using object-oriented (OO) principles, which enforce a *horizontal decomposition* of its capabilities into layers comprising class hierarchies as shown in figure 3. This design is, however, not suited for specializing middleware since

domain concerns tend to map along the vertical dimension, which are shown to crosscut the OO class hierarchies [45] hence necessitating *vertical decomposition*.For example, in OO-based middleware implementations of Real-time CORBA (RTCORBA) [94], the implementation of features related to handling requests at a fixed priority (called the `SERVER_DECLARED` model) or allowing priorities to be propagated from task to task (called the `CLIENT_PROPAGATED_PRIORIY` models) crosscut multiple functional modules such as the object request broker (ORB), the portable object adapter (POA), and request demultiplexing and dispatching modules. Since the two priority models are mutually exclusive, only one configuration can be valid along the critical path between tasks of a DRE system. Thus, any transformation to prune the logic for the unused priority model must necessarily involve modifying several different classes that implement these different modules. Therefore, there is a need to reason about the middleware features induced by the application requirements and the configurations of those features.

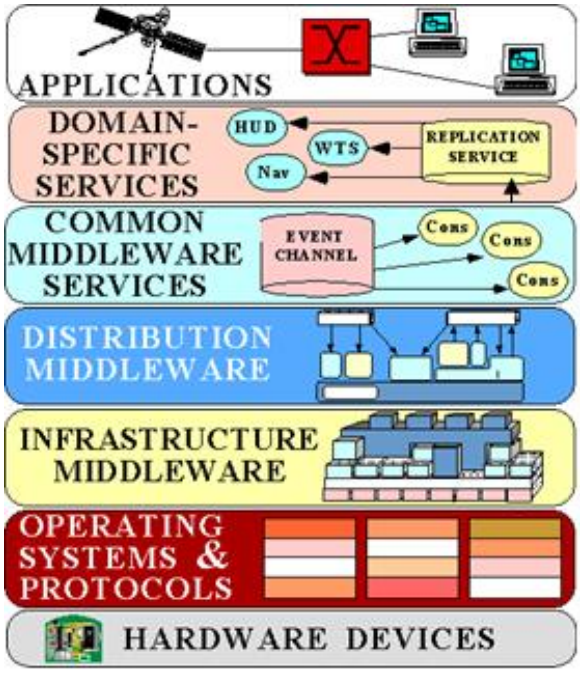


Figure 3: Middleware Layers

I.2.2 Lack of a priori knowledge of specialization requirements due to temporal separation of application lifecycle phases

DRE systems often involve a well-defined application development lifecycle comprising the design, composition, deployment, and configuration phases. Due to the temporal separation between these phases, and potentially a different set of developers operating at each phase, it is not feasible to identify specialization opportunities all at once. Instead, with each successive phase of the development lifecycle, system properties start becoming invariant one by one. For example, the system composition of an end-to-end task chain may reflect the need to differentiate priorities among multiple information flows across the tasks. However, whether the requests within a flow are handled at a fixed priority at each task or whether the priorities are propagated end-to-end will be evident only after the developers configure the system. Thus, any specialization will have to wait until the configuration of the system is known. So there is a need to identify the context for performing specializations by inspecting the application composition, configuration and deployment models.

I.2.3 Lack of mechanisms for reusing specializations

Unlike the years of efforts in documenting good patterns of software design, there is a general lack of a knowledge base documenting reusable patterns for middleware specialization, which leads to reinventing specialization efforts in identifying what specializations are needed, and in realizing them. For example, if there is no approach to document how the specializations for a particular priority model are performed, then developers will be faced with similar challenges every time the same specialization is to be performed on a different DRE system. So what is required is a reusable, systematic and automated process for specializing middleware.

I.2.4 Lack of mechanisms for transparent provisioning of domain-specific semantics

Supporting domain-specific semantics for instance, *application-transparent* failover of a group of components is important to extend the benefits of separation of concerns provided by component-based middleware to dependable 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. Large-scale DRE systems require such flexibility because it simplifies planning for mode change involving graceful degradation in their QoS as opposed to an abrupt denial. 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.

I.2.5 Achieving Safe Adaptability To Runtime Failures While Maintaining The Hard Real-Time

While the middleware can be optimized at design time for the specific application requirements, it is still subject to unexpected events that can occur at runtime i.e., failures. With an increasing trend towards realizing larger system-of-systems that are composed predominantly from existing systems (*e.g.* ultra large-scale systems [53] or cyber-physical systems [138]), which we collectively term as distributed real-time and embedded (DRE) systems. The realization of DRE systems gives rise to interdependencies between individual subsystems. Moreover, a new range of faults arise in the context of DRE systems, which must be handled to maintain the mission-critical nature of the overall DRE system in the context of the induced interdependencies.

Unfortunately, the original design of these subsystems seldom enable their seamless integration within the larger DRE systems. For example, due to the critical nature of the

real-time tasks executing in these subsystems, the individual subsystems are often over-provisioned in terms of resource usage to deal with worst case fault scenarios. This over provisioning in the individual systems is detrimental to realizing reliable DRE systems because fault tolerance solutions for the DRE systems do not have the flexibility to add new resources or modify the real-time schedules of tasks in the individual subsystems. Re-designing and reimplementing the individual systems is not an option due to economic forces. Thus, designing fault-tolerance mechanisms for DRE systems must somehow utilize the available resources without compromising the real-time properties of the individual subsystems. Consequently, there is a need to identify opportunities for resource availabilities so that DRE system fault tolerance solutions can be implemented. The key insight we leverage for our work depends on the existence of a *significant slack* in the over-provisioned individual subsystems. The challenge lies in identifying this slack and making effective use of it, which is the subject of this research.

Our next question is identifying the right fault tolerance mechanisms for DRE systems. Software Health Management (SHM) [35, 118] is a promising approach to providing fault-tolerance in real-time systems because it not only provides for fault detection and recovery but also effective means for fault diagnostics and reasoning, which can help make effective and predictable fault mitigation and recovery decisions. However, this technique does not account for the constraints in the resources and applies only to latent errors in component functional implementations that are known a priori with predefined failover strategies. If SHM were to be used as is in DRE systems, it is likely to result in suboptimal runtime failure adaptations that do not utilize the resources in the most effective manner.

Adaptive Fault Tolerance (AFT) [8] has been known to improve the overall reliability and resource utilizations, however, for soft real-time applications through dynamic runtime failure adaptation techniques. Since they require additional resources to perform failure recovery, they can consume precious time from the real-time schedule of individual subsystems. Therefore, these failure adaptation mechanisms need to be provisioned in a safe

manner (*i.e.*, without compromising the real-time schedules of existing real-time tasks) while still accruing the benefits of adaptations and better resource utilizations. While both SHM and AFT are promising techniques, they have limitations for use in DRE systems.

I.3 Research Approach

To address the middleware specialization challenges identified in Section I.2 this dissertation describes (1) the contemporary research in specializing middleware in terms three dimensional taxonomy, (2) a feature-oriented approach to reasoning about application requirements and composition, deployment and QoS models to determine the middleware features that are required by the application components along with the specialization context that help drive the specializations to be performed on the underlying middleware in order to support their QoS demands, (3) the reverse-engineering, generative and AOP techniques that rely on source code inspection to prune the middleware features, specialize the middleware sources and augment domain-specificity within the middleware runtime entities, and (4) a proposed approach for specializing application server composition and deployment infrastructure. A brief summary of the different aspects of this dissertation is presented below.

1. **Taxonomy of Contemporary Middleware Specialization Techniques** creates a vocabulary to reason about middleware specialization techniques in terms of the development lifecycle, the paradigms and feature-oriented dimensions. Every paradigm used for specialization either prunes or augments features or both and is applicable across one or more of the lifecycle stages. This makes it easy to classify the specialization techniques and reason about their impact on the middleware. The taxonomy also aids the identification and development of a specialization lifecycle which is the described in detail in the subsequent chapters. Chapter II describes the specialization techniques and their taxonomy in detail.

2. **Feature-oriented Reasoning Techniques to Drive the Middleware Specializations** has been demonstrated using the decision tree based reasoning to determine the middleware features that are being used by the applications and model interpretation technique to automatically determine application invariants that provide the context to determine what specializations are applicable. Chapter IV describes the reasoning and deduction techniques for driving specializations in detail.

3. **Automated Realization of Middleware Specializations** enables automated identification of specialization points and the generation of specialization directives that enable transformation of the middleware sources. A build specialization technique is also described that helps automatically prune down the build configurations based on computation of independent closure sets of code artifacts dependencies. Chapter V describes this approach in detail.

4. **Safe Adaptation of Middleware** to enable predictable and efficient adaptation to various granularities of runtime failures while maintaining real-time constraints and improving existing resource utilizations. Existing SHM mechanisms have been enhanced with a flexible and configurable distributed resource monitoring framework and an intelligent failure adaptation algorithm that takes into account the failure types, granularity and failover replica placement. Chapter VI describes the framework architecture and adaptation algorithm and the experimental validations in detail.

I.4 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 the contemporary middleware specializations and their classification into a three-dimensional taxonomy. Chapter IV describes feature-oriented reasoning techniques for discovering opportunities to drive middleware specializations. Chapter V presents the automated and generative transformation approach and the corresponding algorithms for specializing middleware and its build system. Chapter VI presents the safe and predictable middleware adaptation approach and the corresponding frameworks and algorithm for adapting the middleware to runtime failures. Chapter VII presents the future work that discusses the side effects of specializing middleware in the context of application server composition and deployment, which are not adequately addressed by contemporary solutions. A research approach is proposed address the challenges.Finally, Chapter VIII presents the concluding remarks and a dissertation timeline.

CHAPTER II

TAXONOMY OF CONTEMPORARY MIDDLEWARE SPECIALIZATION TECHNIQUES

This chapter surveys the existing body of research in middleware specializations and categorizes it into three dimensions of lifecycle, paradigm and feature manipulation. Examples of each category are described and compared in detail. It organizes the research into a taxonomy representation and proposes a multi-stage lifecycle for specializing general-purpose middleware.

II.1 Middleware Background

II.1.1 Definition

Middleware is the connectivity software or communication infrastructure bus that encapsulates a set of services residing between the operating system layer and the user application layer. Middleware facilitates the communication and coordination of application components that are potentially coordination of application components that are potentially distributed across several networked hosts. Moreover, middleware provides application developers socket programming. In this manner, middleware can hide interprocess communication, mask the heterogeneity of the underlying systems (hardware devices, operating systems, and network protocols), and facilitate the use of multiple programming languages at the application level. Middleware can also be considered as a *"glue"* that enables integration of legacy the use of multiple programming languages at the application level. Middleware ISO OSI reference model [15]. We now discuss the role specialization can play in various application s, effectively implementing the session and presentation layers

II.1.2 Traditional Middleware Specialization

Traditional middleware can be classified based on the type of programming-language abstraction that it provides for interaction among distributed software components: procedural, object-oriented, transactional, or message-oriented. The corresponding primitive communication techniques are distributed remote procedure calls (RPC), remote object invocations, transactions, and message passing respectively.

- **RPC middleware** extends the procedure call in procedural programming languages to include remote procedure calls (RPC), where the body of the procedure resides on a remote host and can be called the same way as a local procedure. Birrell [14] implemented the first full-fledged version of RPC. Sun Microsystems adopted RPC as part of its open network computing. Later, Open Group developed a standard for RPC called distributed computing environment (DCE) [107]. Most Unix and Windows operating systems now support RPC facilities. RPC middleware can be specialized is to use the static and dynamic binding selection such that the right network transport is configured for the application's needs.

- **Object-oriented middleware** combines object-oriented programming paradigm and the RPC architecture. It provides the abstraction of a remote object, whose methods can be invoked as if the object were in the same address space as its client. Encapsulation, inheritance, and polymorphism are often supported by this type of middleware. *e.g.* CORBA [85], Java RMI [121], and DCOM [79]. One approach of specializing OO middleware can be to use frameworks and design patterns such as strategy, factory, adaptor, acceptor-connector, leader-follower to customize the middleware processing to the applications runtime requirements. Another approach could be to use compile-time specialization may include code optimization for specific architectures and hardware platforms. AOP can be useful for both these approaches.

- **Transactional middleware** supports distributed transactions among processes running on distributed hosts. Originally, this type of middleware was targeted at interconnecting heterogeneous database systems. The goals include providing data integrity, high-performance, and availability using the *two-phase commit protocol*. *e.g.* IBM CICS [50] and BEA Tuxedo [46]. Transactional middleware can be customized similar to OO middleware where specific binding mechanisms such as most resource efficient database drivers, query optimization on-the-fly.

- **Message-oriented middleware (MOM)** facilitates asynchronous message exchange between clients and servers using the message-queue programming abstraction [39], a generalization of the operating system mailbox. Messages do not block a client and are deposited into a queue with no specific receiver information. In addition, the message-queue abstraction decouples clients and servers, which enables interaction among otherwise incompatible systems. *e.g.* IBM MQSeries [52] and Sun Java Message Queue [120]. MOM middleware could be specialized to handling a particular routing substrate (using multicast group communication, broadcast), storage management technology or communication protocols (such as publish-subscribe, peer-to-peer, etc.).

The specialization techniques used to customize and optimize traditional middleware are disconnected and need to be integrated and enhanced with the new specialization technologies such as aspect-oriented programming, component-based design and model driven engineering.

II.2 Taxonomy of Middleware Specialization Techniques

Middleware can be categorized with respect to the type of specialization it provides. As suggested before Middleware research can be broadly classified along three dimensions of

application development: (1) feature-dependent, (2) paradigm-dependent, and (3)lifetime-dependent.

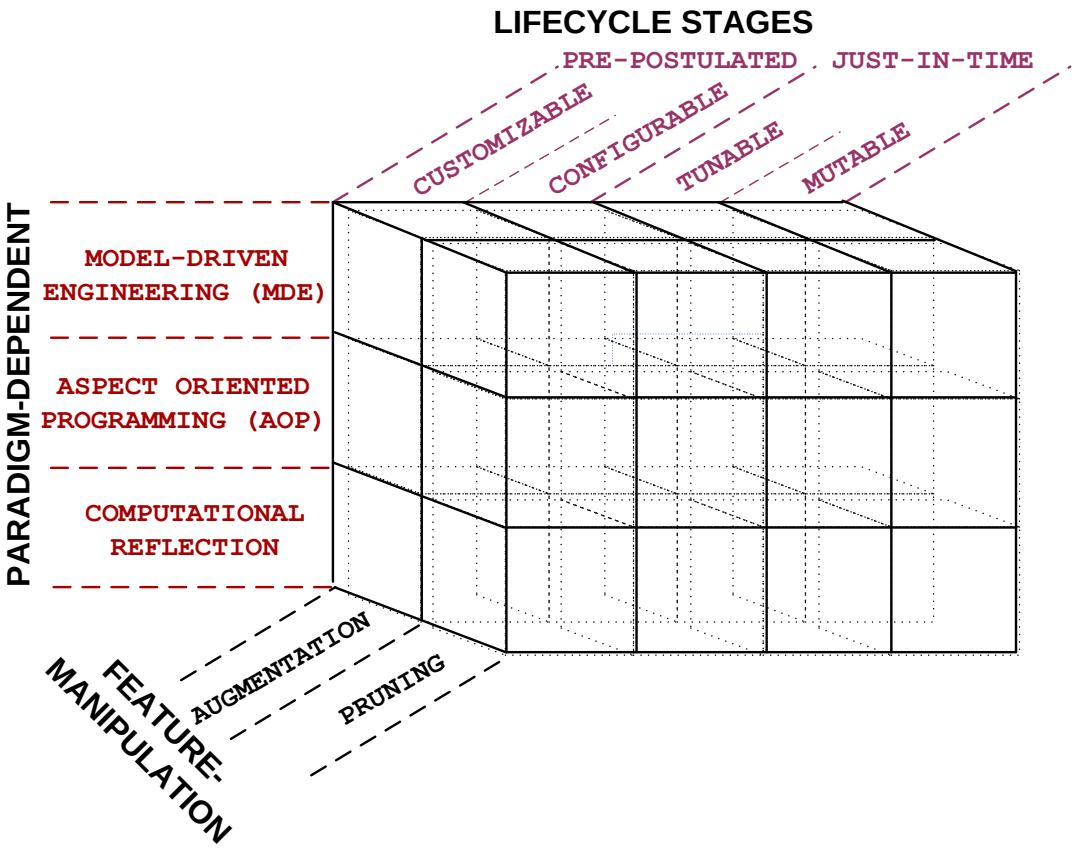


Figure 4: Three Dimensional Taxonomy of Middleware Specialization Research

II.2.1 Feature-Dependent Specialization

Feature-oriented programming (FOP) captures the variants of a base behavior though a layer of encapsulation of multiple abstractions and their respective increments that together pertain to the definition of a feature [78]. FOP decomposes complex software into features which are the main abstractions in design and implementation. They reflect user

requirements and incrementally refine one another. FOP is particularly useful in incremental software development and software product lines (SPLs).

The specialization of a middleware platform along the feature-dependent dimension consists of composing it according to the features/functionalities required by the hosted applications. This is a dynamic process that consists of augmenting/inserting new features as well as pruning/removing unnecessary features. We distinguish between feature pruning and feature augmentation specialization strategies as follows:

II.2.1.1 Feature Pruning

Feature pruning is a strategy applied to remove features of the middleware to customize it. In this case the original middleware provides a broad range of features but many are not needed for a given use case. These unwanted features are pruned from the original middleware. This approach is taken by FOCUS [68] where unnecessary features are automatically removed from general purpose middleware through techniques such as memoization to provide optimizations for DRE systems.

II.2.1.2 Feature Augmentation

Feature augmentation is a strategy applied when the specialization is grounded via the insertion of new features, either because the original middleware did not support it or the middleware is composed out of building blocks [3, 16, 128]. The latter variety of middleware platforms are designed to overcome the limitations of monolithic architectures. Their goal is to offer a small core and to use computational reflection to augment new functionalities.

In Section A.1, AOP can be used to compose middleware platforms where the middleware core contains only the basic functionalities [51, 142]. Other functionalities that implement specific requirements of the applications are incrementally augmented in the

middleware by the weaver process, when they are required and decrementally pruned when they are not required.

II.2.2 Lifetime-Dependent Specialization

One approach to classify specialization techniques is based on the time scale at which it is implemented: *pre-postulated* and *just-in-time* [141]. Figure 5 shows this dimension of our taxonomy. If middleware specialization is performed during the application compile or startup time, we designate it *pre-postulated/static specialization*. For example, Embedded-Java (`java.sun.com/products/em\discretionary{-}{}{}beddedjava`) minimizes the footprint of embedded applications during the application compile time. Similarly, if the middleware specialization is performed during the application run time, we designate it *just-in-time/dynamic specialization*. For example, MetaSockets [109] load adaptive specialization code during run time to adapt to wireless network loss rate changes. Notice that in Figure 5, dynamism increases from left to right.

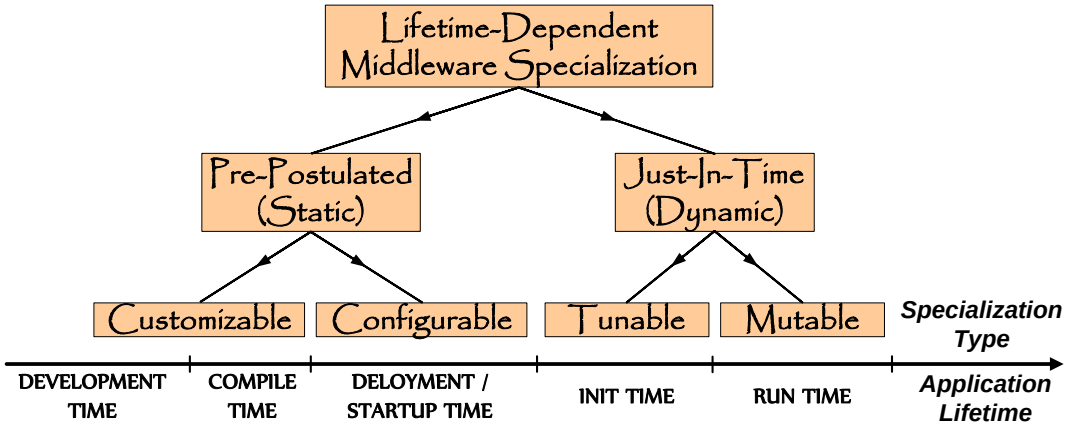


Figure 5: Lifetime-Dependent Middleware Specialization

II.2.2.1 Pre-postulated Specialization

Pre-postulated or Static specialization tailors the middleware before knowing its exact application use case. This process tries to identify the general requirements of possible future applications and defines the middleware configuration that will be used by the applications. It is further divided into customizable and configurable middleware.

- **Customizable specialization** enables adapting the middleware during the application compile/link-time so that a developer can generate specialized (adapted) versions of the application. Note that a customized version is generated in response to the functional and environmental changes realized after the application design-time. Examples of specialization mechanisms provided by customizable middleware are static weaving of aspects [63], compiler flags, and precompiler directives [66]. QuO [144] and EmbeddedJava are examples of customizable middleware.
- **Configurable specialization** enables adapting the middleware during the application startup time thereby enabling an administrator to configure the middleware in response to the functional and environmental changes realized after application compile time during its deployment or startup. Some examples of specialization mechanisms provided by configurable middleware include CORBA portable interceptors [84], optional command-line parameters, for example, to set socket buffer-size, and configuration files such as ORBacus configuration file (`www.orbacus.com`).

II.2.2.2 Just-in-time (JIT) Specialization

Just-in-time (JIT) or Dynamic specialization occurs at run time by identifying the requirements of the running application and customizing the middleware according to the application needs. It can be further classified into tunable and mutable middleware.

- **Tunable Specialization** enables adapting the middleware after the application startup time but before the application is actually being used. Doing so enables an administrator to fine-tune the application in response to the functional and environmental

changes that occur after the application is started. Examples of specialization mechanisms provided by tunable middleware are "two-step" specialization approaches (including static AOP during compile time and reflection during run time) employed by David et. al [27] and Yang et. al [139], the component configurator pattern [112] used in DynamicTAO [67], and the virtual component pattern [24] used in TAO and ZEN middleware.

- **Mutable Specialization** is the most powerful type of middleware specialization that enables adapting an application during run time. This specialization is also called Adaptive Specialization. Hence, the middleware can be dynamically specialized while it is being used. The main difference between tunable middleware and mutable middleware is that in the former, the middleware core remains intact during the tuning process whereas in the latter there is no concept of fixed middleware core. Therefore, mutable middleware are more likely to evolve to something completely different and unexpected. Examples of specialization techniques provided by mutable middleware are reflection [16], late composition of components [66], and dynamic weaving of aspects [139].

II.2.3 Paradigms-Dependent Specialization

Numerous advances in programming paradigms have also contributed to middleware specialization techniques. Although many important contributions have been made in this area, a review of the literature shows that four paradigms, in addition to object-oriented paradigm, play key roles in supporting middleware specialization: computational reflection [19], component-based design [124], aspect-oriented programming [63], and feature-oriented programming [105].

There are other approaches such as program slicing, partial evaluation, policies, automatic tuning of configuration parameters that enable customization of system software.

However these approaches are more fine-grained in the sense that they are used to manipulate, customize and verify the correctness of individual programs. However, each of these approaches can be utilized through the more coarser-grained approaches that are being considered in this paper.

II.2.3.1 Computational Reflection

Computational reflection [19] refers to the ability of a program to reason about, and possibly alter, its own behavior. Reflection enables a system to *open up* its implementation details for such analysis without compromising portability or revealing the unnecessary parts. In other words, reflection exposes a system implementation at a level of abstraction that hides unnecessary details, but still enables changes to the system behavior [72]. As depicted in Figure 6, a reflective system (represented as base-level objects) has a self representation (represented as meta-level objects) that is causally connected to the system meaning that any modifications either to the system or to its representation are reflected in the other.

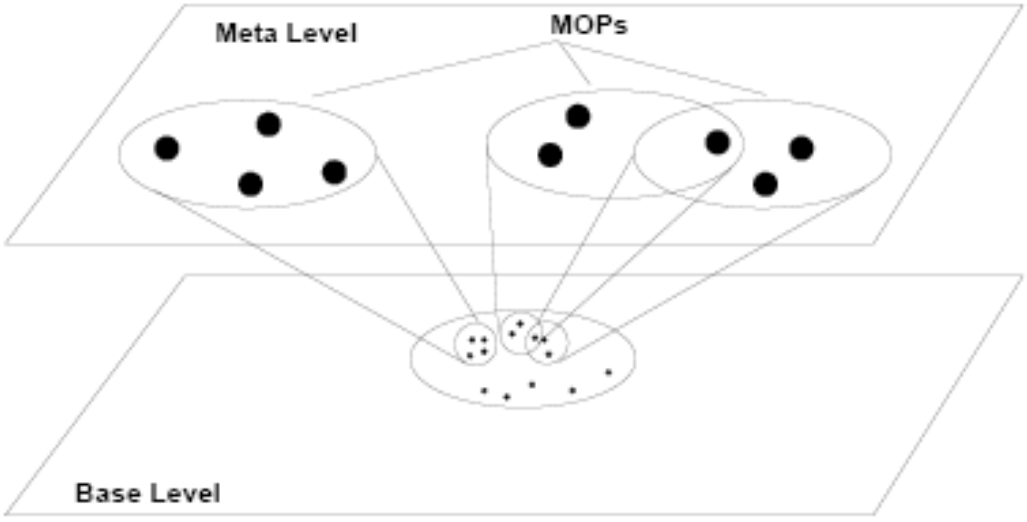


Figure 6: A Reflective System with Causally Connected Meta-level

The base-level part of a system deals with the *normal* (functional) aspects of the system whereas the meta-level part deals with the computation (implementation) aspects of the system. The meta-level contains the building blocks responsible for supporting reflection. The elements of the base-level and that of the meta-level are, respectively, represented by base-level objects and meta-level objects. A meta-object protocol (MOP) [62] is a meta-level interface that enables *systematic* (as opposed to ad hoc) inspection and modification of the base-level objects and abstraction of the implementation details.

A causal connection/interface associates the base-objects with the meta-objects to support the base/meta objects communication and guarantees that modifications to the meta-level are reflected into corresponding modifications to the base-level and vice-versa. Recently, reflection has also been studied in middleware, where it enables adapting the behavior of a distributed application by modifying the middleware implementation. Reflective middleware is often concerned with adapting non-functional aspects of distributed applications including QoS, performance, security, fault tolerance, and energy management.

Computational reflection is an efficient and simple way of inserting new functionalities in a reflective middleware. Thus, it is necessary only to know components and interfaces. The next generation middleware [16, 40] exploits computational reflection to customize the middleware architecture. Reflection can be used to monitor the middleware internal (re)configuration [106]. The middleware is divided in two levels: base-level and meta-level. According to Figure 6, the middleware core is also represented by base-objects and new functionality is inserted by meta-objects. Figure 7 shows that the meta-level is orthogonal to the middleware and to the application. This separation allows the specialization of the middleware via extension of the meta-level.

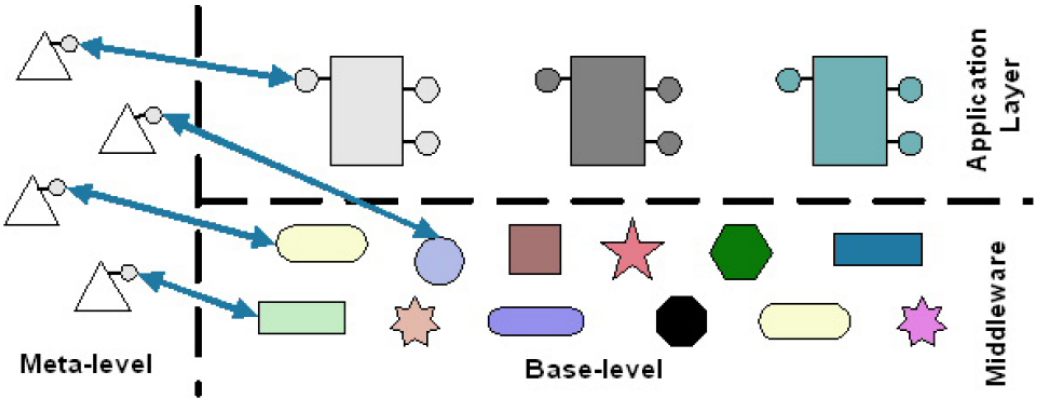


Figure 7: Reflective Middleware

II.2.3.2 Aspect Oriented Programming (AOP) Techniques

Kiczales et al. [63] realized that complex programs are composed of different intervened crosscutting concerns (properties or areas of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are still scattered among different classes thereby complicating the development and maintenance of applications.

AOP [63] applies the principle of "separation of concerns" (SoC) [99] during development time in order to simplify the complexity of large systems. Later, during compile or run time, an aspect weaver can be used to weave different aspects of the program together to form a program with new behavior. AOP proponents argue that disentangling the crosscutting concerns leads to simpler development, maintenance, and evolution of software. Naturally, these benefits are important to middleware specialization. Moreover, AOP enables factorization and separation of crosscutting concerns from the middleware core [119], which promotes reuse of crosscutting code and facilitates specialization.

In the context of middleware, we refer to AOP approaches as existing software platforms that expose hooks for applications using these platforms, to adapt, alter, modify, or

extend the normal execution flow of a service requested. Non-functional features (monitoring code, logging, security checks, etc.) can be transparently woven into the middleware code paths or unnecessary features can be pruned through bypassing code paths or middleware layers. In that sense, the CORBA portable interceptor (PI) mechanisms, although not explicitly positioned as an aspect-oriented approach, belong to this category. Using AOP, customized versions of middleware can be generated for application-specific domains. Yang et al. [139] and David et al. [27] both provide a two-step approach to dynamic weaving of aspects in the context of middleware specialization using a static AOP weaver during compile time and reflection during run time. Other recent examples explicitly positioning themselves as aspect-oriented approaches are the JBoss AOP approach (`www.jboss.org`) and the Spring AOP approach (`www.springframework.org`).

II.2.3.3 Component-Based Design (CBD) Techniques

Software components are software units that can be independently produced, deployed, and composed by third parties [124]. Components are self-contained: components clearly specify what they require and what they provide. CBD supports the large scale reuse of software by enabling assembly of *commodity-off-the-shelf* (COTS) components from a variety of vendors. The independent deployment of components enables late composition (also referred to as late binding), which is essential for adaptive systems. Late composition provides coupling of two compatible components at run time through a well-defined interface. A system developed using CBD is an amalgam of components that can be reorganized easily. When applied to middleware, CBD provides a flexible and extensible system. Specially, a middleware can be customized to specific application domains, through the integration of domain-specific components, and can evolve using third-party components. Moreover, component-based middleware can be dynamically adapted to its environment using late composition. Examples of major component-based middleware solutions are DCOM [79] (discussed earlier), EJB [28], and CCM [12].

Enterprise Java Beans (EJB) is a middleware component model for Java proposed by Sun Microsystems that enables Java developers to use off-the-shelf Java components, or beans. Since EJB is built on top of Java technology, EJB components can only be implemented using the Java language, however. The EJB component model supports specialization by automatically supporting services such as transactions and security for distributed applications. The CORBA Component Model (CCM) is a distributed component model proposed by OMG that can be considered as a cross-platform, cross-language superset of EJB. CCM supports specialization by enabling injection of adaptive code into component containers (i.e., the component themselves remain intact).

II.2.3.4 Model-Driven Engineering (MDE) Techniques

MDE is an emerging paradigm that integrates model-based software development techniques (including Model-Driven Development [111] and the OMG's Model Driven Architecture) with QoS-enabled component middleware to help resolve key software development and validation challenges encountered by developers of large-scale distributed, real-time and embedded (DRE) middleware and applications. In particular, MDE tools can be used to specify requirements, compose DRE applications and their supporting infrastructure from the appropriate set of middleware components, synthesize the metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. These activities can be performed in a cyclic fashion until the QoS constraints are satisfied end-to-end.

Conventional middleware architectures suffer from insufficient module-level reusability and the ability to adapt in the face of functionality evolution and diversification. As reported in [142], *"intrinsic"* and *"extrinsic"* properties interact non-modularly in conventional middleware architectures. Consequently, middleware architects are faced with immense architectural complexities because the concern density per module is high. The code-level reusability of the *"common abstractions"* is also drastically reduced because

the generality of intrinsic components is restricted by the *"extrinsic"* properties in the face of domain variations. A contributing factor to this complexity, is that the code-level design reusability in conventional middleware architectures is incapable of adequately dealing with *"change"* in two dimensions: time (functional evolution) and space (functional diversification).

The reusability in conventionally developed software components is insufficient due to the lack of explicit means to effectively distinguish intrinsic and extrinsic architectural elements. Conventional middleware architectures also lack effective means to reuse *"extrinsic"* properties, especially ones that are crosscutting [63] in nature, *i.e.*, not localized within modular boundaries. Conventional architectures have fallen short of doing so because they are incapable of componentizing and reusing crosscutting concerns as analyzed in [140]. Being able to componentize and to reuse these functionalities tremendously facilitates the construction of middleware systems. To tackle the aforementioned problems, Zhang et. al. [142] propose a new architectural paradigm called Modelware which embodies the *"multi-viewpoints"* [83] approach.

II.3 Assessment of Modularization Techniques for Middleware Specialization

In this section we use our taxonomy to assess the strengths and weaknesses of various modularization approaches used for specializing middleware. We then develop a framework for systematic and automated middleware specialization that provides guidelines for middleware application developers to reason about, optimize, customize and tune the middleware according to their domain-specific requirements.

II.3.1 Qualitative Evaluation of the Middleware Specialization Taxonomy

In the following we use a combination of artifacts of individual dimensions of our taxonomy to assess the pros and cons of various modularization techniques when applied to middleware specialization.

Table 1 summarizes our assessment of different modularization techniques. We briefly discuss below each paradigm with respect to the lifetime dimension of the taxonomy

a. Pre-postulated Specializations: FOP, AOP and MDE are widely used at design-time and compile-time respectively to perform feature augmentation and pruning. Although feature modules – the main abstraction mechanisms of FOP – perform well in implementing large-scale software building blocks, they are incapable of modularizing certain kinds of crosscutting concerns [5]. This weakness is the strength of aspects. Caesar [77], AFMs [5] combine FOP with AOP to overcome the shortcomings of "purely hierarchial" feature specifications in FOP. However, reflection has limited application during the pre-postulated phases except during deployment it could be used to inspect the target platform features before the application is deployed.

b. Just-in-time Specializations: AOP has few use cases at just-in-time where dynamic weaving of feature aspects could be set up with the help of native compile-time platform support, such as Java Virtual Machine (JVM) [103]. JAsCo [130] is an adaptive AOP language used to specialize Web Services implementations [132] whereas PROSE [82] and Abacus [141] are just-in-time aspect-based middleware. Beyond design-time, MDE cannot be applied since it relies mainly on predetermined system feature requirements. However, it can configure dynamic augmentation or pruning of features at run-time. Recently models at run-time has been used for self-healing systems. The principles from those domains need to be applied for specializing middleware dynamically based on models. Computational reflection can be used to support the runtime introspection of the application and perform dynamic augmentation and pruning of features to adapt its internal implementation and reconfigure itself depending upon the dynamic conditions prevalent at run-time. However, This enables support for more powerful dynamic specializations which are useful for power and resource management, and dynamic adaptation as in wireless sensor networks, embedded systems, etc.

II.3.2 Guidelines for Middleware Specialization

We now provide guidelines for middleware specialization using our taxonomy. We use the lifecycle dimension as the dominant dimension since it imparts a systematic ordering to the process of performing middleware specialization. We believe the guidelines can apply to any systems software, such as an operating system, web server or a database management system.

a. **Development-time specializations:** During development-time the middleware developer can program the application code with features that need to be loaded at initialization-time and features that can be swapped in/out at run-time through strategies. MDE and AOP based techniques are more effective to program development-time specializations. In this phase, feature-augmentation should be the goal.

b. **Compile-time specializations:** Compile-time specializations can be used to transparently weave-in (augment) or weave-out (prune) features code. AOP is the key enabler for performing compile-time specializations.

c. **Deployment-time specializations:** Deployment-time specializations mainly address target platform-specific concerns such as type of data transport, database drivers, etc. The middleware features are matched to make optimal use of the underlying platform feature constraints. Special tools which perform the task of setting up the deployment can use reflection to query the platform features and use AOP to transparently change the underlying bindings or supply the required configuration parameters when launching applications.

d. **Initialization-time specializations:** Feature configuration is performed at initialization-time using the configuration parameters that are pre-programmed either at development-time and/or compile-time or supplied during the application startup-time.

e. **Run-time specializations:** At run-time, features can be swapped in or out using either reflection or dynamic aspect weaving depending upon the conditions prevalent after the

application is executing. However, too much dynamism can lead to unpredictable application behavior leading to unstable specializations that are difficult to verify for safety criticality and correctness. To benefit from mutable middleware, we should harness its power using techniques such as safe specialization. So most of the dynamic feature swapping needs to be statically programmed before hand.

f. Integrated specializations: Since no single modularization technique can specialize middleware over all phases of the application lifetime, multiple techniques need to be applied and validated in unison starting with MDE and AOP at pre-postulated time whereas computational reflection at just-in-time. It is important to restrict feature changes at runtime that conflict with the design-time feature configurations. Applying overlapping specializations may cause inconsistencies in the applications. This is the same problem as the feature interaction problem in pattern recognition that needs to be addressed in middleware specialization also. Inconsistency can be caused when FOP, AOP or MDE augments a dependent feature set during pre-postulated phases but reflection prunes one of the features from the set during just-in-time phases which may lead to unpredictable runtime behavior and failures. Inconsistencies can also occur within the same life-time phase. Hence, tools and techniques are needed to validate specializations when multiple customization techniques are applied in tandem not only within a phase but across entire application lifetime.

g. Optimal specializations: Finally specialization tools should not only validate but also optimize various feature changes so that they are not only consistent but satisfy the quality of service (QoS) requirements of the applications.

II.4 Discussion

Adaptive middleware specialization is still an ongoing research that requires more work in the following areas. First, domain-specific middleware services requires serious attention as specialization approaches tend to be addressing domain problems. Several projects

have successfully provided common-services in middleware. To enable reuse and separation of concern in each specific application-domain, however, domain-specific middleware services should also be widely available. Second, mutable middleware specialization provides a powerful and at the same time dangerous dynamic specializations that are more likely than other types of middleware specializations to turn an application into something totally different and unexpected. This can be confirmed from the Table 1 that a very few approaches employ mutable specialization. To benefit from mutable middleware, we should harness its power by techniques such as safe specialization. Third, applying overlapping specializations to a distributed application may cause inconsistency in the application. This is the same problem as feature interaction problem in pattern recognition that needs to be addressed in middleware specialization also. Finally, we realized that there is no one middleware specialization solution that can suit all distributed applications. There are a few new areas such as context-aware middleware and publish-subscribe (pub-sub) middleware that could benefit a lot from the various specialization techniques. While there is ongoing research, there is still substantial amount of work to be done in order to achieve the benefits of specialization.

Finding an optimized and adaptive middleware specialization solution using current state-of-the-practice middleware specialization approaches is not an easy task. A developer needs to know all available middleware approaches and should spend a lot of time and money to find the optimized solution. Developing tools, techniques and high-level paradigms that assist a developer in this tedious process is a useful research area that promotes development of adaptive software. Inventing domain-specific specialization pattern languages can serve as guidelines for the synthesis of such tools.

Based on the insights gained from the taxonomy of middleware specializations, we have come up with an automated middleware specializations process in the next chapter.

Table 1: Evaluation of the Combinations of Dimensions

COMBINATIONS	USE CASES	STRENGTHS	WEAKNESSES	RELATED WORK
Pre-postulated + AOP	Weave/Prune at compile-time	Transparency without affecting core	Code Bloating	FACET, CLA, FOCUS, Bypassing Layers, AspectOpenORB
Pre-postulated + MDE	Weave/Prune only known features	Elegant design	Runtime specializations not possible	DTO, CLA, Modelware
Pre-postulated + Reflection	Inspect target platform features	Useful during deployment	Difficult to predict runtime conditions	AspectOpenORB, DTO
Just-in-time + AOP	Dynamic weaving of features	Dynamic Adaptation	Requires native platform support	JAsCo, PROSE, Abacus
Just-in-time + MDE	Self-healing systems	Validation of Specializations	Incur runtime overhead	Models@Runtime
Just-in-time + Reflection	Introspect runtime application features	Dynamic Adaptation reconfiguration	Can cause unpredictable behavior	AspectOpenORB
AOP + FOP	ISD and SPLs	Better modularization of crosscutting features	Runtime specializations not possible, cause conflicts	AFMs, Caesar
FOP + MDE	SPLs	Better composition of features	Runtime specializations not possible, cause conflicts	FOMDD [129]
AOP + Reflection	Composition based on application requirements	On-demand feature weaving	May cause conflicts	AspectOpenORB
AOP + MDE + FOP + Reflection	De-sign/Weave/Prune valid features combinations	Systematic, correct specialization process	Safe specializations is challenging	*Research Needed*

CHAPTER III

THE AUTOMATED MIDDLEWARE SPECIALIZATION PROCESS

This section presents the automated and generative middleware specialization lifecycle and the resulting framework for middleware specialization using generative, reverse engineering and AOP techniques. We assume that middleware developers develop module code bottom-up based on a design template and subsequently create the corresponding build configurations for their modules through mechanisms, such as Makefiles or Visual Studio Project files. We identify the requirements for an automated solution based on the taxonomy we developed in the previous chapter.

III.1 Unresolved Challenges

Since the requirements desired by the application are bound to change over the application lifecycle, the need for an extensible and portable automated specialization approach becomes even more apparent. Current specialization techniques do not provide a automated, generic, reusable, extensible and systematic mechanism for refining existing, and accommodating new specializations, as well as accounting for different middleware platforms. This dissertation research has identified a middleware specialization lifecycle (as show in figure 8) in the form of six key steps involved in providing an automated middleware specialization solution - 1) Specification of specialization concerns, 2) Deduction of specialization context, 3) Mapping of concerns to code artifacts, 4) Generation of specialization transformations 5) Transformation of the code artifacts and, 6) Composition and Configuration of specialized middleware forms. Some of the key research issues encompassing these specialization lifecycle steps include:

However, automating middleware specializations for DRE applications with stringent

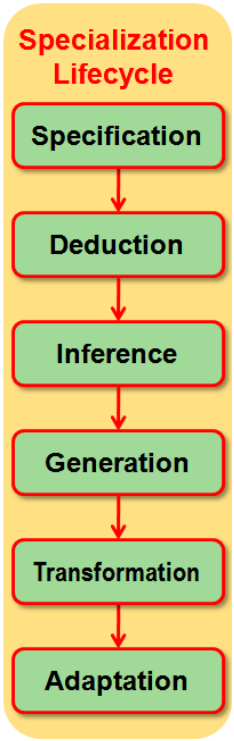


Figure 8: The Middleware Specialization Lifecycle

QoS requirements is a hard problem, which requires resolution of several research challenges described next. The research challenges in specializing middleware for DRE systems are encountered in all the stages of development lifecycle.

Detecting the system invariants manually on a case-by-case basis is infeasible, not to mention the subsequent manual efforts at specializing the middleware for each of the system under consideration. Many questions arise if automation is desired: How are the systems invariants to be identified automatically? Once these invariants are identified, how are they mapped to the underlying middleware-specific features that will indicate what parts of the middleware must be pruned and how the rest of the middleware be optimized? This problem is hard given that domain concerns crosscut class hierarchies of middleware design, and because system properties become invariant in different phases of the system lifecycle. For example, structural composition is often invariant after the design phase and remains so over the remainder of the application lifecycle. Similarly, the configuration and

deployment properties are invariant after the deployment and configuration decisions are made, however, they may vary on a per-deployment basis. We present the key requirements of an automated solution to middleware specialization.

III.1.1 Challenge 1: Inference of the Middleware Features

There is a general lack of reasoning methodologies that help inferring the middleware features directly from the requirements specifications by the application developers. There are techniques that help infer application features but they are not systematic and are completely manual. What is required is a reasoning methodology that is semi-automated and requires only a few higher-level features to automatically infer the lower-level features. Moreover, there is a need for a systematic and automated process that not only gives a standard way of requirements and feature reasoning but also scopes down the space of requirements to that only provided by the middleware. Such a reasoning process should help reason the application requirements in terms of middleware features which will further enable simplifying the specialization process.

III.1.2 Challenge 2: Determination of the Specialization Context

We define *specialization context* as the intent that drives the specialization process. Deriving the specialization context relies on detecting the system invariants [75], which become known over the application lifecycle stages. Examples of system invariants include periodic invocations such as timeouts that provide status updates in publish-subscribe communication paradigms, readonly operations, single interface operations that always get dispatched to the same server-side handlers, state synchronization tasks in stateful group failover [126]. Thus, in order to discover the specialization context, it is important to identify the *invariant system properties* from these high-level system models. However, the current state of art still relies on manual identification of the specialization context from the application composition, configuration and deployment models [68].

III.1.3 Challenge 3: Inferring the Specializations from the Specialization Context

DRE system developers must be able to map the specialization context to one or more known patterns of specialization. To eliminate the existing manual and non-scientific approaches, Inferring the set of specializations will require a repository of specialization patterns that can be queried using the context, which then returns a set of specializations applicable in that context. Such a repository must be extensible to include new patterns as they are discovered.

III.1.4 Challenge 4: Identifying the Specialization Points within the Middleware

The inferred patterns of specialization manifest at a higher level of abstraction than the level of middleware source code that actually must be transformed. Thus, there is a need to identify the collection of *Specialization Points,* which are regions of code within the middleware where specialization patterns will apply [63]. Although it is important to rely on patterns of specializations, such patterns are described at a higher level of abstraction than the level of middleware source code that actually gets transformed as an outcome of specialization. Thus, there is a need to identify the collection of *specialization points* within the middleware where specialization patterns can apply. The notion of a specialization Point is akin to that defined by AOP [63].

III.1.5 Challenge 5: Generating the Specialization Transformations

Although the specialization points are determined, the exact nature of the transformation to be carried out at those points corresponding to the specialization patterns must be specified as a set of transformations, which we call *Specialization Advice*. Currently, these transformation rules are manually developed [68] which is a tedious task that requires detailed knowledge of the middleware implementation architecture and can cause undesirable side effects within the middleware if developed incorrectly. Moreover, the maintenance of

these rules as the middleware frameworks and their respective sources evolve with changing application requirements.

III.1.6 Challenge 6: Executing the Specialization Transformations on Middleware Source

Once the specialization points are determined, the final step is applying the set of specialization techniques, which essentially are *specialization advice*, on the middleware source code so that the code paths are transformed and the code is optimized to reflect the intended specializations. Applying the advice requires a staging of backend tools, such as AspectJ and AspectC++, specific to the programming language in which the middleware is developed, or language-agnostic tools, such as Perl. Naturally, the manifestation of a specialization advice is specific to the programming language in which the middleware is developed. Examples include AspectJ or AspectC++ advice, or Perl expressions.

III.2 Process Overview

Figure 9 illustrates the automated middleware specialization process and we briefly describe the steps of the process below:

1. **Feature Mapping Wizard -** The application developer starts the middleware specialization wizard and begins describing the characteristics of the application to be developed specifying the PIM application, domain-level concerns needed for the application variant. The Feature Mapping wizard maps the PIM application domain-level concerns to PIM middleware features. The wizard asks questions about the configuration requirements and options of the application for which middleware is to be developed. The selected features are also configured along the way as they are selected for composition. The PLE developer response determines the next question that will be asked.
2. **Deduce Invariants -** application invariants provide the specialization context to drive the specializations. The invariants are detected by parsing the system models through model

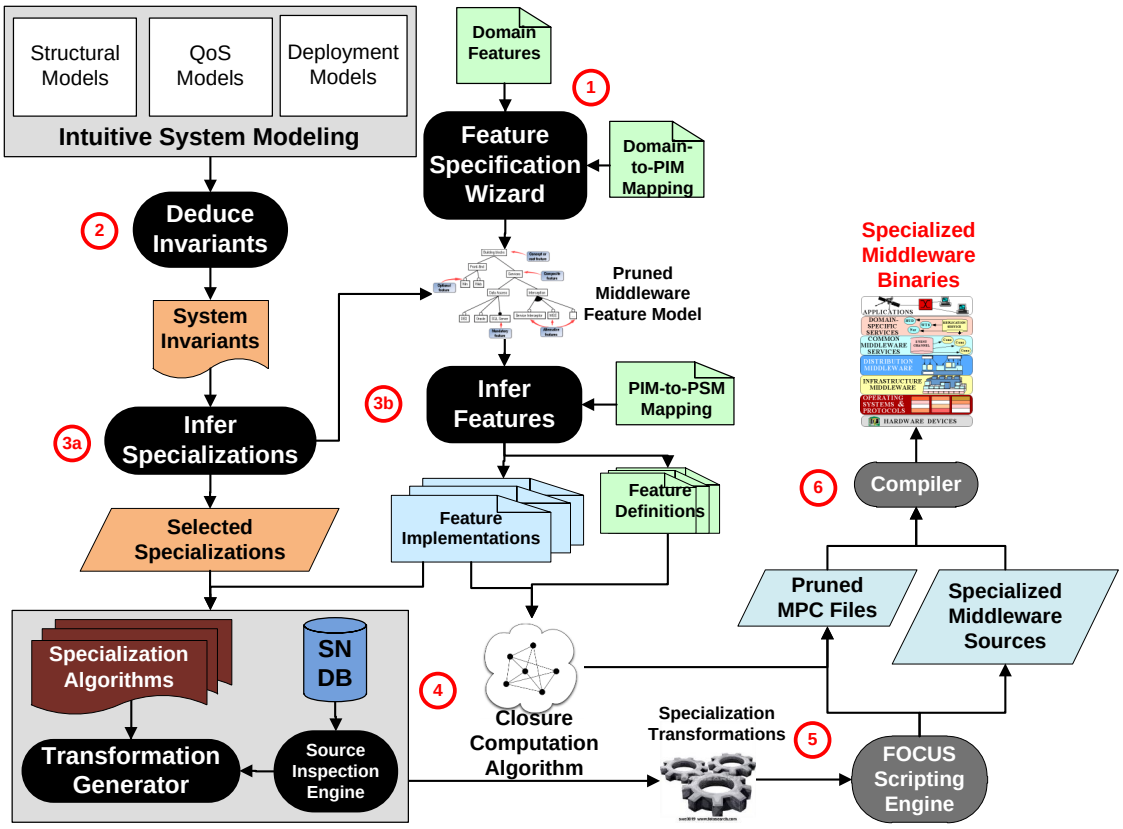


Figure 9: The Automated Middleware Specialization Process

interpreters. Invariants are application properties which maybe structural or configuration or deployment that don't change over the application use case. The fact that these invariants don't change leads us to believe that the middleware control path used by their implementations gets executed repeatedly.

3. **Infer Features -** Once the pruned PIM middleware feature set is obtained from the wizard, it is then mapped to the actual PSM middleware features that implement the individual PIM features using the PIM-PSM mappings that are provided by the middleware developer.

```
Extracted Features \bullet PIM-PSM Mapping = Source Feature Definitio
```

4. **Infer Specializations -** Once the specialization invariants are determined, they are looked up in the specialization knowledge base - `SP-KBASE` to determine the specializations

that are applicable. The specializations are then ordered according to the dependencies specified in the `SP-KBASE`.

5. **Transformation Generator -** The transformation generator includes a source inspection engine that parses the middleware source code and modularizes it into code blocks which indirectly help identify the specialization points. The generator specializes the middleware frameworks based on the inferred invariant features by removing all the framework indirections and hardcoding the use of that feature directly.
6. **Closure Computation:** Once the hints are obtained, they are used to create closure sets using an algorithm that systematically composes the source code and files that are associated with each feature into a feature module (FM). The closure sets are essentially all the dependencies that are gathered by the tool.
7. **Specialized Middleware Synthesis:** The build configuration is specialized by adding source files from individual closure sets of feature modules to the build descriptor thereby generating the build configuration file, such as a Makefile. For our evaluations, the process generates the Make Project Creator (MPC) [38] build configuration file. This MPC file represents the part of the specialized middleware that is to be built for the application variant. The generated MPC file is then used to create PSM Makefiles by running the MPC-supplied perl-based scripts. The platform-specific Makefiles are then used to for compilation of the specialized middleware for the application component or entire application variant. Thus multiple middleware *forms* may be synthesized depending upon the whether they were compiled for for the entire application or individual components.

Notice that this process is entirely repeatable and reusable. A repository of requirements for application variants can be maintained. There is no need to maintain the customized versions of the middleware since it can be synthesized through this process on demand. In the next two chapters we focus on some of the important building blocks of specialization process.

CHAPTER IV

FEATURE-ORIENTED REASONING TECHNIQUES TO DRIVE THE MIDDLEWARE SPECIALIZATIONS

The previous chapter realized a taxonomy for categorizing contemporary middleware specialization research. The taxonomy lead insights in developing a multi-stage middleware specialization lifecycle. However, realizing each of the stages of the specializing lifecycle is a tedious process which requires reasoning about the applications to discover opportunities for specializing middleware.

This chapter addresses the first challenge outlined in Section I.2 Ű feature-oriented reasoning to drive middleware specializations. First, an overview of the existing research in the field of specification and reasoning techniques is presented. Second, a list of challenges that are still unresolved is presented. Finally, a solution approach is presented that provides a feature oriented reasoning technique for determining the middleware feature requirements and an automated deduction technique for identifying the application invariants that provide the specialization *context*.

IV.1 Related Research

We survey and organize related work along two different dimensions: forward engineering and reverse engineering, and the techniques they use to realize these processes.

IV.1.1 Forward Engineering Approaches

IV.1.1.1 Feature-oriented programming (FOP) for feature module construction

Current PLE research is supported primarily through feature-oriented programming (FOP) techniques as advocated by AHEAD [11], CIDE [60], and FOMDD [129]. These

approaches are based on processes that annotate features in source code and compose feature modules that are essentially fragments of classes and their collaborations that belong to a feature. Being forward engineering techniques that they rely on clear identification of features, their dependencies and their interactions right from the requirements gathering stage of the PLE software lifecycle. Some efforts in this direction stem from the identification of feature interactions, their dependencies, granularity and their scope [5].

The reasoning process presented in this dissertation research encompasses the AHEAD and CIDE FOP methodologies by leveraging reverse engineering to enable automatic identification of features and their dependencies and composing only the features that directly serve the domain concerns of the application variant application. However both approaches rely on manual identification of features in legacy source code and manual definition of composition rules. Closure computation can be potentially extended by integrating both AHEAD and CIDE based FOP approaches to support fine-grained composition of feature modules.

IV.1.1.2 Addressing the spatial disparity between OO design and domain concerns

Both aspect-oriented programming (AOP) and feature-oriented programming (FOP) have been used extensively for specializing systems by addressing the disconnect between the vertical decomposition of OO design and horizontal decomposition of domain concerns. For example, Lohmann et. al. [70] argue that the development of fine-grained and resource-efficient system software product lines requires a means for separation of concerns [127] that does not lead to extra overhead in terms of memory and performance which they show using AspectC++.

The *FACET* [51] project identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects that represent domain concerns, which then can be woven into the core middleware. Our work used FOP-based reverse engineering in a tool called FORMS [26] that prunes unnecessary features from the

middleware by deducing the necessary middleware features from high-level application requirements (*i.e.*, domain concerns).

IV.1.1.3 Specialization in temporally distinct phases of application lifecycle

The *Modelware* [142] methodology adopts both the model-driven engineering (MDE) [111] and AOP. The authors use the modeling views – *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains, and *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change. Edicts [22] is an approach that shows how optimizations are also feasible at other application lifecycle stages, such as deployment- and run-time. Just-in-time middleware customization [141] shows how middleware can be customized after application characteristics are known.

Polze *et. al.,* [102] 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.

IV.1.1.4 Combining modeling and aspects for refinement

The *Modelware* [142] methodology adopts both the model-driven architecture (MDA) [87] and AOP. Borrowing terms from subject-oriented programming [47], the authors use the term *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains. They use the term *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change.

Modelware advocates the use of models and views to separate intrinsic functionalities

of middleware from extrinsic ones. Modelware considerably reduces coding efforts in supporting the functional evolution of middleware along different application domains. These are mainly forward engineering approaches that are dependent upon an efficient design process. However, most of the existing general purpose middleware has already been developed and there is a need to facilitate its specialization for domain-specific use through top-down reverse engineering approaches like FORMS.

Moreover, both FACET and Modelware being forward engineering approaches there is no automatic solution to manually annotating features and identification of cross-cutting concerns and modularizing them.

IV.1.2 Reverse Engineering Approaches

IV.1.2.1 Design Pattern Mining from source

Substantial research has been conducted on discovering design and architectural patterns from source code [32]. However, most such techniques are informal and therefore lead to ambiguity, imprecision and misunderstanding, and can yield substandard results due to the variations in pattern implementations. In order to specialize middleware such design pattern mining techniques need to be well supported by round-tripping techniques provided by our methodology that will enable any specializations at design level to reflect back into the source code. We are investigating the application of such techniques to automate feature annotation in source code.

Since forward engineering techniques focus on feature identification, static, and dynamic composition, they rely on strong modular boundaries. However, reverse engineering approaches like source code analysis which is the base of FORMS can prove to be beneficial to identifying features that span module boundaries and identifying discrepancies in the intended logical design of the middleware and their physical implementations.

IV.2 Unresolved Challenges

IV.2.1 Challenge 1: Identifying Opportunities to Drive Middleware Specializations

The higher-level application composition, QoS configuration and deployment models provide opportunities for detection of the specialization context which is used to determine and drive the specializations and optimizations that can be performed within the middleware. The application models of composition, QoS configurations and deployment specify the performance constraints such as response-time, throughput, timeliness and reliability that are placed on individual application components, their connections as well as the end-to-end workflows of components (known as component assemblies). Similarly by interpreting the QoS configurations it is possible to determine in advance what features from the underlying middleware will be utilized by the component applications. Additionally, the deployment of the application assembly can also provide useful hints for optimization from how individual components are mapped to machines, whether they are collocated, what kind of platform bindings, protocols, endianness they use, etc. Thus in order to discover the specialization context, it is important to identify the *invariant system properties* [75] from these high-level system models.

However, existing specialization techniques don't examine the application composition, configurations and interactions to deduce the repetitive and redundant tasks performed by the application. The application context that represents these repetitive tasks manifests itself in terms of periodic invocations such as timeouts that provide status updates in publish-subscribe communication paradigms, readonly operations, single interface operations that always get dispatched to the same server-side handlers, state synchronization tasks in stateful group failover [126]. Such repetitive that can be potentially sped up by optimizing the underlying middleware through caching [68], bypassing middleware layers [29], or fusion of layers [68], etc. to eliminate redundant processing at each middleware layer.

Moreover, the automatic detection of the specialization context also reduces the need

for a dedicated modeling annotation language to identify the context within the application models. Most coarse grained contexts can be detected automatically by examining the modeling structure and attributes but finer-grained contexts may need explicit identification. After automating the tedious task of identification of the middleware specialization context, the DRE system developer will still need to determine what specializations are applicable for a particular context. Current techniques of determining this mapping are still manual [68].

IV.3 Feature Oriented Reasoning

IV.3.1 Feature Mapping Wizard

In the development process, the automated specialization process's role is applicable in the packaging and assembly phases where the PLE application variants along with the hosting middleware is configured and packaged. The requirements reasoning wizard performs the difficult job of mapping the PIM application domain concerns to PIM middleware features. Domain concerns describe the characteristics of the application being developed. These characteristics may include functional concerns as well as non-functional (QoS) concerns. Functional concerns describe the way a particular application/application behaves, and its configuration. Non-functional concerns usually describe the way a application is supposed to perform which includes dimensions of concurrency, usually describe the way a application is supposed to perform which includes

Normally, domain concerns and middleware features manifest themselves into separate hierarchial representations. Therefore, a mapping is required to transform domain concern hierarchies to middleware feature hierarchial models. In order to create a systematic mapping, this wizard makes use of model transformations to navigate through the concern and feature hierarchies. Interestingly, both the functional and non-functional concerns can map within the same middleware feature model. The higher-level features in the decision tree

represent the functional concerns and since the lower-level features configure the higher-level features, they represent the non-functional concerns.

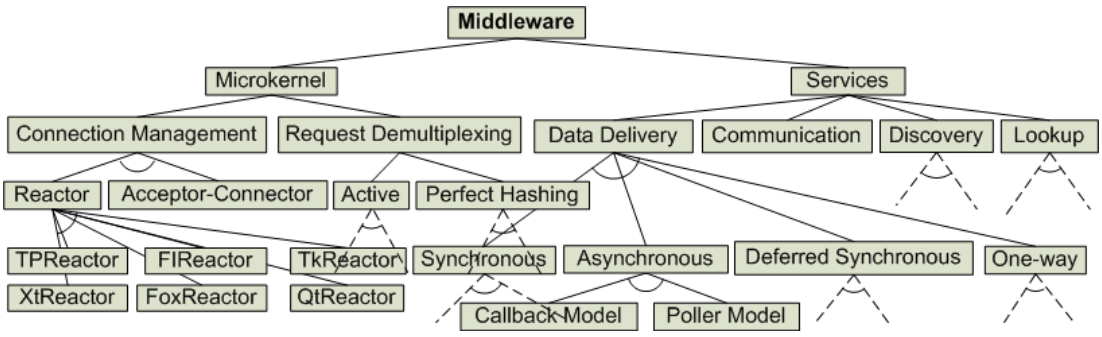


Figure 10: Middleware PIM Feature Model

Feature models of the general-purpose middleware as shown in Figure 10 tend to be very complex and huge making it very cumbersome to analyze for modularity. Fortunately, the feature sets for application variants are limited, which makes the mapping of concerns tangible within the middleware feature set. This helps us map known domain concerns to the middleware features in advance resulting in a $m : n$ correspondence between the domain concern model and middleware feature model. Thus, based on the domain concern model, the middleware feature model needs to be pruned to remove the unwanted features that do not map to the domain concerns. This is done through the feature model interpreters provided by the process.

The feature mapping wizard traverses an internal decision tree as shown in figure 11 to ask different questions to the application developer to infer the application variant characteristics. These characteristics include distribution features, such as client/server; concurrency features, such as single/multi-threaded, in that order. It asks questions ranging from coarse-grained ones like whether the application variant is client-server or peer-to-peer, to fine-grained questions like what kind of thread-spawning strategy is desired. Each

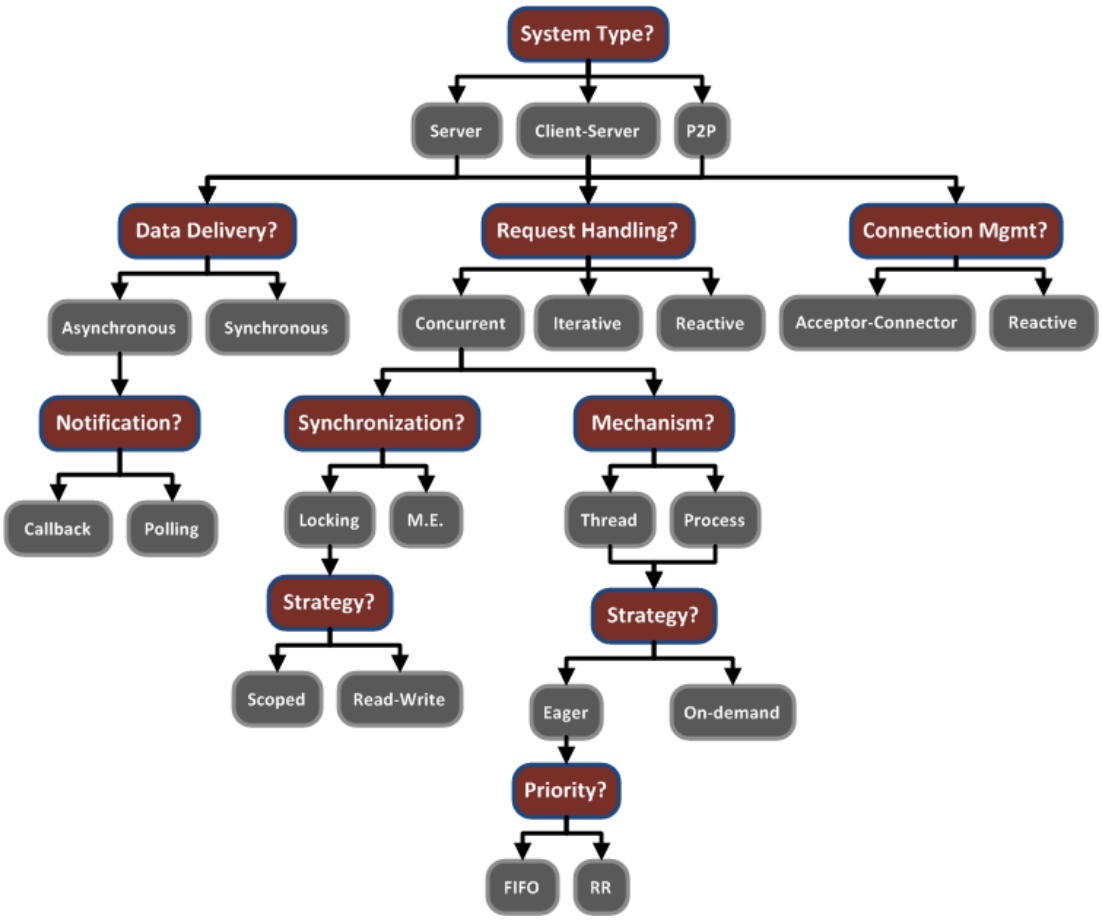


Figure 11: Decision Tree used by the Feature Mapper Wizard

coarse-grained answer scopes down the application characteristics based upon which the next fine-grained questions are asked that configure the application behavior.

After performing this mapping, a pruned PIM middleware feature set is generated that is mapped to the PSM middleware feature definitions through the transformations. We assume that the mapping of PSM middleware features to their PSM feature definitions i.e., source code, is already performed a priori by the middleware developer at design time thus enabling us to directly determine the PSM source code that implements the PSM middleware feature set. The wizard outputs the PSM source code hints that act as the starting point of the closure computation algorithm.

IV.3.2 Deducing the Specialization Context from System Models

Approach System invariant properties provide an indication of what features from the underlying middleware will be utilized by the applications. Since system invariant properties become evident only with every successive phase of application lifecycle, we classify the system invariants as (1) structural invariants, which are obtained from the structural composition of the system; (2) configuration invariants, which are obtained from the QoS configuration parameters selected for the middleware hosting platforms that specify the performance constraints. These constraints include latency, throughput, timeliness and reliability that are placed on individual application components, and their connections as well as the end-to-end workflows of components (known as component assemblies); and (3) deployment invariants, which are obtained from the resource allocations including the mapping of application software components to processors, platform bindings, endianness, languages, compilers, and collocation of different application software components.

An approach to identifying these invariants is through model interpreters that traverse the application models and establish the specialization context. Such a step eliminates the need for dedicated modeling annotations to identify the context within the application models. Most coarse-grained contexts can be detected automatically by examining the modeling structure and attributes but finer-grained contexts may need explicit identification.

Implementation We have developed a model interpreter that traverses the system models to detect the invariants that provide the specialization context. The interpreter makes use of well-defined matching patterns that were specifically developed for the PICML component-based DRE system modeling language [10] to ease the traversal to specific granularity levels (assembly, component, connection, port, interface, methods, parameters, config properties, etc) of the system model. The interpreter proceeds by starting from the highest level of granularity (assembly) to the lowest (parameters, configuration properties). Once it discovers the invariants, it gathers the configuration data associated with them that will be further used to deduce the specialization context. The interpreter maintains an extensible

catalog of these matching expressions that can be predefined by the model developer and if necessary can be further extended to accommodate the discovery of newer invariants.

IV.3.3 Inferring Specializations from Specialization Context

Approach Depending upon where they occur in the application model, the invariants that form the specialization context have certain semantics that implicitly determine the specializations that can be performed. For instance, application invariants such as repetitive tasks can provide a different specialization context based on the semantics they have, *e.g.*, periodic tasks can manifest in terms of periodic invocations that have synchronous request-response semantics which provide opportunities to optimize the redundant processing along the middleware call processing path. Since the specialization contexts map to different patterns of specialization, an extensible repository that can be queried for the right specializations is needed.

Implementation We have synthesized an extensible and intuitive repository called `SP-KBASE`, which serves as a knowledge base and is implemented as a complex multi-dimensional hashmap that stores the specialization patterns corresponding to the specialization. Note that a pattern also encodes the ordering in which individual specializations must be executed. Such an ordering is useful to the specialization staging algorithm that can correctly determine the next specialization to be performed. Another important piece of information that is stored is the incompatibilities or conflicts with other specializations in terms of common code paths or features being manipulated by them.

The snippet of `SP-KBASE` knowledge base shown in Table 2 has been developed based on the intuition of the middleware developers who have expert-level knowledge of the middleware design and implementation. The model interpreter from Step 1 parses the `SP-KBASE` using the uniquely inferred specialization contexts for each invariant and obtains the set of specializations. It then orders them based on the dependency information extracted from the dependency fields and emits out an ordered set of specializations that

Table 2: SP-KBASE: Extensible Catalog of Specialization Techniques

#	System Invariants	Optimization Principles	Specialization Techniques
S1	**Periodic Invocations**	P1, P4	Memoization
S2	**Fixed Priorities**	P1, P4	Aspect Weaving
S3	**Homogenous Nodes**	P1	Constant Propagation
S4	**Same Call Handler**	P1, P4,	Memoization + layer-folding
S5	**Known Implementation**	P2	Aspect weaving
S6	**Fixed Platform**	P2	autoconf

#	System Invariants	Specialization Joinpoints	Depends On	Conflicts
S1	**Periodic Invocations**	Request Creation	–	S3
S2	**Fixed Priorities**	Concurrency	–	S5
S3	**Homogenous Nodes**	Demarshaling Checks	–	S1
S4	**Same Call Handler**	Dispatch Resolution	S2	–
S5	**Known Implementation**	Framework Generality	–	S2
S6	**Fixed Platform**	Deployment Generality	S2, S5	–

Table 3: Performance Optimization Principles [131]

Principle	Description
P1	Avoid obvious waste
P2	Avoid unnecessary generality
P3	Don't confuse specification and implementation
P4	Optimize the expected case

are to be performed. It reports the incompatible set of specializations to the end-user or simply skips them if running in 'silent' mode.

CHAPTER V

AUTOMATED REALIZATION OF MIDDLEWARE SPECIALIZATIONS

The previous chapter developed a reasoning methodology for determining the features that are desired from the middleware which ultimately pruned the middleware feature set to only the features that are being directly used. It also presented a automated deduction methodology for identifying application invariants and inferring the specializations that are applicable to the specialization context of the detected invariants. However, the difficult task of the actual realization of the middleware specialization still remains which if performed manually becomes tedious and unproductive for the middleware developer.

This chapter addresses the second challenge outlined in Section I.2 Ű automated realization of middleware specializations. First, an overview of the existing research in the field of dynamic middleware adaptation techniques is presented. Second, a list of challenges that are still unresolved is presented. Finally, a solution approach is presented that provides two automated techniques for generation of specialization transformation directives and for transformation of the middleware build configurations.

V.1 Related Research

V.1.1 Aspect-oriented programming (AOP) for modularizing crosscutting concerns

AOP provides a novel mechanism to reduce footprint by enabling crosscutting concerns between software modules to be encapsulated into user selectable aspects. *FACET* [51] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. To support functionality not found in the base code, FACET provides a set of features that can be enabled and combined subject to some dependency constraints. By using AOP techniques, the code for each of these features can be

weaved at the appropriate place in the base code. However FACET requires manual refactoring of the middleware code into fine grained aspects for composition. FORMS does not require manual refactoring of the middleware code necessitated by the AOP techniques through its automated detection of features and feature dependencies within middleware source code.

Alexandersson *et al.* [4] 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++ [117]) 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.

JReplica [48] 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.* [2] 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). Although they provide "base" aspect with reusable pointcuts, concrete aspect implementation must be provided by the application developer.

V.1.2 Higher-level abstractions and generative mechanisms

The DADO project [96, 136] has shown how AOP can be used in a software development process to bypass the rigid layered processing by extending the middleware platform with new aspect-oriented modeling syntax and code generation tools. The FOCUS project [68] relies on manual identification of the application invariants, the specialization context and the specialization points within the middleware source, and manual writing of scripts to feed into a transformation tool that specializes the middleware sources.

Sevilla *et al.* [115] 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.

Automatic aspect generation is used in [123] to shift method call logging from FT-CORBA [92] 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.

The CORRECT [13, 21] project describes a project that is looking at applying step-wise refinement and OMG's Model Driven Architecture [91] 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.

Meta-object protocols (MOP) have been used [108, 125] to introduce fault-tolerance transparently in dependable systems. Taiani *et al.* [125] 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.

V.1.3 Limitations in related research

Even if AOP is shown to be effective, it still suffers from the overhead of excessive memory footprint due to the additional code required for instrumenting the aspects within the source codes. Moreover, the learning curve required leads to additional complexity in maintaining and debugging AOP programs. The FACET like AOP approaches additionally require redesigning and refactoring the traditional middleware into aspects. Our work on FORMS does not address the vertical decomposition problem in its entirety since it only accounts for coarser-grained features. As shown later, however, tools like FORMS can be leveraged to add a systematic process to the higher-level requirements reasoning and to customize the middleware build configurations.

Although the FOCUS tool itself is reusable, the specializations required manual identification of opportunities for specialization within the middleware code. Naturally, these solutions are not maintainable, reusable and extensible, and therefore cannot be easily transitioned to apply to different middleware and are cumbersome to evolve with the middleware. Similarly, the manual writing of bypass IDL files required by DADO and refactoring of middleware mandated by FACET hampers reusability.

Modelware demonstrates an interesting approach to specializing middleware, however, its success hinges on generating the entire middleware code from model artifacts. On the contrary our work is focused on specializing existing middleware code. The framework presented in this chapter incorporates the promising ideas from these related research while maximizing the opportunities for automation and reuse.

V.2 Unresolved Challenges

V.2.1 Challenge 1: Reducing Manual Effort in devising Specializations

In order to alleviate the manual efforts of the developers in designing and devising the middleware specializations, the following steps need to be resolved:

- ***Identification* of the Specializations Points within the Middleware Architecture**

  - Middleware is usually developed using the layered architectural style where each layer is composed using reusable components that are organized using sophisticated frameworks [55]. Each middleware layer therefore provides commonality as well as variability to the layer above it. While some of the middleware specializations can be ad-hoc, most of them really end up specializing these frameworks to remove unwanted commonality by pinning down the variability. These commonalities and variabilities usually form the source of performance bottlenecks since they comprise repetitive and redundant processing where the output provided by one layer to the layer above it does not change.. The variabilities usually manifest themselves into polymorphic behaviors programmed within the middleware patterns and frameworks in order to provide additional indirection that enables the required processing strategies to be chosen on-the-fly. Thus if these points are known in advance the additional indirection due to polymorphism can be eliminated. This requires recognizing the specialization points within the middleware source code. Current techniques for doing this involves annotating the source code with special labels/tags that map to individual specializations. These specialization tags are then need to be processed by special scripts or tools to transform the middleware code into a optimized code. Manually inspecting the vast middleware source code for identifying the specialization points and annotating them is a tedious, time-consuming and cumbersome task for the middleware developer. Moreover, as the middleware evolves, maintaining the locations of these specialization points and their semantics becomes an extremely difficult task.

- ***Realization* of Specializations -** In order to execute the specializations, the middleware code first needs to be transformed. The transformation tools require input transformation directives that are realized using the specialization tags to perform these source-to-source transformations.These source-to-source transformation directives can be realized using scripting techniques or advanced programming techniques like AOP. However, AOP techniques suffer from unbounded quantification which is not suitable for selectively transforming the middleware source that is to be specialized. Moreover, AOP techniques result into code bloating and testing nightmares for the developer. Additionally, it is tedious and cumbersome to manually write the complicated source transformation scripts requiring detailed knowledge of the middleware implementation architecture and can cause undesirable side effects within the middleware if developed incorrectly. Therefore, there is a need to automatically generate these transformation scripts correctly.
- ***Execution* of Specializations within the Middleware -** To transform the source code the identification of specialization points becomes crucial. Once the specialization points are identified, the middleware source needs to be transformed according to the optimizations programmed by each specialization. In order to execute the specializations, two steps are involved - transformation and staging. Once the transformation derivatives are realized, they need to be executed on the middleware source code. Tools need to be built that are able to automatically perform these transformations. Other alternative is to develop direct source transformation tools that inspect the sources, find the specialization points and perform the transformations. However such tools are difficult to implement and cumbersome to maintain. Once a middleware developer identifies the specialization points within the middleware architecture and the specializations that apply at those points, it is important to ensure that no two specializations conflict with one another in unpredictable ways. Specializations need to be compatible with one another at both the logical (architecture design) level as

well as physical (source code) level. At the logical level their compatibility can be checked through architectural constraint checks. However at physical level it is necessary to ensure that any two specializations that impact same or shared control flows ensure that correctness is ensured. This becomes difficult to verify since even if two specializations compatible at logical level can cause conflicts at the physical level. This incompatibilities need to be captured and codified in a form that is easily interpretable by specialization staging tools.

V.2.2 Challenge 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 proprietary middleware solution for each application domain is not a viable 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 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) [63] 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 [57, 80], 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* [25] offers a promising choice to address this question.

V.3 Automated Realization of Middleware Specializations

V.3.1 Identifying Specialization Points

Approach To identify the specialization points within the middleware we rely on the fact that most standards-based middleware implementations use frameworks that are based on well-known design patterns. Therefore it is possible to optimize the frameworks by specializing their constituent design patterns. Traditional frameworks and patterns are designed to be extensible by using indirections and dynamic dispatching through virtual hooks to support newer features that support newer functionality and processing methodologies. Examples of such frameworks are mainly transport protocol handlers, request demultiplexing and concurrency models. Rather than relying on the source code annotation alone to specify the specialization points, other techniques like code profiling and inspection, and feature

identification and composition can also be leveraged. Specialization points for functional artifacts can be identified by examining the design patterns in the middleware frameworks whereas the points for the execution threads of control can be identified by examining the middleware call paths. We leverage well-known optimization patterns (shown in Table 3) to specialize traditional middleware frameworks. A preliminary catalog identifying the middleware specialization points and the specialization techniques that apply to these points is shown in Table 2. We expect this catalog to be extended as new points are discovered.

Implementation To specify the specialization points, we first figure out the source code files that need to be transformed. The transformation rules only need to manipulate the source files that are actually implementing the salient framework features. To that end we have leveraged and extended our previous work, FORMS [26], to figure out the file dependency structure for the framework/pattern that needs to be specialized. The closure computation can take the required features as input and compute the closure set of source file dependencies that are independent of other closures. This gives us the files we need to process to perform the required source transformations. Based on the predefined set of

We have developed a *generic inspection engine* that uses source code inspection to identify the various individual components of a class such as header includes, forward declarations, scopes, methods, and data members. This pre-processing implicitly helps to identify the specialization points. Once the pre-processing is done, it provides the necessary information for the following operations – method removal, class movement, scope section replacement, checking for already defined methods, checking the order of typedefs and forward declarations needed for ensuring clean compilations – which form the basis of the specialization advice the algorithm generates.

V.3.2 Generation and Execution of Specialization Advice

Approach Once the specialization points are identified, to specialize the frameworks into their optimized equivalents, we require rules needed to perform the corresponding source-to-source transformations on the frameworks sources by using the available tools and scripts. One way of performing this is to represent these middleware and patterns in terms of high-level domain-specific architectural models [43]. Then perform model-to-model (M2M) transformations to convert these models into their optimal equivalents and later perform model-to-source (M2S) transformations to produce the optimized source. A drawback of this approach is the additional burden on the middleware developers to construct these models and two-level transformations [98]. Another way is to annotate the framework and pattern source code to identify the specialization points and write source-to-source transformations (S2S) [68]. However it is cumbersome to manually annotate and identify the design patterns and the corresponding implementing sources.

Algorithm 1 Generic Specialization Advice Generation Algorithm with the Pattern Specialization Plug-Ins

F : Framework Feature to be specialized/concretized.
M : Middleware Sources
D : Developer specified advice/specialized code
M_s : Specialized subset of Middleware Sources M
Input - F, M, D
Output - M_s (Initially empty)

begin
F_s := FIND all the framework files that contain the usage of the concrete *Framework Feature Class* f using *FORMS*
P_s := FIND the pattern implementation files using *FORMS*
P_d := COLLATE the data necessary for transformation using *FORMS* and D

{PATTERN SPECIALIZATION PLUG-IN}

REPLACE *Base Class* occurences with *Concrete Class* in all framework files F_s
REMOVE the *Includes* for the *Alternative Features* from the framework files F_s
REMOVE other *Alternative Features* from the build configuration using *FORMS* M_s
end

Implementation In order to avoid these cumbersome techniques, we have developed different generic transformation algorithms for optimizing/transforming each of the commonly used patterns (Bridge, Strategy, Template Method) in contemporary middleware. We have

opted to design the transformation algorithms to work with C++ – the most complex middleware implementation OO language being used. In case of other less complicated languages like C#, Java, etc., the algorithms will be much simpler and easier to implement. For example, unwanted indirections (virtual hook methods) in the Strategy pattern can be removed by collapsing class hierarchies, whereas dynamic dispatching (to concrete strategy/feature classes) in the Bridge Pattern can be eliminated by replacing with concrete instances of the strategy/feature implementations. On the other hand, the redundant computations in the middleware call processing path can be optimized by applying layer folding as shown in figure 12 and memoization optimizations.

The generic advice generation algorithm 1 generates rules at two levels: (1) the middleware framework level and (2) the constituent design patterns that implement the framework. The framework-specific transformations are performed to accommodate their corresponding constituent patterns-specific transformations. These include specializing the use of the pattern features in the other framework source code, particularly callbacks, feature instantiations and their usages, and the compilation of the framework code. Thus, the algorithm basically performs three major tasks by leveraging and extending the *FORMS* tool - (1) Determines all the framework implementing classes that utilize the feature to be specialized and leverages the corresponding specialization advice provided by the middleware developer, (2) It delegates the pattern specializations to the respective specialization plug-ins as described in algorithm 2, and (3) Specializes the build configuration files for compilation. We have developed similar algorithms for other commonly occurring design patterns within middleware frameworks such as Strategy, Adapter, Template Method, etc. which haven't shown in this chapter due to lack of space.

Any specialized code/data for the transformations is provided by the middleware developer since they can best determine how to optimize a particular code path within a particular framework. These rules are ultimately fed to the source transformation tools like FOCUS

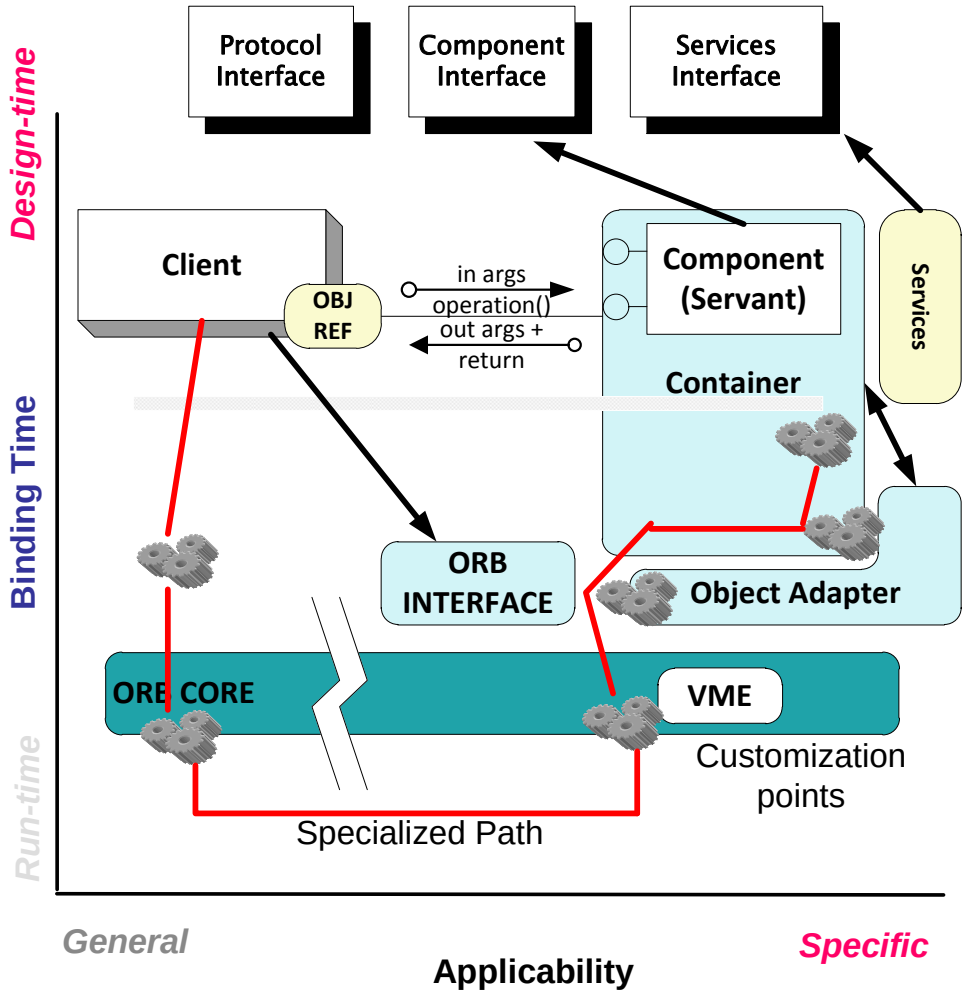


Figure 12: Middleware Specialization Path

[68] whose Perl scripts execute the transformations on the sources and subsequently the build specialization tools generate the specialized middleware source build configurations.

V.3.3 Discovering Closure Sets

Once the PSM source code hints that directly implement the domain concerns are determined, their dependencies on other code within the middleware needs to be determined. All such code that is interdependent on each other is what implements the domain concern.

Algorithm 2 Bridge Pattern Specialization Plug-In

{Eliminates Indirections - Removes Virtual Method Dispatches}
Input - P_s, M_s
begin
for each concrete *Feature Class Headers* $h \in P_s$ **do**
  ADD *Forward Declarations* & *Public Methods* from the *Bridge Impl Class*
  REMOVE *Base Inheritance*
  REMOVE all 'virtual' keywords
  CREATE *Concrete Feature Class* within the main class *Constructor*
  REMOVE all *Alternative Feature* references
end for
REPLACE the *Bridge Impl Class* occurrences with the *Concrete Feature Class* {also replaces the #includes} in all relevant files M_s
end

Algorithm 3 Template Method Pattern Specialization Plug-In

{Collapses Hierarchies - Fuses Derived class into Base class}
Input: F_s, M
Output: M_s (Initially empty)

begin
for each *Base Feature Class* $c \in P_s$ **do**
  REPLACE *Forward Declarations*, *Includes*, *Public Methods*, *Private Methods* and *Private Data* from the *Derived Feature Class*
  REPLACE *Base Constructor* methods with the *Derived Constructor* methods
  DEFINE 'typedef' *c* as the Derived Feature Class
  REMOVE all 'virtual' and pure 'virtual' keywords
  REPLACE *Base Feature Constructor* with *Derived Feature Constructor*
  COMMENT the *c* methods that are overridden in the *Derived Feature Class*
end forM_s
end

We call such a set of source files as a *closure set* in which there are no source file dependencies going out of the closure set. We differentiate between feature definition and feature implementation files. Feature definition makes it easier to identify and annotate features whereas feature implementations which capture the feature behavior may differ from one middleware implementation to another depending upon the language of implementation. Thus the closure computation identifies the set of dependent features definitions and their definitions, and composes them into a coherent and independent feature module.

Algorithm 4 Strategy Pattern Specialization Plug-In

Input: P_s
Output: M_s (Initially empty)

begin
for the concrete *Strategy Class* $f \in P_s$ **do**
  REMOVE *Base Inheritance*
  ADD *Forward Declarations*, *Includes*, *Public Methods* from the *Abstract Base Strategy Class*
  REMOVE all 'virtual' keywords from *Method Declarations*
end forM_s
end

We have designed a recursive closure computation algorithm that walks through the source code dependency tree and identifies the source that is dependent on the feature. However, opening each file on-the-fly and checking the dependencies is inefficient since it requires numerous I/O operations. Instead we run an external dependency walker tool like Doxygen or Redhat Source Navigator [31] to extract out the dependency tree.

Algorithm 5 Algorithm for Computing Closure Set for a product variant

1: M_s : Mapping of PSM middleware features to PSM definitions
2: F_p : Feature Set for Product Variant p
3: C_p : Closure set for product $p \in F_p$
4: C_f : Closure set for feature $f \in F_p$
5: C_s : Closure set for source hint $s \in M_s$
6: P_i : Pending set of feature implementations whose closure set needs to be calculated
7: **Input:** F_p, M_s
8: **Output:** C_p (Initially empty)
9: **begin**
10: $C_p := \emptyset$
11: **for** each feature $f \in F_p$ **do**
12: s := FIND feature definition from M_s for feature f
13: $C_f := \emptyset$
14: $C_s := \emptyset$
15: C_s := COMPUTE closure for feature definition s
16: $C_f := C_f \cup C_s$
17: P_i := FIND new feature implementation files for each feature definition in C_s
18: **while** $P_i is not empty$ **do**
19: $C_s := \emptyset$
20: C_s := COMPUTE closure for feature implementation file $i \in P_i$
21: $C_f := C_f \cup C_s$
22: $P_i := P_i \cup$ FIND new feature definition & implementation files that were found in the closure computation
23: **end while**
24: $C_p := C_p \cup C_f$
25: **end for**C_p
26: **end**

1. **Lines (1-7):** The middleware developer provides the mapping from the PIM middleware features to the PSM feature definition files i.e., PSM source hints in which the features are defined.
2. **Lines (10-17):** Once these PSM source hints are obtained the algorithm computes the closure set for each of the source hints. This step produces additional dependent PSM feature definition files which automatically form part of the closure set. Hence, their closure set need not be recalculated.
3. **Line (18):** The previous step gives rise to potentially more dependent feature definitions

that are not directly used by the product-line variant but required by the PSM source hints. The algorithm identifies the PSM feature implementation files for the dependent feature sets.

4. **Line (19):** The closure for the corresponding feature implementation files may need to be calculated. These new files form the pending implementation set and are added to the list of pending files whose closure needs to be calculated.
5. **Lines (20-26):** Now the algorithm iteratively calculates closure sets for each pending feature implementation file until all the pending implementation files are accounted for. The closure computation will always give rise to more pending feature implementation files as described in the 2nd step.

The closure sets corresponding to the application variants that are discovered in Section V.3.3 are different from cliques or maximally independent sets in graph theory. Closure sets, though transitive, are completely self-sufficient so they can also be called independent transitive closures.

V.3.4 Transparent Augmentation of Domain-specific Semantics in System Architecture

DRE systems may operate in particular modes that require certain specific operational semantics. For example, some applications may change modes depending on runtime changes or support dependability by failing over to an entirely new operational string (workflow of components). To achieve transparent provisioning of domain-specific semantics for component-based DRE systems, both the components and the middleware runtime infrastructure must be instrumented *automatically* in a coherent fashion.

To provide domain-specific dependability semantics, 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). 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 the primary technique applied to achieve application-transparent failover. Several flavors of interception such as linker-level [81], ORB-level [86], container-level [23], service-level [92], and aspect-oriented [63] 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.

V.3.4.1 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 V.4.7.1, 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.

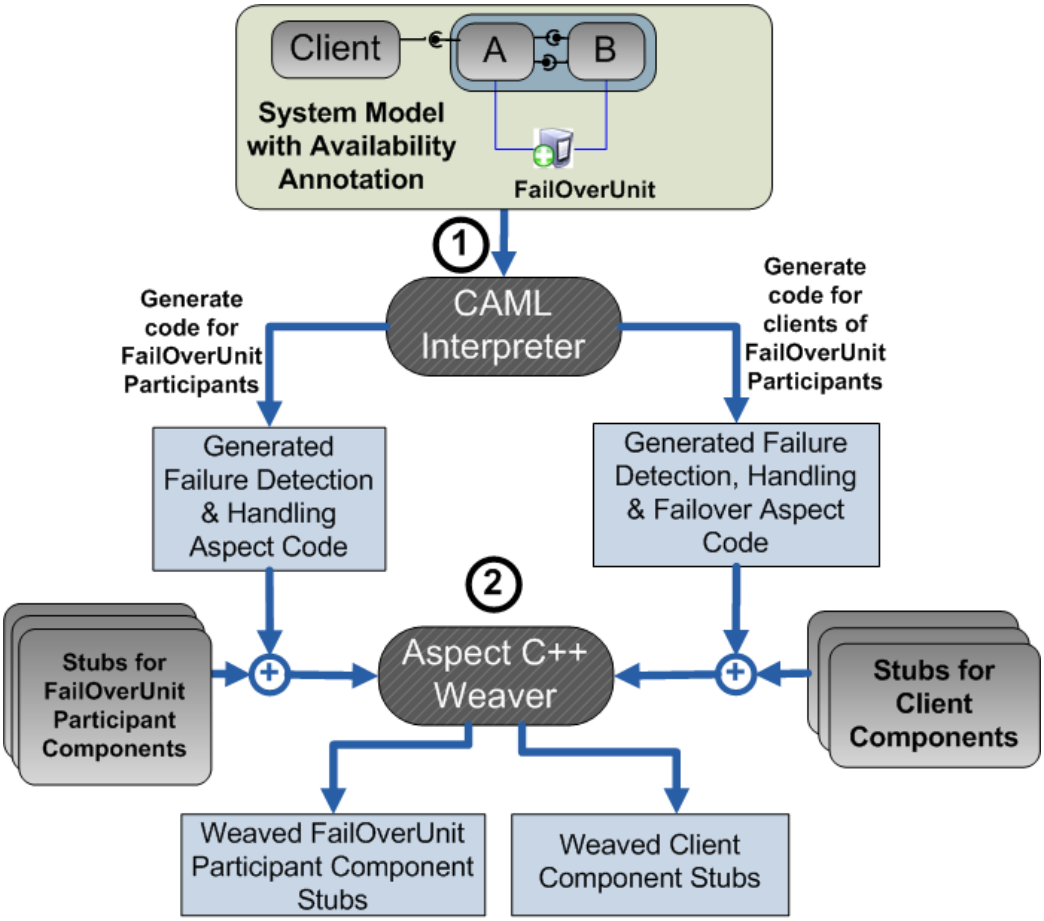


Figure 13: Automated Generation of Failure Detection and Handling Code

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) [63] support provided by the AspectC++ [117] 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.

V.3.5 Middleware Composition Synthesis through Build Specialization

Different middleware use sophisticated techniques to compile its source code into shared libraries. Some of these techniques rely on straightforward scripting *e.g.*, shell script,

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

batch files, perl scripts, or ANT scripts while some of them rely on descriptor files such as make file system or advanced cross-compiler build facilities like MPC (Make Project Creator) [38]. We leverage the MPC cross-compiler facility since it supports multiple compilers and IDEs and is therefore more generic and widely applicable for synthesizing middleware shared libraries written in different programming languages.

The MPC projects of the general-purpose middleware do not necessarily represent the feature modularization per se. The closure sets are converted into MPC files for synthesis of the specialized middleware represented by the closure sets through the respective language tools. These MPC files are specialized versions of the combination of the original MPC files of the general-purpose middleware and are the real representation of feature modularization in terms of application variant requirements.

V.4 Evaluation

V.4.1 Logging Server Case Study

In order to explain and evaluate the FORMS middleware specialization process, we use a motivating example of a application variant of networked logging servers as shown in Figure 14. We choose this particular application variant since logging various status and error messages is a very frequent and widely used facility for monitoring the system performance as well as system survivability in different domains such as enterprize, or distributed real-time and embedded systems like shipboard computing and mission critical aviation software.

A logging server has different performance requirements depending upon the type of application that is using the logging facility. Depending upon the application domain the need for logging varies from sporadic to frequent logging. Enterprize applications may require sporadic logging where logging is restricted to mostly error and status messages whereas certain high security mission critical application that are susceptible to infiltrations may require more detailed logging traces of the system behavior in order to detect

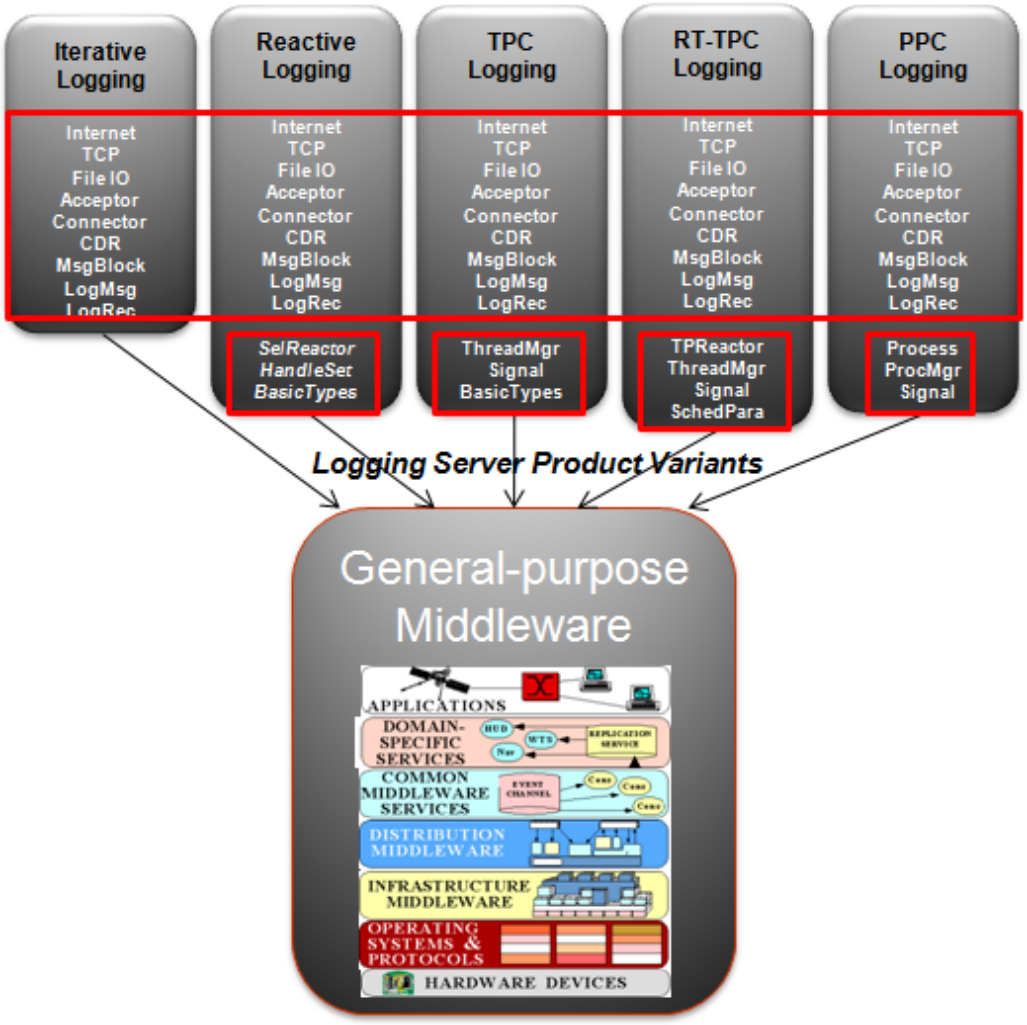


Figure 14: Logging Application Variant

discrepancies and errors that may lead to discovering an impending or in-progress security attack. Hence sporadic logging may require iterative or reactive logging servers whereas frequent logging may require multithreaded or multiprocess logging servers.

We evaluate FORMS by modeling a application variant of networked logging applications based on contemporary, widely used communication middleware such as ACE [54]. ACE is a free, open-source, platform-independent, highly configurable, object-oriented (OO) framework that implements many core patterns for concurrent communication in software. It enables developing product variants using various types of communication paradigms such as client-server, peer-to-peer, event-based, publish-subscribe, etc. Within

each paradigm it supports various models of computation (MoC) which are highly configurable for different QoS requirements. We have designed the networked logging application variant servers based on the client-server paradigm with individual models conforming to various MoCs including iterative, reactive, thread-per-connection (TPC), real-time thread-per-connection (RT-TPC) and process-per-connection (PPC). Each application variant may in turn have different QoS requirements for event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization.

Figure 14 shows the representation of the logging server application variant in terms of commonality and variability of the features. We have showcased only those features that are required since we are not interested in how the individual logging server variant is implemented but rather what PIM features it desires from the underlying middleware platform.

V.4.2 Evaluation of the Closure Computation Algorithm

Table 4: Outcome of applying FORMS to a Product-line of Networked Logging Applications

Networked Logging Applications Application Variant		**Outcome of Closure Computations**		**Synthesized Middleware**
application variant (described in Domain Concerns)	***# of Middleware PIM Features***	***# of Middleware PSM Features***	***Size of Closure Set (PSM files)***	***Static Footprint (KB)***
Simple (Iterative) Logging	9	107	502	1,456
Reactive Logging	12	109	502	1,456
Thread Per Connection Logging	11	176	502	1,456
Real-Time Thread Per Connection Logging	12	178	502	1,456
Process Per Connection Logging	12	120	508	1,500

By creating specialized variants of ACE middleware for different types of logging servers, the profiling tools estimate the memory footprint savings, dependent middleware features, source files that implement the features, and exercise unit tests to determine

whether the expected performance is met. We showcase the compile-time metrics that result from middleware specialization.

V.4.2.1 Footprint and Feature Reductions

Our experiments provide interesting insights about the relationship between the number of middleware features being used and the footprint of the synthesized middleware. The ACE middleware is implemented in *1,388* source files and *436* features with a resulting footprint of *2,456 KB*. Table 4 shows that the algorithm has achieved significant optimizations - a *64%* reduction in the number of source files used, a *60-76%* reduction in the number of features used, and a *41%* reduction in memory footprint. The ACE middleware was compiled on Windows using Visual Studio 8.0 compiler. Similar improvements were also observed with GNU GCC compiler on Linux.

Table 4 also shows that the PLE variants share many middleware PIM features as verified by the almost similar footprint measurements (*1,456 KB - 1,500 KB*). This means that the middleware forms a homogenous core that supports the entire application variant. In this case, a single version of the ACE middleware could be synthesized for the entire application variant instead of synthesizing individual variants for each product. Thus, the process also provides guidelines as to whether to synthesize individual variants or a single variant for the application variant thereby eliminating the need to provide and maintain multiple specialized middleware variants.

V.4.2.2 Modularization Discrepancies

On the other hand as shown in Figure 15, there is a wide disparity between the number of PSM middleware features required by the individual product variants (107-178) variants and the PSM source files (502-508) implementing them. More specifically after inspecting the individual application variant's generated MPC build configuration, there were some unused PSM features that percolated into the feature modules of a application variant.

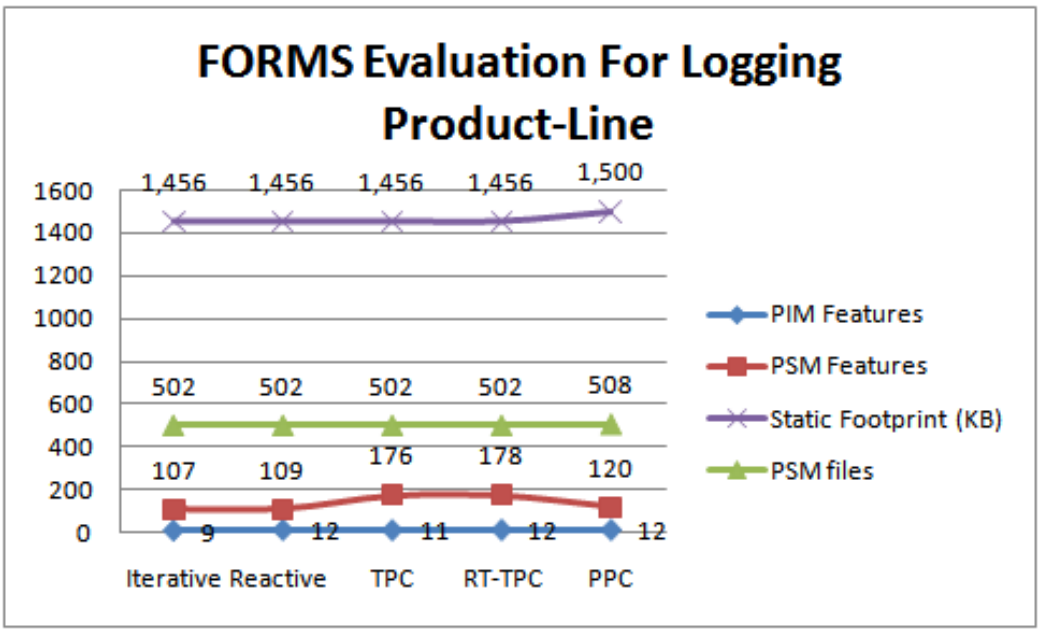


Figure 15: Modularization Disparities

This means that there are several unused middleware features that find their way in the specialized middleware for the Iterative, Reactive and PPC product variants that originally required fewer features.

V.4.3 Additional Insights provided by the algorithm

The closure computation algorithm can be enhanced to give additional insights to middleware developers about the middleware modularization, ease of testing and maintenance overheads.

1. **Discovering Modularization Discrepancies:** The reason for the modularization discrepancies described in section V.4.2.2, are due to the physical implementation dependencies between the logical feature modules. These results from the conflicts between the design goals envisioned by the middleware designers and the implementation goals of the middleware developers. This happens if a single PSM implementation source file implements more than one PIM feature or vice versa. Thus the logical PIM feature independence does not always translate to their actual physical

PSM implementation independence. Thus even though general-purpose middleware is designed in a modular way, the modularity does not manifest exactly in the same way in their implementations of the middleware layers. The algorithm can thus provide a guideline to the middleware developers to detect and break unnecessary dependencies within their source code and thereby reduce the tight coupling between the modules within the middleware layers.

2. **Automated Test Case Selection:** The algorithm reduces the amount of features, in turn reduces the functionalities that are expected from the middleware. Thus it can enable automatic test case selection of functional unit tests in order to alleviate the testing and maintenance overhead for the middleware developers

3. **Discovering Middleware Core:** The algorithm helps in identifying the core middleware features needed by the application variant. The algorithm can take a multiset intersection of all the closure sets that are generated for the different application variant variants. This intersection represents the commonality whereas the rest of the features represent the variability. Thus, closure computation can potentially figure out the differences between the logical middleware core as designed and envisioned by the middleware architect and physical middleware core estimated by the closure computation.

V.4.4 Validation of the Algorithm

As middleware is statically specialized, checking the correctness of its functionalities becomes paramount. In this case a simple successful compilation of the specialized middleware and shared library generation are not sufficient. It becomes necessary to verify the runtime correctness of the specialized middleware through exhaustive testing processes. We validated the closure computation methodology by re-executing the tests on the specialized middleware that were originally designed for the general-purpose middleware. However,

we also ensured that the tests that have been invalidated due to the missing features from the specialized middleware are pruned away and not re-executed.

V.4.5 Evaluation of the Generative Middleware Specialization Algorithms

Since middleware specialization is a software engineering process, we demonstrate its applicability and evaluate its merits along the following dimensions: (1) We first show how the algorithms can be applied to specialize middleware for a representative DRE system case study; (2) We show the savings in effort (and hence improvement in productivity) on the part of a DRE system developer accrued by using the algorithms in contrast to manually performing the specializations; and (3) We show the improvement in latencies and static and runtime memory footprints of the specialized middleware version compared to traditional middleware.

V.4.6 Illustrating the generative algorithms on a DRE Case Study

We now show how the algorithms are applied to specialize middleware for a representative DRE system case study using the specializations cataloged in the knowledge base `SP-KBASE` shown in Table 2.

V.4.6.1 Avionics: The Boeing Boldstroke Basic Single Processor (BasicSP) Application

Scenario Description BasicSP (Basic Single Processor) is a scenario from the Boeing Bold Stroke avionics mission computing application variant [116], which is a component-based, publish/subscribe platform built atop the TAO Real-time CORBA Object Request Broker (ORB) [114]. It supports the Boeing family of fighter aircraft, including many product variants, such as F/A-18E, F/A-18F, F-15E, F-15K, etc. Figure 16 illustrates the *BasicSP* application scenario, which is an assembly of avionics mission computing components reused in different Bold Stroke product variants.

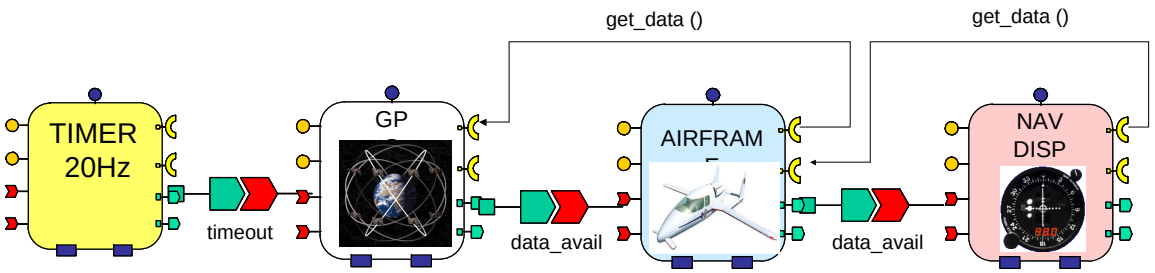


Figure 16: The Basic Single Processor (*BasicSP*) Application Scenario

BasicSP involves four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays at a rate that is configurable. The time to process inputs to the system and present output to cockpit displays should thus be less than the rate, which as shown in the figure is a single 20 Hz frame.

Problems The real-time concerns are orthogonal to the traditional horizontal middleware decomposition. In the BasicSP scenario the real-time requirements of predictable latency of 20Hz is desired by each of the individual components so that the aircraft pilots receive their location in real-time. At the same time, these application invariants are not known in advance so they cannot be automatically used to deduce the specializations that can be potentially performed. Moreover, the system requirements may change if the system is deployed in a different physical domain or a different aircraft. For example, a different variant of this scenario for different customer requirements, however, may use different framework components or may send different events to consumers or may service operations via different request dispatchers or may run on nodes with different byte orders, but with the same compiler/middleware implementation, in which case data need not be aligned. These changing requirements render point specialization solutions useless and therefore the need for a systematic, extensible and reusable specialization approach becomes even more apparent.

V.4.6.2 Applying Generative Specializations to Specialize Middleware for BasicSP

We show how the model interpreters traverse the BasicSP model to realize the specializations. Figure 17 shows the specialization context and specialization points with BasicSP.

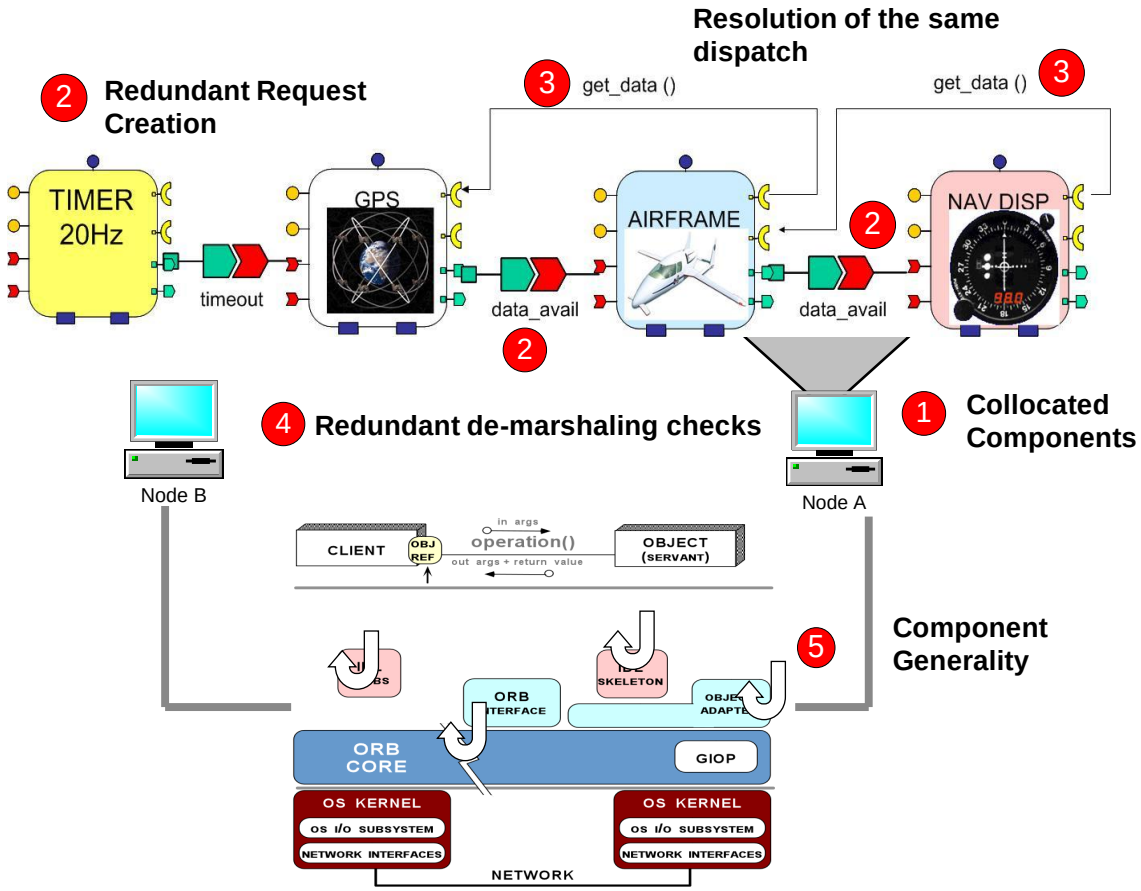


Figure 17: Specialization Context in BasicSP

Applying Step 1 (Deducing the Context) *Structural Invariants* - The *BasicSP* case study uses a "push-event, pull-data" communication model, which forms the basis of the structural composition of the system. On receiving an event, the `Airframe` and `Nav_Display` components repeatedly use the same `get_data()` operation to fetch new GPS and Display updates, respectively. In a connection between `GPS` and `Airframe` components, therefore, the `get_data()` operation is sent and serviced by the same request dispatcher.

Configuration Invariants - In *BasicSP*, the connection properties such as the pulse rate

of 20 Hz, and corresponding data delivery deadlines form the application QoS configuration model. In this case study, the processing rate is fixed at a maximum latency rate of 20 Hz, the transport protocol used is VME backplane, and the request demultiplexing mechanism within the middleware is reactive.

Deployment Invariants - The target nodes on which the *BasicSP* components are deployed (not shown in the Figure) have the same byte order (endianness) since the processors used in this case study are homogeneous.

Applying Step 2 (Inferring the Specializations) *Structural Invariants* - The *BasicSP* push-event, pull-data communication model imposes the need for features that support event communication as well as request-response semantics from the underlying middleware. Since there are no concurrent requests, no concurrency support is needed of the middleware, and hence we can deduce only a single request dispatcher is involved which translates to the 'S4' specialization in Table 2.

Configuration Invariants - In *BasicSP*, the constant pulse rate of 20 Hz indicates the periodic nature of events and the rate at which data will be pulled. It also indicates the deadline for communication and computation for the periodic task. Periodicity maps to the 'S1' specialization. Since the period of the end-to-end task is fixed, such hard real-time requirements call for features that support fixed priority scheduling translating to the 'S2' specialization. In RTCORBA, the feature that supports this requirement is the `SERVER_DECLARED` model. Since no other priorities and concurrent requests are involved, it needs a simple reactive event demultiplexing and single threaded event processing model within the underlying middleware. Hence, it calls for a single threaded Select Reactor-based [112] request handling. For RTCORBA, this property indicates there is no need for the thread pool mechanisms. Moreover, since only one transport mechanism is used, there is no need for sophisticated software solutions that support pluggable transport protocols, such as the extensible transport mechanism in RTCORBA. Both these invariants translate to the 'S5' specialization.

Deployment Invariants - In *BasicSP*, since there is no need for byte order checking and codeset negotiations (by virtue of using a homogeneous set of processors), there is no need for marshaling data according to the byte order and data encoding rules including those involving alignment of data along word boundaries. Similarly, there is no need for mapping priorities between sending and receiving components. All these translate to the 'S3' specialization.

Applying Step 3 (Identifying Joinpoints) The identification of specialization joinpoints for the middleware through optimizing the design patterns is automatically performed by the *generic inspection engine* as described in Section V.3.1. The necessary annotations get automatically inserted in the pattern implementation sources which are recognized by the FOCUS source code manipulation tool. However, for the other non-structural specializations, the annotations need to be manually defined by the middleware developer since those require explicit specification of the specialized advice that may exhibit different behavior from the original code at which it is applied.

Applying Steps 4 and 5 (Advice Generation and Execution) For lack of space we do not show the complete generated specialization advices. Instead, Listing 1 shows a snippet for the rules that get generated for the bridge pattern corresponding to the steps specified in the Algorithm 2. The FOCUS tool subsequently specializes the middleware code.

V.4.6.3 Improvements in Developer Productivity through Auto-Generation

We leverage FOCUS [68] to execute the generated specialization advice on the middleware source code. The FOCUS source transformation rules for specializing the design patterns and middleware frameworks are represented in XML. Manually writing these rules by the middleware developer on a per instance basis is not only cumbersome and excessively tedious but also complex to maintain as the middleware source code evolves. Auto-generating them using the generative algorithms as described in Section V.3.2 alleviates the burden on the developers as well as makes them easy to extend and maintain. Table 5

Listing 1 Generated Transformation Rules for Bridge Specialization

```
<module name="ACE/ace">
  <file name="Select_Reactor_Base.h">
    <add>
      <hook>REACTOR_SPL_INCLUDE_FORWARD_DECL_ADD_HOOK</hook>
      <data>class ACE_Sig_Handler; </data>
    </add>
    <remove>virtual</remove>
    <remove>: public ACE_Reactor_Impl</remove>
    <remove>#include "ace/Reactor_Impl.h"</remove>
    <substitute>
      <search>ACE_Reactor_Impl</search>
      <replace>ACE_Select_Reactor_Impl</replace>
    </substitute>
  </file>
  <file name="Reactor.cpp">
    <add>
      <hook>REACTOR_SPL_CONSTRUCTOR_COMMENT_HOOK_END</hook>
      <data> ACE_NEW (impl, ACE_Select_Reactor); </data>
    </add>
  </file>
</module>
```

shows how many lines are auto-generated on a per-pattern basis and how these translate to cumulative savings for the entire middleware framework that is implemented using that pattern.

Table 5: Middleware Developer Effort Savings

Design Pattern (Middleware Framework)	**#lines Generated**	**#lines Handwritten**	**% Savings**
Bridge (Reactor)	115/443	17	96.16 %
Strategy (Flushing)	29/201	4	98.01 %
Strategy (Wait On)	29/141	4	97.16 %
Template Method (Pluggable Protocol)	172/974	25	97.43 %

However, developers will still need to provide the specialized code if they wish to specialize a particular middleware call path in their own way. This specialized code is applied like an *aspect advice* at the code joinpoints specified through annotations. As shown, the

auto generation almost completely eliminates the burden of manually writing the transformations and figuring out the specialization joinpoints with savings in excess of $\tilde{9}7\%$. For the sake of terseness, we have only shown a few of the frameworks that were optimized.

V.4.6.4 Empirical Evaluations

We evaluated the outcome of applying the generative algorithms by measuring the following criteria: (1) the static footprints of the middleware binaries, (2) dynamic footprints of the BasicSP applications, (3) the average latencies of requests, and finally (4) the overall throughput of the application components. We have applied the generative algorithms to the widely used TAO Real-time ORB implementation for DRE systems software. Table 6 reveals that the resultant savings are substantial for DRE applications meant to be deployed on resource constrained embedded devices. The dynamic footprints are a lot higher (5x) than the static footprints of the middleware binaries since the specialized middleware binaries were generated for each BasicSP application components.

Table 6: Middleware Performance Improvement Metrics

Metrics	**Before Specialization**	**After Specialization**	**% Savings**
Footprint (Static)	3,226 KB	2,082 KB	35.4 %
Footprint (Dynamic)	13,588 KB	10,657KB	21.57 %
Average Latency	3367 μs	2160 μs	35.84%
Throughput	0.26 reqs/s	0.41 reqs/s	36.59%

V.4.7 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.

V.4.7.1 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 18 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 18, 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.

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

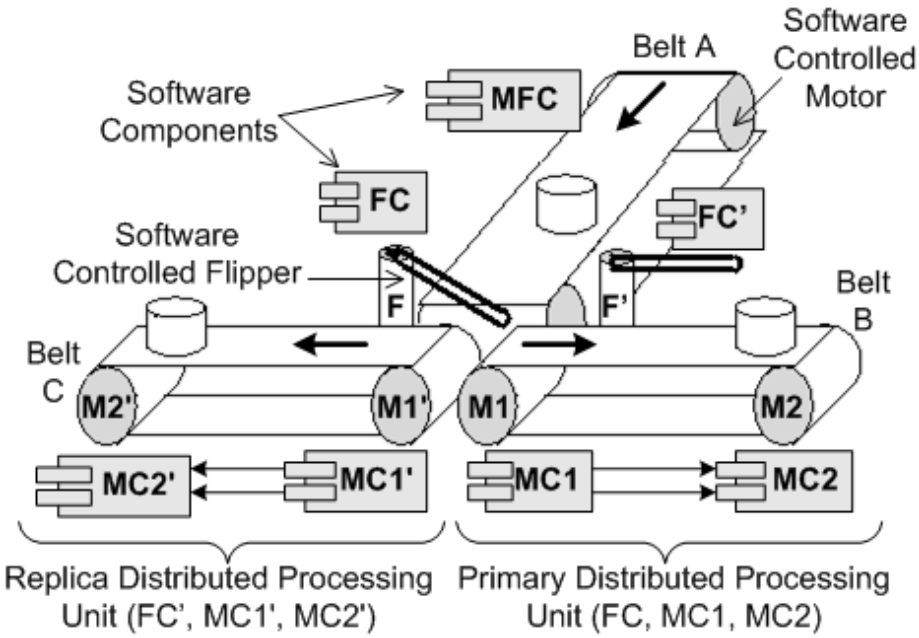


Figure 18: A Distributed Processing Unit Controlling Conveyor Belts

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 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 7: Savings in Fault-tolerance Programming Efforts in Developing MHS Case Study Without/With GRAFT

V.4.7.2 Evaluating savings in effort to specialize middleware

A significant reduction in programming efforts is achieved due to automatic generation of code that handles failure conditions at runtime in the MHS system. 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 7 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.

V.4.7.3 Qualitative validation of runtime behavior

Figure 19 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.

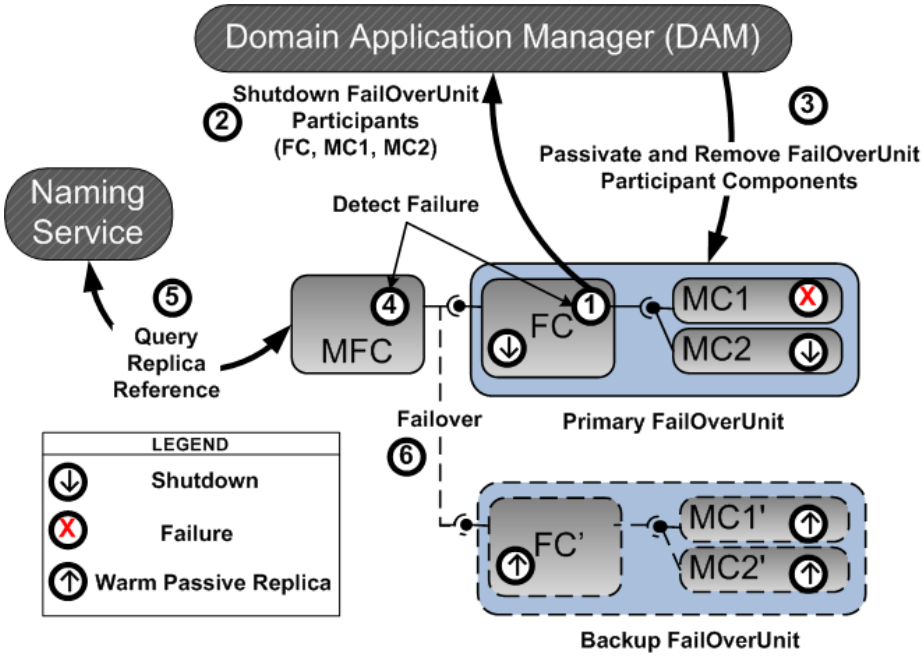


Figure 19: Runtime Steps Showing Group Recovery Using GRAFT

a. As shown in Figure 19, 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 19.

b. 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.

c. DAM instructs the containers hosting the primary DPU components (FC, MC1, and MC2) to passivate and remove the components.

d. Removal of FC component triggers a system-level exception at the MFC component, which is again caught by the specialized stub at MFC-side.

e. 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.

f. 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.

CHAPTER VI

RELIABLE DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS THROUGH SAFE MIDDLEWARE ADAPTATION

The past chapters focused on developing a taxonomy for categorizing and reasoning about middleware specializations and realized a feature-oriented, automated and generative process for inferring middleware features, deducing application invariants, and ultimately synthesizing the middleware specializations, respectively. Although the presented techniques significantly reduce the developer efforts involved in driving and synthesizing middleware specializations, they do not adequately address the runtime issues that arise when middleware needs to adapt to satisfy the stringent requirements of DRE systems.

This chapter addresses the final challenge outlined in Section I.2 Ű safe adaptation of middleware to failures within stringent real-time QoS constraints. First, an overview of the existing research in the field of specialization implementation techniques is presented. Second, a list of challenges that are still unresolved is presented. Finally, a solution approach is presented that enables safe and predictable middleware adaptation to failures through a distributed resource monitoring framework and the corresponding resource aware middleware adaptation algorithm that accounts for failure type, granularity and failover replica placements.

VI.1 Related Research

In this section we discuss the existing body of research in the area of adaptive fault tolerance in distributed real-time and embedded systems and compare and relate our work on SafeMAT. We categorize adaptive fault tolerance research in following areas:

VI.1.1 Dynamic Scheduling

Common methodologies to leverage the slack in execution schedule have focussed on dynamic scheduling depending upon the runtime conditions. The Realize middleware [58] provides dynamic scheduling techniques that observes the execution times, slack, and resource requirements of applications to dynamically schedule tasks that are recovering from failure, and make sure that non-faulty tasks do not get affected by the recovering tasks.

VI.1.2 Resource-aware Adaptations

Resource-aware Adaptations: The DARX framework [74] provides fault-tolerance for multi-agent software platforms by focusing on dynamic adaptations of replication schemes as well as replication degree in response to changing resource availabilities and application performance. [44] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. Research performed in AQUA [69] dynamically adapts the number of replicas receiving a client request in an ACTIVE replication scheme so that slower replicas do not affect the response times received by clients. Eternal [59] dynamically changes the locations of active replicas by migrating soft real-time objects from heavily loaded processors to lightly loaded processors, thereby providing better response times for clients. FLARe [8] proactively adjusts failover targets at runtime in response to system load fluctuations and resource availability. It also performs automated overload management by proactively redirecting clients from overloaded processors to maintain the desired processor utilization at runtime. [42] focuses on an adaptive dependability approach by mediating interactions between middleware and applications to resolve constraint consistencies while improving availability of distributed systems.

VI.1.3 Real-time fault-tolerant systems

Real-time fault-tolerant systems: IFLOW [20] and MEAD [101] use fault-prediction techniques to reduce fault detection and client failover time to change the frequency of backup replica state synchronization to minimize state synchronization during failure recovery, and by determining the possibility of a primary replica failure and redirecting clients to alternate servers before failures occur, respectively. The Time-triggered Message-triggered Objects (TMO) project [64] considers replication schemes such as the primary-shadow TMO replication (PSTR) scheme, for which recovery time bounds can be quantitatively established, and real-time fault tolerance guarantees can be provided to applications. FC-ORB [135] is a real-time Object Request Broker (ORB) middleware that employs end-to-end utilization control to handle fluctuations in application workload and system resources by enforcing desired CPU utilization bounds on multiple processors by adapting the rates of end-to-end tasks within user-specified ranges. Delta-4/XPA [104] provided real-time fault-tolerant solutions to distributed systems by using the semi-active replication model. Other research [61] uses simulation models to analyze multiple checkpointing intervals and their effects on fault recovery in fault-tolerant distributed systems.

VI.1.4 Need for Safe Fault Tolerance

For the hard real-time DRE systems, applying dynamic load balancing, dynamic rate and scheduling adjustments, adaptive replication and redundancy schemes add extraneous dynamism and therefore potential unpredictability to the system behavior. Altering the redundancy strategies require altering the real-time schedules which is not acceptable for hard real-time systems that are strictly specified. Constantly redirecting clients upon overload and promoting backups to primaries adds unnecessary resource consumptions for fixed priority systems. Such approaches do not attempt to minimize the number of resources used; their goal is to maintain service availability and desired response times for the given number of resources in passively replicated systems. However, in hard real-time systems

exceeding the RMS bound of 70% of the processor utilization is not a concern as the tasks are guaranteed to not be preempted until their allocated quantum is over. So as long as task utilizations are guaranteed to be under 100% processor load, their deadlines and profiled WCETs are guaranteed to be satisfied. In SafeMAT we guarantee through exhaustive application performance profiling by establishing runtime utilization and failover overhead bounds that the dynamic failure adaptations will not violate the real-time deadlines and overload the resources. Moreover, as the system resources are over-provisioned we use semi-active replication which subsumes the need for expensive state-synchronization and load balancing mechanisms.

VI.2 Unresolved Challenges

This section brings out the challenges that motivate the need for the three primary vectors of the SafeMAT middleware presented in this chapter. Before delving into the challenges, we present a model of the system and the underlying platform we consider in this chapter.

Platform Assumptions Our research focuses on a class of DRE systems where the system workloads and the number of tasks in the individual subsystems that make up the DRE system are known *a priori*. Examples of individual subsystems that make up DRE systems include tracking and sensing applications found in the avionics domain, the automobile system found in the automotive domain (*e.g.*, reacting to abnormalities sensed by tires), conveyors systems in industrial automation (*e.g.*, periodic monitoring and relaying of health of physical devices to operator consoles), or resource management in the software infrastructure for shipboard computing domain. These systems demonstrate stringent constraints on the resources that are available to support the expected workloads and tasks. For this chapter we focus on the CPU resource only.

In our research we assume that the individual subsystems of the DRE system use the

ARINC-653 [6] model in their design and implementation because of its support for temporal and spatial isolation, which are key requirements for real-time systems. ARINC-653 uses fixed-priority preemptive scheduling where the platform is specified in terms of modules that are allocated per processor which in turn are composed of one or more partitions that are allocated as tasks. Each partition has its own dedicated memory space and time quantum to execute at the highest priority such that it gets preempted only when its allocated time quantum expires. Multiple components or subtasks can execute through multitasking within each quantum. For evaluating our design of SafeMAT and experimentation, we have leveraged an emulation [34] of the ARINC-653 specification.[1]

Section I.2 highlighted the need for resource-aware and safe adaptive fault tolerance for DRE systems that also incorporated principles of software health management. Realizing these objectives is fraught with a number of challenges, which are presented below. The three primary vectors of our SafeMAT solution stem from the need to resolve these challenges.

VI.2.1 Challenge 1: Identifying the Opportunities for Slack in the DRE System

As noted in Section I.2, DRE systems are composed often from individual legacy subsystems. Many of these subsystems comprise real-time tasks with strict deadlines on their execution times. To ensure the safety- and mission-criticality of these subsystems, they are configured with predefined execution schedules computed offline that are fixed for their execution lifetime once they are deployed in the field. This ahead-of-time system planning ensures that such subsystems will behave deterministically in terms of their expected behavior and their provided services, and the critical tasks with hard real-time requirements will always satisfy their deadlines. To achieve this predictability, these subsystems are over-provisioned in terms of the allocated time and required capacity of resources. Naturally, for most of the time many of these resources remain under-utilized and hence provide

[1] We used the emulation environment since it was readily available to us, and has been used previously to demonstrate key ideas of software health management for avionics applications.

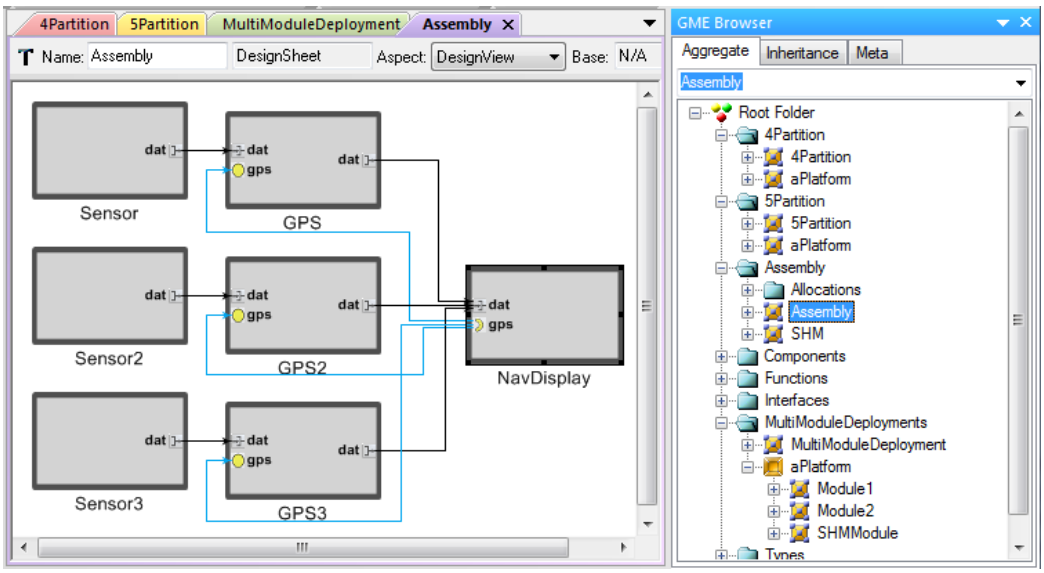


Figure 20: GPS (BasicSP) Subsystem Assembly

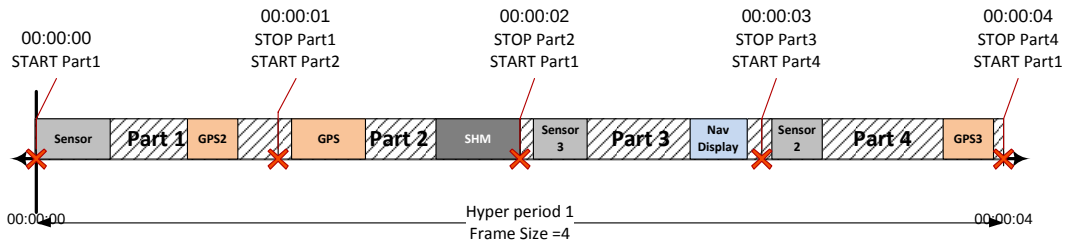


Figure 21: Slack in GPS Schedule

an immediate opportunity to host the fault tolerance mechanisms needed for DRE systems. However, due to the dynamic nature of faults, the amount of slack available in each subsystem may vary at runtime thereby rendering any offline computation of slack for DRE fault tolerance useless. Therefore, there is a need to obtain a runtime snapshot of available slack in the system that then will enable the runtime execution of fault tolerance mechanisms for DRE systems. Such a monitoring capability must provide real-time information while at the same time not impose any significant overhead on the system. Section VI.3.3 presents our solution to a scalable Dynamic Resource Monitoring (DRM) capability in SafeMAT. In the context of our ARINC653-based scheduling of the DRE systems, DRM is not only

able to obtain the actual CPU utilizations of the partition tasks but also of the subtasks (*i.e.*, application components) that are allocated within the partition.

VI.2.2 Challenge 2: Designing Safe and Predictable Dynamic Failure Adaptation

Failures in DRE systems may manifest in different types and granularities. For example, some component failures may be logical or critical. The granularity of failures could be a component, group of components (subsystem), processes or processors. Moreover, the induced interdependencies in DRE systems due to composition of individual subsystems may lead to cascading failures of the dependent components (domino effect). Such an effect has the potential to increased deadline violations and over-utilization of system resources. Statically defined fault tolerance schemes will not work to completely handle these kinds of failures. Dynamic failure adaptation techniques can provide better capabilities to tolerate different kinds and granularities of failures, and can achieve better resource utilizations. However, given the criticality of hard real-time system execution, the failure adaptations that can be performed need to be safe and predictable. By utilizing the slack (which is obtained using the DRM capabilities), we can provision dynamic fault adaptation, however, we must ensure that the execution deadlines are not violated while achieving such runtime adaptations. Consequently, it is necessary to reduce the amount of recovery, which calls for failure detection and mitigation mechanisms that are fast and lightweight in terms of their space and runtime overhead as well are adaptive to the failure type and granularity, and component replica placements. Section VI.3.4 describes the adaptive fault tolerance mechanism supported by SafeMAT.

VI.2.3 Challenge 3: Validating System Safety in the Context of DRE System Fault Tolerance

Although it may be feasible to design dynamic fault tolerance techniques for DRE systems by leveraging the slack, there is no easy approach to validate the safety and correctness

of the resulting system and it is difficult to develop a mathematical proof of correctness of the system due to its dynamic nature. Thus, there is a need for a scalable and accurate capability that can validate the overall DRE system for safety and predictability. SafeMAT provides a framework to profile a DRE system to validate if the real-time properties are met in the context of faults that can be artificially injected into the system. Section VI.3.5 describes such a framework that provides empirical validation of the system safety and predictability.

The rest of this chapter presents our SafeMAT middleware that resolves these three challenges.

VI.3 Design of SafeMAT

This section presents our SafeMAT solution to provide adaptive and dynamic fault tolerance to DRE systems. Since SafeMAT is designed to extend an existing emulation environment for ARINC-653, we first briefly describe the underlying system and the existing fault management approach. Subsequently, we describe our SafeMAT solution.

VI.3.1 The ARINC-653 Component Model Middleware

The emulation middleware we use in our research is called the ARINC-653 Component Model (ACM) middleware, which essentially implements the CORBA Component Model [95] abstraction over the ARINC-653 emulation environment. ACM components interact with each other via well-defined patterns, facilitated by ports: asynchronous connections (event publishers & consumers) and/or synchronous provided/required interfaces (facets/receptacles). ACM allows the developers to group a number of ARINC-653 processes into a reusable component. Since this framework is geared towards hard real-time systems, it is required that each port be statically allocated to an ARINC-653 process whereas every method of a facet interface be allocated to a separate process.

ACM provides a design-time graphical modeling environment to enable a developer

to assemble the components of the application, deploy them into ARINC-653 partitions (essentially OS processes) of ARINC-653 modules (essentially the processors), and configure various real-time properties of the components. A runtime middleware honors these decisions. The ACM middleware comprises multiple different functionalities. Of interest to us in this research is the *Module Manager (MM),* which is a controller responsible for providing *temporal partitioning* among partitions.[2] For this purpose, each module is bound to a single core of the host processor. Using offline analysis, the MM is configured with a fixed cyclic schedule computed from the specified partition periods and durations. It is specified as offsets from the start of the hyper period, duration and the partition to run in that window. Once configured and validated, the MM implements the schedule using the `SCHED_FIFO` policy of the Linux kernel and manages the execution and preemption of the partitions. The MM is also responsible for transferring the inter-partition messages across the configured channels. In case of a distributed system, there can be multiple MMs each bound to a processor core that are controlled hierarchically by a system-level module manager.

VI.3.1.1 Software Health Management in ACM

We have extended and augmented the ACM software health management framework [35] with resource-aware adaptive fault tolerance (AFT). ACM supports the notion of Software Health Management (SHM), which provides incremental fault mitigation strategies and operates at two levels. The first and basic level of protection is provided by component-level health management (CLHM), which is implemented in all components. It provides a localized timed state machine with state transitions triggered either by a local anomaly or by timeouts, and actions that perform the local mitigation. The second and global level is called system-level health management (SLHM). The SLHM comprises an aggregator of alarms that are received from individual CLHMs. The Aggregator feeds these alarms to a

[2] Partitions are mapped to Linux processes.

diagnostics engine, which is configured with a failure propagation graph to reason about the root cause of failures. The decisions are then fed to a fault mitigation capability called a *Deliberative Reasoner* [36].

VI.3.1.2 Task Model

We employ a hierarchial fixed-priority preemptive task model of N partition tasks (denoted as $G = \{T_1, T_2, ..., T_N\}$) using Linux `SCHED_FIFO` scheduling class and are allocated within a module deployed on a predesignated CPU among a cluster of hardware nodes. All partition tasks can have periodic and sporadic subtasks (ARINC-653 processes) that constitute the application components which have hard real-time requirements as well as soft real-time requirements. Each component subtask is also scheduled on a FIFO basis. Each partition task T_i inside a module is configured with an associated period (denoted as P_i) that identifies the rate of execution. Upon a *hard deadline violation*, the faulty process is prevented from further execution by the partition scheduler by default. It is possible to change this action to allow a restart. *Soft deadline violations* results in a warning issued by the middleware and logs the warning by default. We assume that the networks within this class of DRE systems provide bounded communication latencies for application communication and do not fail or partition.

VI.3.1.3 Fault Model

An ACM component can be in one of the following three states: `active` (where all ports are operational), `inactive` (where none of the ports are operational) and `semi-active` (where only the consumer and receptacle ports are operational, while the publisher and facet ports are disabled). We focus on fail-stop failures within hard DRE systems that prevent clients from accessing the services provided by hosted applications. Failures can be masked by recovering and failing over to redundant backup replica components. Due to hard real-time constraints and to avoid state synchronization overhead, we use *semi-active*

replication [30] to recover from fail-stop processor failures. In semi-active replication, one replica—called the primary—handles all client requests in active state. Backup replicas are in semi-active state where they process client's requests but do not produce any output.

We consider two main sources of failure for each component port (a) logical failure - internal software, concurrency (deadline violations due to lock timeouts) and environmental faults, and latent error in the developer code to implement the operation associated with the port or (b) a critical failure, such as process/processor failures, or undetected component failures. By convention, to recover from logical failures, we failover to similar backup replicas with identical interfaces but alternate implementations (from different vendors/developers). In case of critical failures, we failover to identical backup replicas or to alternate backup replicas if available. Also by convention, alternate backup replicas can be deployed within the same partition whereas identical backup replicas are always deployed to different partitions in the same module or different modules of ACM.

VI.3.2 SafeMAT Architecture

We have designed the **Safe** **M**iddleware **A**daptation for Real-**T**ime Fault **T**olerance (SafeMAT) middleware to safely provision adaptive failure mitigation and recovery mechanisms in DRE systems that is resource-aware and leverages the benefits of software health management. The design of SafeMAT is driven by a holistic approach to answering the following three questions that emerge in fault tolerance for DRE systems:

VI.3.2.1 How to be resource-aware?

To answer this question requires fine-grained information on the resource utilization in the system, which can then be used in the adaptive decisions to deal with faults. The Distributed Resource Monitoring (DRM) framework in SafeMAT described in Section VI.3.3 provides this capability.

VI.3.2.2 How to deal with failures in the system of systems context by being aware of resources?

To answer this question requires a dynamic fault tolerance capability that can be adaptive to account for resource availabilities. The Adaptive Failure Management (AFM) framework in SafeMAT described in Section VI.3.4 provides this capability.

VI.3.2.3 How to ensure that the solutions do not compromise the safety and timeliness of existing real-time systems?

To answer this question requires a capability to validate that the dynamic and adaptive fault tolerance mechanisms will not compromise on the safety and timeliness of the already deployed systems. The Performance Metrics Evaluation (PME) framework in SafeMAT described in Section VI.3.5 provides this capability.

Figure 22 illustrates the architectural components of SafeMAT and their interactions. It depicts the underlying ARINC-653 Component Middleware solution upon which SafeMAT is designed and implemented.

SafeMAT has been architected in the form a hierarchy of cooperating components implemented atop ACM. At the topmost level resides the System Module Manager along with the SLHM that hosts the System Resource Monitor (sRM) and the Resource-Aware Deliberative Reasoner (RADaR), respectively. At the second level are the different Module Managers that are deployed on each computing processor core or machine, each hosting a Module Resource Monitor (mRM). At the third level are the different Partition Managers responsible for managing each partition, each hosting a Partition Resource Monitor (pRM). Each of the managers consequently have Failure Handlers to detect the failures in the partitions or modules and notifying them to the RADaR. The logical failures in components are notified by the respective CLHMs (from the ACM framework) residing in each application component. The different monitors form the DRM framework whereas the RADaR along with the various Failure Handlers form the AFM framework in SafeMAT. SafeMAT

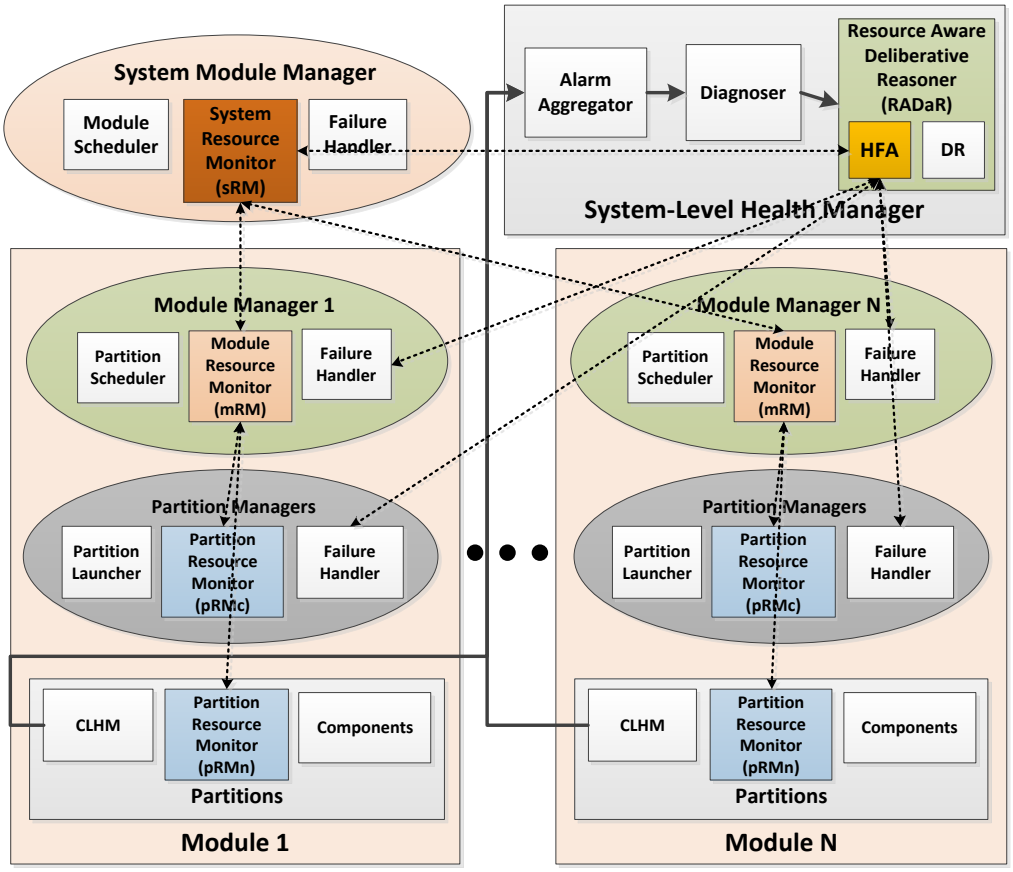


Figure 22: SafeMAT Architecture

extends ACM by providing an additional level of lower-level fault mitigation in the form of a partition manager and its resource monitor (pRM). Doing so helps to isolate failures in partitions and mitigate partition faults by taking actions right away instead of involving the module manager.

VI.3.2.4 Isolating the Impact of Failed Partitions

SafeMAT extends ACM by providing an additional level of lower-level fault mitigation in the form of a partition manager and its resource monitor (pRM). As with any multiprocess system, processes can fail due to external factors such as driver faults, buffer overruns,

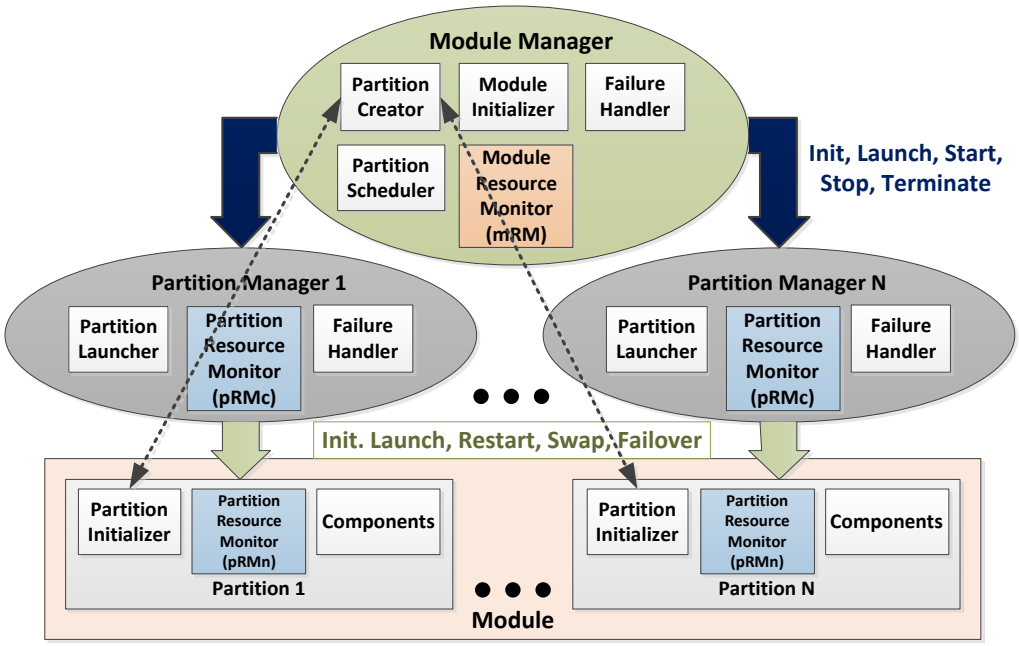


Figure 23: Partition Manager

segmentation faults, etc. It is necessary to enhance the safety of the real-time application by preventing failed partition processes from affecting the real-time schedule. The *Module Manager* handles the scheduling and execution of the partitions so whenever a partition process fails, it needs to ensure a quick recovery of that partition in a way that will not affect the real-time application schedule. However, in order to achieve this, the *Module Manager* needs to stop its scheduler and focus on restarting and initializing the partition. So in order to detect and isolate the effect of the partition failure and relieve the *Module Manager* from handling the partition recovery, we have developed a *Partition Manager* as shown in the Figure 23. The *Partition Manager* is instantiated for each partition and is responsible for handling the execution and failure management of each individual partitions. *Partition Manager* coordinates with the RADaR described in Section VI.3.4 for

managing the partition failures and their recovery. Whenever, it detects a partition failure, it restarts the partitions if instructed by the RADaR. Moreover, it also hosts the pRM to enable computation the resource utilization of the partition process and its constituent component threads. It also ensures that if the partition is restarted then it does not need to re-perform the synchronization with the *Module Manager* in order to save time and be ready and initialized before its next scheduling quantum arrives.

VI.3.3 Distributed Resource Monitoring

The Distributed Resource Monitoring (DRM) framework resolves Challenge 1 of Section VI.2 by providing a highly configurable and flexible distributed, hierarchical framework for monitoring the health and utilizations of system resources at various granularities, such as processor, process, component and thread. The framework comprises a distributed hierarchical network of a single System Resource Monitor (sRM) controlling multiple distributed Module Resource Monitors (mRM) that in turn control multiple Partition Resource Monitors (pRM) local to them in client-server configurations. The sRM resides in the system module whereas the mRMs are always deployed within the Module Managers and the pRMs are deployed within the individual partitions and their Partition Managers. The pRMs are of two types depending on their configured modes (a) *pMRc* in the COMPUTE mode and (b) *pMRn* in the NOTIFY mode.

VI.3.3.1 Configurability in DRM

It is possible to configure the DRM framework using different strategies, depending on the overall system configuration and amount of system resources available. These strategies include `reactive` and `periodic` monitoring strategies that can be used in conjunction with different granularities of monitoring system resources ranging from processes to threads. The reactive monitoring strategy is the least resource consuming since the CPU utilizations are computed only when instructed by the RADaR (in case of a failure). The

periodic monitoring strategy is the most resource consuming since the monitors compute utilizations periodically and keep the historic record of the utilizations to provide a better prediction regarding the utility of the resources. In the periodic strategy, the mRM periodically sends utilizations of all components to the sRM so that the information is readily available but may not be the most current one. The periodic strategy is also useful for profiling the resource utilizations during the profiling and tuning of the system execution characteristics in Section VI.3.5. Finally, it is also possible to configure the DRM framework to supply only the utilizations of the specific entities that RADaR is interested in.

VI.3.3.2 Discovering Resource Allocations

The DRM framework is also capable of discovering the runtime deployment and allocations of components to specific partitions and modules at runtime thereby obviating the need to configure the framework manually thereby enabling fast monitoring. It infers the assignments of the different subtasks to their components as well as allocations of components to their partitions when the monitors initialize their state. The pRMn runs within the partition in the NOTIFY mode where it does not compute the resource utilizations but only sends the mappings of the deployed components and their subtasks. These mappings are collated by the mRM and sent to the sRM which maintains the global allocations of subtasks to components, deployment of components to partitions, and the assignments of partitions to their modules. This capability enables the application of the DRM framework more generally to other types of systems where the allocations and deployments can change at runtime. Once the component deployment and allocations are learned by the sRM, it updates them with the primary-backup information about the components, component groups, and modules.

VI.3.3.3 Resource Liveness Monitoring

The DRM framework has been additionally entrusted with monitoring the health of its own monitors by periodically making the monitors in the lower level send their health status to the upper level monitors. This monitoring capability is auxiliary to the existing signal handlers that also detect partition and partition manager failures thereby creating a more robust dual health monitoring capability. Thus, if the health status beacon is not received from the pRMn and pRMc by the Module Manager and Partition Manager then it is assumed that the Partition (process), and the Partition Manager (process) have crashed respectively. Similarly, it is assumed the module (processor/core) has crashed if the mRM has not reported its health status beacon. Every time a failure is detected by the parent entity, the failure status is sent to the RADaR. Thus, the major advantage of SafeMAT over ACM is that while the SHM framework in ACM can only detect logical component failures, the DRM framework in SafeMAT can detect critical module, partition and component failures.

VI.3.4 Resource-Aware Adaptive Failure Mitigation

To perform resource-aware failure adaptation and address Challenge 2 of Section VI.2, we have developed the Adaptive Failure Mitigation (AFM) engine that leverages the DRM framework and augments the ACM-SHM framework through different cooperating runtime mechanisms, such as hierarchical failover and safe failure isolation. The AFM is designed as a collection of different components including the Failure Handlers and RADaR that integrate the *Hierarchical Failure Adaptation (HFA)* algorithm we developed with the Deliberative Reasoner (DR) [36] of the SLHM. The Failure Handlers are responsible for detecting process and processor failures and the simultaneous logical and critical component failures that have occurred but not reported to the HFA. The Failure Handlers along with the DRM framework and the HFA algorithm work together to provide quick and efficient failure adaptation at runtime.

VI.3.4.1 Failover Strategies

The type of failover strategy employed by the runtime failure adaptation mechanism is highly dependent on the failure type (*i.e.*, logical or critical), the failure granularity (*e.g.*, component, subsystem, partition or module), and the primary-backup deployment topology. The primaries can constitute individual components or groups of components (also called subsystems) and also the modules themselves. The ACM modeling paradigm allows various deployment scenarios for the primary components and their backups as shown in the Figure 24. For instance, the application component primaries and their corresponding backups can be deployed within the same module or can be spread across multiple modules. Moreover, they can either be deployed within the same partition or different partitions depending whether they are identical instances or alternate implementations of the primary replica. If the backups are an identical replica then by convention they are never deployed within the same partition as they are meant to handle critical failures that usually result in the process or the processor crashing. However, backups with alternate component implementations can be deployed within the same partition as they are meant to handle latent errors in the component's implementation logic.

Due to the different primary-backup deployment possibilities, it is necessary to implement adaptive failover mechanisms that take into account the failure type, granularity and deployment topology that can enable the ability to failover and recover the application component(s) at the component, subsystem, process and processor levels. Moreover, to remain resource-aware, our algorithm chooses the best candidates at each level for failover by ranking the backups dynamically in increasing order of either their processor or partition or component utilizations for which we leverage the DRM framework.

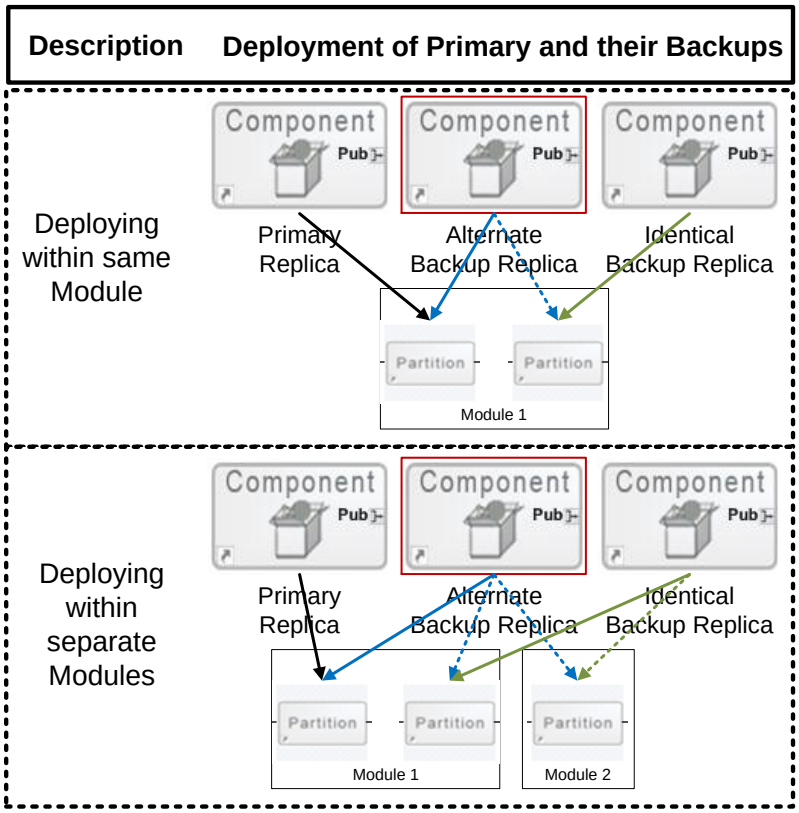


Figure 24: Backup Deployment Scenarios

VI.3.4.2 Enabling Hierarchical Failure Adaptation (HFA)

We have developed a Hierarchical Failure Adaptation (HFA) algorithm that adapts its failover targets depending upon the failure type, granularity and the primary-backup deployments. The algorithm is invoked whenever any of the DRM or the ACM-SHM frameworks detect a failure. In order to provide quick and efficient failover once the ACM Alarm Aggregator and the Failure Handlers detect a failed primary (component/partition/module), the sRM proactively pre-computes the sorted list of least utilized backups and the message is sent to the RADaR already containing the failed primaries piggybacked with the sorted list of failover target backups. The least utilized resource indicates maximum available slack. It then hands over the control to the SLHM.

It is the responsibility of the SLHM to determine as to when to activate the failure recovery mechanisms which is dependent upon the number of failures the system can withstand that have been programmed in advance within the ACM-SHM framework. It is also dependent upon the time taken by the system to stabilize till all alarms/errors are collected, which is usually a hyperperiod long in duration. Additionally, the AFM failure handlers and the DRM liveness monitoring is capable of detecting simultaneous module, partition, logical and critical component failures and are intelligently mitigated by the HFA algorithm in an hierarchical fashion.

VI.3.4.3 The HFA Algorithm

At the core of the HFA algorithm (Figure 25) are three functions: `DetermineFailover`, `DRWrapper`, and `Restart`. DetermineFailover is a function that determines how best to choose a failover target component and rewire it with the rest of the application. On a failure, HFA first detects the failure type (module/partition/component group/critical/logical). If it's a module failure (M_F), the algorithm fails over to the least utilized identical module and calls REWIRE on all the components in that module. If it's a partition failure (P_F), the algorithm invokes the DRWrapper function for each component deployed in that partition. Otherwise a component failure (L_F/(C_F)) is assumed and the DRWrapper function is called for that component. DRWrapper then calls the DeliberativeReasoner function to determine group failure (G_F) *i.e.,* if the component has any dependent components that will also require failover and it selects the least utilized backup target group of components and finally calls DetermineFailover on each component in the failed group.

In case of logical failure (L_F), DetermineFailover function checks if alternate backup replica is available. Otherwise, it checks for critical failure (C_F), and if true selects the least utilized identical backup replica if available. If not available, it checks if alternate backup replica is available. If not available, it restarts that partition to provide degraded QoS. If available, it checks for a simultaneous partition failure (P_F), in which case it selects

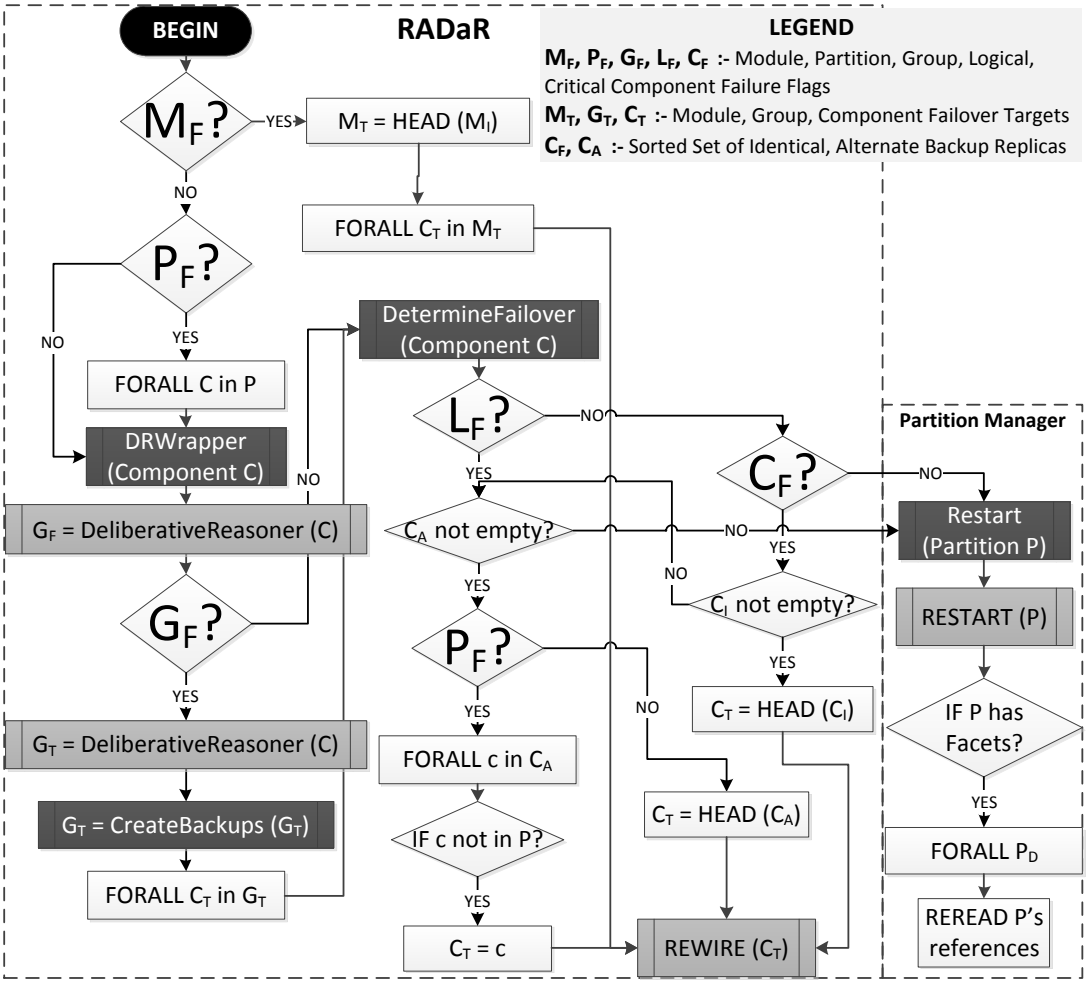


Figure 25: The HFA Algorithm

the least utilized identical replica in a different partition. If not a critical or logical failure, it restarts the partition. DetermineFailover handles the simultaneous partition failure as a special case where it has occurred simultaneous with a logical component failure. In case of a simultaneous critical component failure, it does not need to handle this special case as identical backup replicas are always deployed on a different partition as primary. If the restarted partition contained facets, the Restart function ensures that the dependent partitions reread the restarted partition's new component references.

Algorithm 6 The Hierarchical Failover Adaptation (HFA) Algorithm

Input:

M, P, G, C : Module, Partition, Group, Component Failed Primaries

M_F,P_F, G_F, C_F, L_F : Flags for Module, Partition, Group, Critical and Logical component Failures

M_I, G_I, C_I, C_A : Sorted List of Identical & Alternate Backup Replicas

Output:

M_T, G_T, C_T : Failover Target Backup Replicas

Begin *HFA*
if M_F **then**
  $M_T \leftarrow HEAD(M_I)$
  for all $C_T \in M_T$ **do**
    $REWIRE(C_T)$
  end for
else if P_F **then**
  for all $C \in P$ **do**
    $DRWrapper(C)$
  end for
else
  $DRWrapper(C)$
end if
End

Begin *DRWrapper* (Component C)
$G_F \leftarrow DeliberativeReasoner(C)$
if G_F **then**
  $G_T \leftarrow DeliberativeReasoner(C)$
  $G_I \leftarrow CreateBackups(G_T)$
  $G_T \leftarrow HEAD(G_I)$
  for all $C_T \in G_T$ **do**
    $DetermineFailover(C_T)$
  end for
else
  $DetermineFailover(C)$
end if
End

Begin *DetermineFailover* (Component C)
if L_F **then**
  CheckAlternate (C)
else if C_F **then**
  if $C_I \neq \emptyset$ **then**
    $C_T \leftarrow HEAD(C_I)$
  else
    CheckAlternate (C)
  end if
else
  $Restart(P)$
end if
$REWIRE(C_T)$
End

Begin *CheckAlternate* (Component C)
if $C_A \neq \emptyset$ **then**
  if P_F **then**
    for all $c \in C_A$ **do**
      if $c \ni P$ **then**
        $C_T \leftarrow c$
      end if
    end for
  else
    $C_T \leftarrow HEAD(C_A)$
  end if
else
  $Restart(P)$
end if
End

Begin *Restart* (Partition P)
$RESTART(P)$
if P has provided interfaces **then**
  for all $p \in P_d$ **do**
    $REREAD(P$'s references)
  end for
end if
End

VI.3.5 Pre-deployment Application Performance Evaluation

The real-time system execution schedule specifies the period of execution along with the allocated start and end times of the system tasks forming the scheduling quantum within the system execution time period (P). To address Challenge 3 of Section VI.2, we have developed an application Performance Metrics Evaluation (PME) framework that can profile the application execution times and CPU utilizations by leveraging the DRM framework to measure the actual utilizations of various component tasks within their allocated scheduling quantum in the system execution period. The profiling of a system's resource utilization during execution, both in the presence and absence of failures, helps in determining post-failover processor utilization of the application and SafeMAT components. We measure the approximate worst case execution times (WCETs) of the SafeMAT adaptation mechanism to estimate the additional runtime overhead incurred. This can also help in safely predicting whether the application is capable of recovering within the hard real-time deadline. Moreover, the fine grained performance evaluation of the application component subtasks can also provide the basis for the system integrator for determining the slack in the system and thereby alter the task allocations within the application execution schedules to enable provisioning the necessary runtime adaptation mechanisms and additional new/upgraded functionalities.

VI.3.6 SafeMAT Implementation

SafeMAT has been implemented atop the ACM hard real-time ARINC-653 emulation middleware. It is implemented in around 5000 lines of C/C++ source code excluding the ACM code. We describe the implementation details of the individual frameworks of SafeMAT.

VI.3.6.1 Partition Manager

We have implemented the *Partition Manager* as a separate process that gets spawned by the *Module Manager* for each partition that needs to be spawned. The *Module Manager* sends the necessary partition information through environment variables and command line parameters to the *Partition Manager* which in turn spawns the partition with the right parameters and the same environment variables set. In order for the partitions to correctly synchronize back with the *Module Manager*, the *Module Manager*'s PID is also set as one of the environment parameters along with the partition name, and the boolean indicating whether the partition is being restarted. The *Partition Manager* implements signal handlers as a means of handling partition failures. Whenever the failure handlers detect a partition failure, they check it's exit status after receiving a `SIGCHLD` and if it's an abnormal termination, the *Partition Manager* restarts the partition and also sets the boolean to true. If the boolean is set to true then the partitions do not need to re-perform the synchronization with the *Module Manager* and can quickly recover in time before their next scheduling quantum in the next hyperperiod. Furthermore, in order to handle partition restarts as described in the HFA algorithm in Section VI.3.4, if the facet side partition needs restarting the facet reference is reread for the receptacle side partition. This can be achieved through catching the invalid object reference exception and/or by sending a message to the partitions.

VI.3.6.2 Distributed Resource Monitoring (DRM) Framework

We have developed the DRM using the client server paradigm that can be configured with two different communication strategies - `reactive` and `periodic`. The communication between the mRM and the pRMs is established through plain UDP sockets for performance. We didn't employ TCP sockets as we assume the closed network that the avionics systems operate on have high reliability and high bandwidth performance with a small bounded network propagation delay. The sRMs are in charge of configuring the mRMs and pRMs with the communication strategies so that the clients need to worry

about correctly configuring all the monitors and thereby alleviating the need for configuration checking before deployment. This is achieved by making the mRM always initiate the first communication to setup and configure the monitors with the right strategy, the CPU number on which the module is deployed, the name of the partition to be monitored. The port at which they are expected to receive the messages is set through the environment variables while spawning the Partition Managers which forwards this information to the partitions that configure the pRMs. Additionally each of the monitors of the DRM framework are also programmed to perform their health monitoring by periodically sending their health status beacons to the their immediate parents through `ALIVE` socket messages. This aids in the detection of the partition (process) and module (processor) failures.

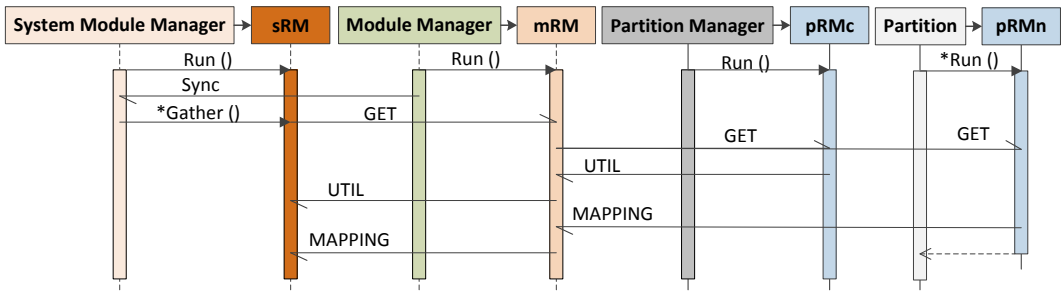


Figure 26: Distributed Resource Monitoring (DRM) Communication Sequence

The pRMc computes the processor, process and thread utilizations from the `/proc/stat`, the corresponding `/proc/<PID>/stat` and `/proc/<PID>/task/<TID>/stat` files on Linux. In the ACM emulated middleware we associate the module, partition and component utilizations with the respective processor, process and thread group utilizations. The partition and component utilizations are always computed as a fraction of the processor utilizations on which they are deployed on to reflect their true utility to the AFM engine. To

allow for efficient querying, the mRMs and the sRM maintain the mappings of the component allocations to their partitions and modules so that the AFM engine can selectively query the utilizations of specific components, partitions and modules. The PIDs are reported back by the pRMs to their corresponding mRM within the *Module Manager* when the partitions notify their initialization statuses to the *Module Manager* through the Linux message queues `/dev/mqueue/<Q-NAME>`. The pRMns within the partition communicate with the partition initialization logic and obtains the list of components assigned and deployed within that partition. The partition initializer also reports the corresponding ARINC-653 process (Linux Pthreads) identifiers (TIDs) that execute the different ports and methods within a component. This mapping of the components to their corresponding TIDs is reported back by the pRMns to their corresponding mRM. Once the mRM has the necessary partition PIDs and the component to TID mappings, it enables the sRM to report CPU utilizations on a per component or a per process or a per module basis whenever queried by the SLHM components. The sequence of communications that occur between the sRM, mRM, pRMc, and pRMn components is shown in the Figure 26.

VI.3.6.3 Adaptive Failure Mitigation (AFM) Engine

The Diagnoser and Deliberative Reasoner components from the SLHM framework have been extended by integrating the HFA algorithm and DRM frameworks. We have developed the Resource Aware Deliberative Reasoner (RADaR) by improving the reasoning algorithm employed by the Deliberative Reasoner (DR) within the SLHM framework to compute component failover targets by considering the CPU utilizations, failure type, failure granularity and the deployment topology. We have incorporated the failure detection of the partitions and modules through failure handlers and DRM health status monitoring. Additionally, in order to detect logical or component failures in case of simultaneous partition or module failures, the RADaR traverses the history of any failures that were caught by the Alarm Aggregator and the output files generated out by the CLHMs that indicate

the failure types. This gives us the capability to handle both logical component failures as well as critical process and processor failures simultaneously within the same framework. The HFA algorithm provides a wrapper over the DR's reasoning algorithm. We integrated the HFA algorithm in the decision making part of the deliberative reasoning algorithm that gets executed each time the DR gets invoked with the failed components. First it uses the DR's dependency tracking phase to figure out the dependent group of component's that require failover and the failover candidates initially generated by the DR. The DR achieves this through a search of the component's assembly specification and deduces the failed component's dependencies and determines whether the dependent components also need recovery. When the DR comes up with the initial failover target component or a group of components, they may not be necessarily the best candidates. We select the best failover target for the failed component by executing the HFA algorithm on the initial result of the DR and manipulate the DR's output with the better candidates provided by our algorithm. The HFA algorithm achieves this by querying the sRM for the sorted rank lists of failover target backup replica components based on their relative utilizations.

VI.3.6.4 Application Performance Metrics Evaluation (PME) Framework

We profile the SafeMAT component's actual WCETs and actual online CPU utilization percentages within each execution quantum of the hyperperiod by analyzing the timing logs generated by the Module Manager and the Partitions and the performance logs generated out by the DRM framework respectively over a large number of iterations. To achieve this we can configure the DRM to periodically collect the CPU utilizations only at the end of each hyperperiod. The analysis of the timing log files is performed by parsing the standard tags such as `START_*`, `STOP_*` corresponding to the start and stop times of the various processing blocks using Python scripting. We compare these to the actual measured CPU utilization between those times to the duration of the quantum to get an idea of the slack that is available within each quantum. We particularly profile the utilizations of the

Health Management and SafeMAT framework components to verify that the utilizations don't reach 100% so that they are able to finish their decision making within the allocated quantum . This ensures that the recovery from failures is made as fast as possible.

VI.4 Empirical Evaluation of SafeMAT

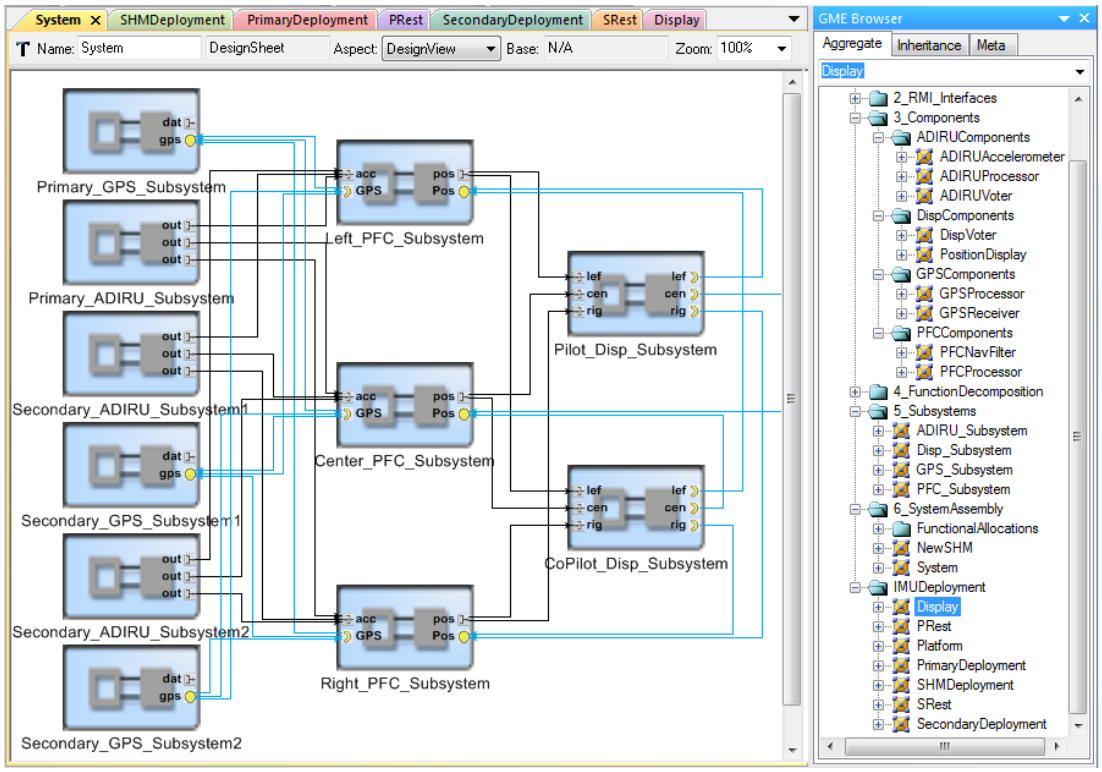


Figure 27: IMU System Assembly.

To measure the performance of the various SafeMAT adaptive mechanisms, we used a representative DRE system called the Inertial Measuring Unit (IMU) [37] from the avionics domain. IMU is rich and large enough to provide a large number of components and redundancy possibilities that stem from the composition of its subsystems comprising the Global Positioning System (GPS), the Air Data Inertial Reference Unit (ADIRU) [76], the flight control (PFC) subsystem, and the Display subsystem. Figure 27 shows the IMU system assembly comprising primary subsystems of GPS and ADIRU, and their two secondary

backup replica subsystems connected to redundant PFC and Display subsystems. When the GPS processor has an updated position, it sends a pulse out of its publisher port and the subscriber GPS Receiver can asynchronously detect it and fetch the data coordinates. The ADIRU subsystem comprises actively replicated 6 Accelerometers, 4 Processors, and 3 Voters and is designed to withstand 2 Accelerometer failures. The 6 Accelerometers feed acceleration values to each of the four Processors which compute the body acceleration data and fed it to each of the three Voters. In turn the Voters choose the middle value and output it to the PFC subsystem. The GPS Processor and The ADIRU Voter feed the 3D location coordinates and acceleration values respectively to each of the PFC subsystem that integrates the acceleration values over the 3D coordinates and computes the next coordinate position and outputs it to the Display subsystem which further votes and chooses one of the three coordinate values received. The Secondary GPS and ADIRU subsystems are semi-actively replicated. The GPS subsystems and ADIRU subsystems run at a frequency of 0.1 Hz and 1 Hz respectively. The PFC fetches the GPS data a slower but accurate rate of 0.1 Hz whereas the Display subsystem fetches the data from the PFC subsystem at a rate of 1 Hz. Thus, the hyperperiod of the IMU is 10 seconds (LCM of 1 and 10). Deployment information of all the subsystems is not shown in this paper for similar reasons. However, we discuss the impact of various primary-backup deployments on the overall runtime adaptation overhead added by SafeMAT by going into the deployment details of the standalone adaptation of the GPS Subsystem in Section VI.4.2. [3]

[4] Deployment information of all the subsystems is not shown in this paper for similar reasons. However, we discuss the impact of various primary-backup deployments on the overall runtime adaptation overhead added by SafeMAT by going into the deployment details of the standalone adaptation of the GPS Subsystem in Section VI.4.2

[3]The details of each subsystem and their deployments have been omitted in this paper for the lack of space, which can be found in [37].

[4]The details of each subsystem and their deployments have been omitted in this paper for the lack of space, which can be found in [37].

VI.4.1 Evaluating SafeMAT's Utilization Overhead

We use SafeMAT's PME framework to determine the overhead imposed by the SafeMAT's fast failure adaptation capability by measuring the CPU utilizations of its components. Measuring the actual utilizations at the end of each execution hyperperiod is an indicator of the slack available for accommodating failure adaptation mechanisms. Since SafeMAT builds over ACM, we executed 100 iterations of the IMU system each for the plain vanilla ACM-SHM and the SafeMAT adaptation failure recovery mechanisms. We artificially introduced failures at 15, 20, 30, 35 iterations in the GPS Processor, Accelerometers 6, 5 and 4, respectively such that the values output by them are exceedingly high (*i.e.* deviate from the expected trend). Once Accelerometer 4 fails at iteration 35, the system begins to malfunction and the Display starts receiving erroneously high acceleration values.

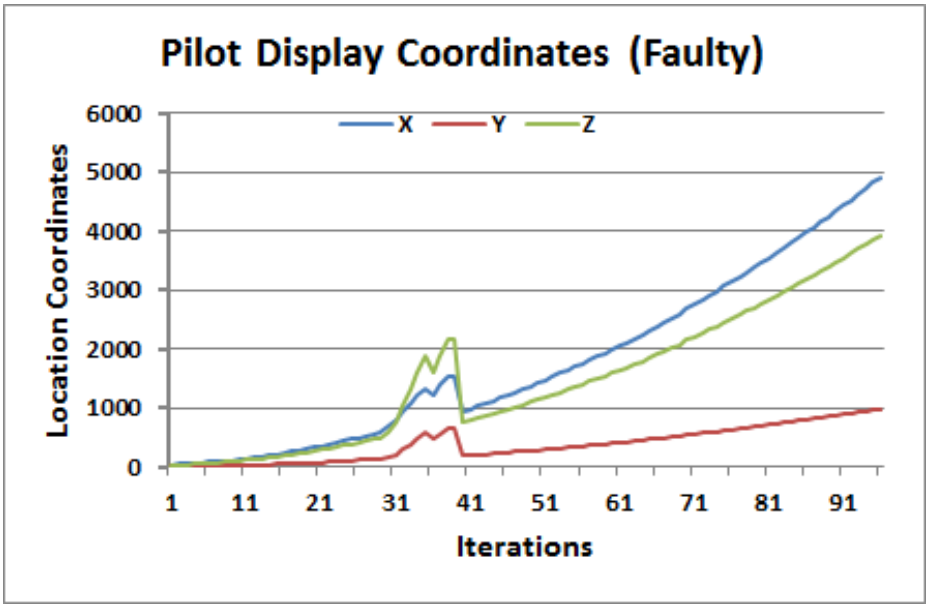


Figure 28: Application Recovery after Failover

Figure 28 shows the 3D-position (X, Y, Z coordinates) values received by the `Pilot_Display_Sub` getting out of sync between iterations 30 and 40, and recovering after the SafeMAT failure

adaptation takes place. The perturbation is caused by the erroneous acceleration values because the IMU is solely capable of operating using just the acceleration values without the need for continuous GPS input. GPS coordinates are used to just supply the initial coordinates for the integration over the acceleration values computed by the PFC subsystem. At this moment the SafeMAT failure adaptation starts executing and makes the ADIRU and GPS primary subsystems failover to one of their semi-active secondary subsystems depending upon their overall least average utilizations. In this execution scenario the `Primary_ADIRU_Subsystem` fails over to the `Secondary_ADIRU_Subsystem2` whereas the `Primary_GPS_Subsystem` fails over to the `Secondary_GPS_Subsystem1`. Figure 29 shows that the SafeMAT does not add significant utilization overhead (2-6%) over the existing ACM-SHM imposed utilizations (26-73.26%).

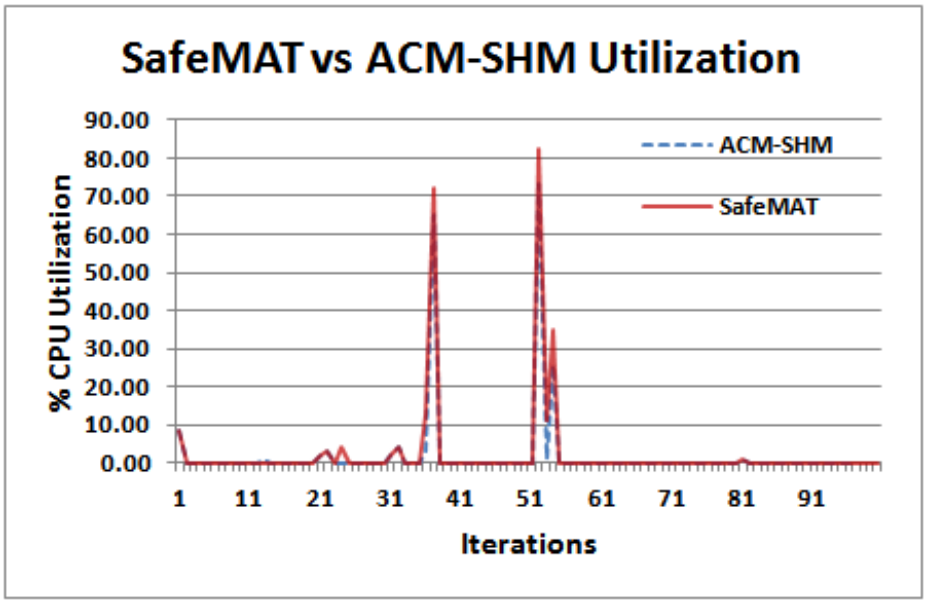


Figure 29: SafeMAT Utilization Overhead

VI.4.2 Evaluating SafeMAT-induced Failover Overhead Times

To qualitatively measure SafeMAT's runtime failover overhead times we measure the worst-case execution times (WCETs) of the SafeMAT's components based on two main parameters: (1) the impact of component replica placements relative to their primaries and (2) the number of nested components within the component group that need failover. We measure the failover overhead (T_{FO}) as:

$$T_{FO} = T_{Diag} + T_{DR} + \sum_{i=1}^{m} \left(T_{mRM} + \sum_{j=1}^{p} T_{pRM} \right) + T_{sRM} + T_{HFA}$$

where

m - number of modules

p - number of partitions within each module

T_{Diag} - WCET for Failure Diagnosis

T_{DR} - WCET for Deliberative Reasoning

T_{sRM} - WCET for the sRM to collect utilizations

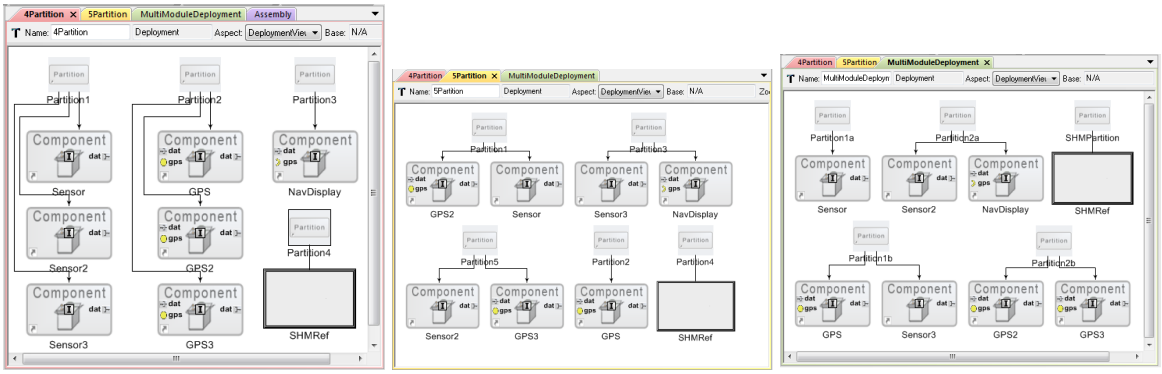
T_{mRM} - WCET for each mRM to collect utilizations

T_{pRM} - WCET for each pRM to collect utilizations

T_{HFA} - WCET for Hierarchical Failover Algorithm

VI.4.2.1 Impact of Component Replica Deployments

To measure the impact of component replica deployments, we focused on the GPS subsystem from the IMU case study. Figure 20 shows the assembly for the BasicSP system with a redundant set of Sensor and GPS components (Sensor2 Sensor3, GPS2, GPS3). Sensors publish an event every 2 sec for their associated GPS. The GPS consumes the event published by its sensor at a periodic rate of 2 sec. Afterwards, it publishes an event, which is sporadically consumed by the Navigation Display. Thereafter, the NavDisplay component updates its location by using getGPSData facet of the GPS Component. In the



(a) Same Partition as Primary (b) Different Partition as Primary (c) Different Module as Primary

Figure 30: Different Component Replica Deployments

initial setup of the assembly, the Sensor, GPS, and NavDisplay components are used and hence set to be in *active mode*. The redundant Sensor and GPS (Sensor2 Sensor3, GPS2, GPS3) are not used. The GPS2 & GPS3 is set to a *semi-active mode*, leaving the Sensor2 & Sensor3 components in active mode. This would allow the GPS2 & GPS3 to keep track of the current state (by being in semi-active mode where the GPS2's and GPS3's consumers are active) but not affecting NavDisplay.

We created different deployment scenarios by altering the placements of the component replica by either placing them either within the same partition as primary(Figure 30a), or a different partition in the same module (Figure 30b)or a different partition within a different module(Figure 30c). We executed the GPS subsystem with the existing vanilla ACM-SHM recovery mechanisms in place and with the new SafeMAT failure adaptations enabled. We have considered the WCETs of both ACM-SHM and SafeMAT in this case. As shown in Table 31, SafeMAT incurs comparable execution times to the existing ACM-SHM execution times as this scenario has been evaluated on a per component basis. The times go up as the replica partitions move further away from the primaries. The high recovery overhead per component are due mainly to the unavoidable network latency to collect the utilizations. However, the minuscule overhead on the order of a few milliseconds are very insignificant in this case and will not cause deadline violations when there is a large amount of slack available, which is usually the case. Therefore, this is not a cause

of concern as shown in the next evaluation where we progressively increase the number of components that need failover – a scenario that is more common in real systems.

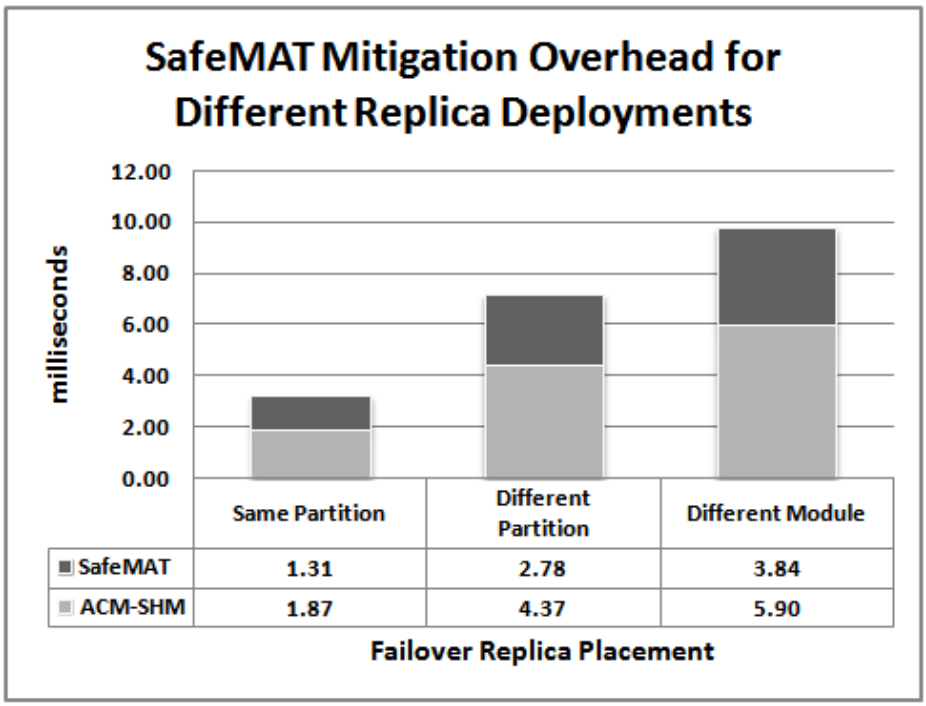


Figure 31: SafeMAT Mitigation Overhead for Different Replica Deployments

VI.4.2.2 Impact of Component Group Size

To measure the impact of size of the group of components that require failover, we measure the overhead incurred by SafeMAT for the GPS and ADIRU subsystems where the number of components increase from just 2 to 13. As shown in the evaluation Table 32, when the number of components increase, the SafeMAT overhead costs gets amortized over larger number of components. The effective additional runtime overhead incurred by SafeMAT's adaptive mechanisms becomes significantly less (9-15%) compared to the ACM-SHM's diagnostic and reasoning overhead. SafeMAT's overhead is largely dependent on the size of the recovery group, deployment complexity of the components within the recovery group, and the amount of network communication required within the DRM

as shown in the T_{FO} equation. However, it does not grow exponentially, as recovery group size increases. The more the number of components that need failover, the more the amount of utilization data that can be bundled together in the network messages that are sent by the DRM monitors to RADaR. Conversely, the smaller the number of components affected, the greater the overhead incurred by SafeMAT due to the network communication that is mandatory even for relatively small number of messages exchanged.

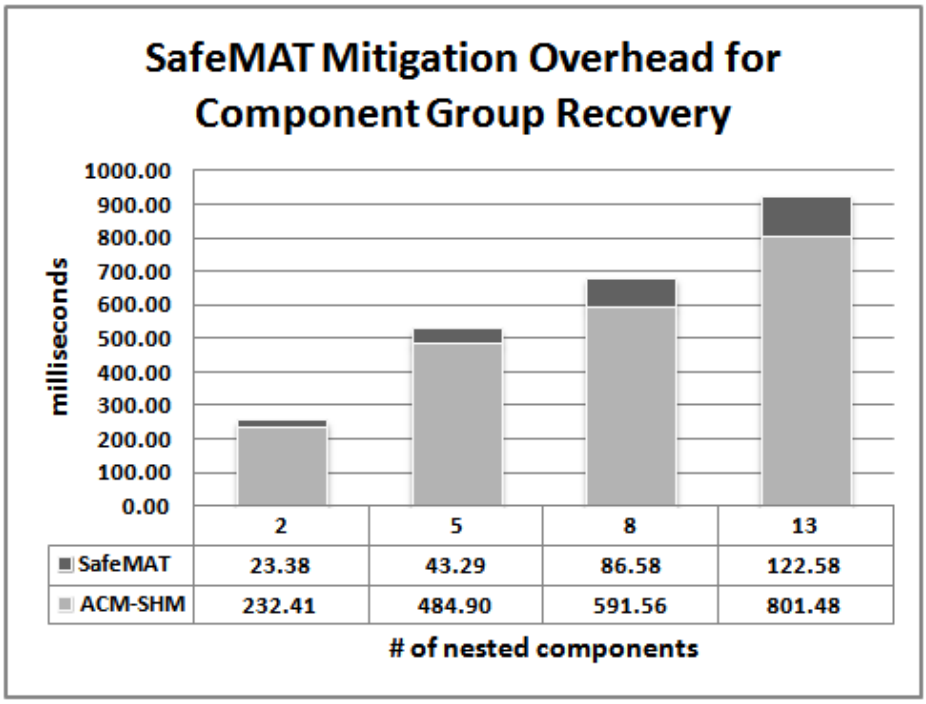


Figure 32: SafeMAT Mitigation Overhead for Component Group Recovery

VI.4.3 Discussion: System Safety and Predictability

Compared to the vanilla ACM-SHM mechanisms, SafeMAT adds negligible runtime utilization overhead without overloading the system while performing better failure recovery within the available utilization slack. Moreover, by selecting the least-utilized failover targets, SafeMAT maintains more available post recovery slack within the system compared to ACM-SHM, while potentially improving the task response times as well. Figure 33

shows that there was no noticeable impact on the Display jitter values using SafeMAT over vanilla ACM-SHM and therefore the response times remained largely unaffected while at the same time failure recovery was superior. Moreover, there were no missed real-time deadlines for the application tasks. Moreover, SafeMAT adds negligible runtime failover overhead thereby maintaining the predictability of the overall system Thus, these results illustrate that SafeMAT maintains the safety of the system and also the predictability.

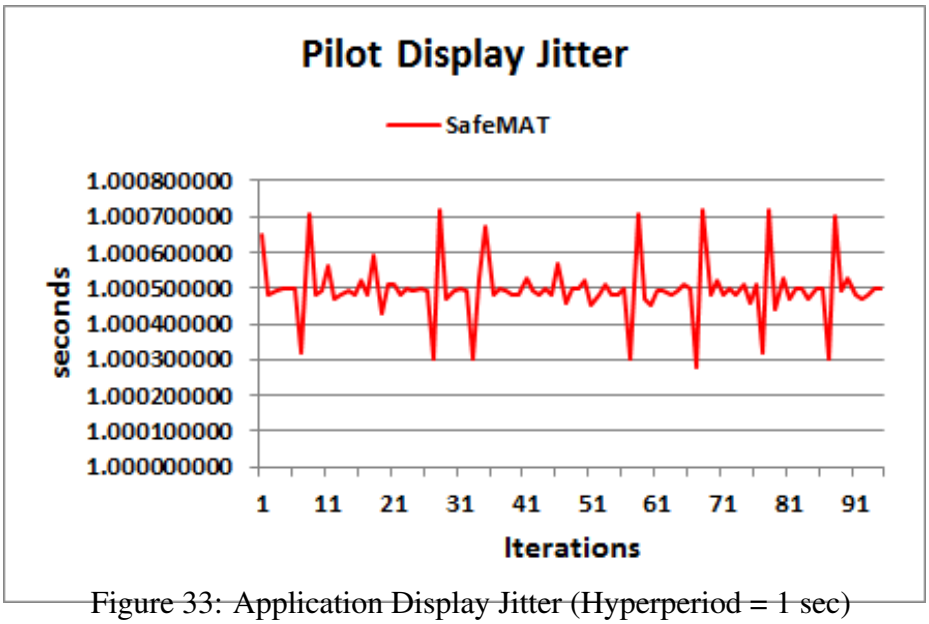


Figure 33: Application Display Jitter (Hyperperiod = 1 sec)

VI.5 Conclusion

Mission-critical hard real-time applications being in-service for many years, have too rigid execution schedules to incorporate additional evolving domain requirements in the form of new functionalities and better failure adaptation techniques even if their resources are over-provisioned to ensure their safety and predictability. While, existing SHM techniques are predictable, they are too static and do not offer the best case failure adaptation in real-time. In order to evolve these systems and improve their predictability, reliability

and resource utilizations, it is necessary to discover the existing slack within their execution schedules and utilize it to safely provision additional and efficient dynamic failure adaptation mechanisms.

In this chapter, we presented a dynamic, safe middleware adaption technique and a performance metric evaluation framework that provided a fast and adaptive failover through flexible and configurable fine-grained resource monitoring and an hierarchical failure adaptation algorithm that is not only resource-aware but also took into account the failure type, failure granularity, the relative component replica placements. Our approach manifested in the form of the SafeMAT middleware and the PME framework. We also rigorously evaluated our adaptive middleware by measuring the runtime utilization and the execution overhead for different replica deployments as well as an increasing number of components.

CHAPTER VII

FUTURE WORK – DEPLOYMENT AND COMPOSITION OF SPECIALIZED MIDDLEWARE

The past chapters focused on developing a taxonomy for categorizing and reasoning about middleware specializations and realized a feature-oriented, automated and generative process for inferring middleware features, deducing application invariants, and ultimately synthesizing the middleware specializations, respectively. Although the presented techniques significantly reduce the developer efforts involved in driving and synthesizing middleware specializations, they do not adequately address the runtime issues that arise when multiple specialized middleware *forms* are synthesized by the middleware specialization process applied to satisfy the requirements of DRE systems.

VII.1 Side-effects of Specializations on System Composition and Deployment

The generation of specialized *forms* of general-purpose middleware will impact the way the middleware is utilized by the application components at runtime. This directly impacts the overall runtime system composition and the deployment of its applications. In order to utilize the features of the specialized middleware *forms* correctly and seamlessly over the entire application operational string, the specialized *forms* need to be composed and configured correctly within the existing middleware runtime infrastructures.

1. ***Composition* of the Specialized Middleware Forms within the Middleware Architecture -** Specialization of general-purpose middleware generates multiple middleware *forms* that specifically cater to the feature requirements of the components. Hence, this gives rise to a new problem where the specialized middleware components need to be composed with the application server fabric. Even if the interfaces that compose the

application server and the specialized middleware components are not modified, the reduced features provided by the middleware will finally impact the QoS provided by the container. Therefore, it is necessary to ensure the features and QoS provided by the specialized middleware is utilized consistently throughout the application server. Moreover, as multiple components are composed together it is necessary to ensure that the corresponding application servers on which the components are hosted work seamlessly and correctly. Not only the components need to work correctly between neighboring components but over the entire operational string that consists of a chain of application components.

2. ***Deployment* of the Specialized Middleware Forms -** As the application servers are composed/specialized, the corresponding deployment infrastructure needs to be able to pre-install them onto the target nodes and prepared to be ready to host the application components. Moreover, the deployment infrastructure needs to be oblivious to the fact that there are multiple specialized middleware servers being installed. The D&C infrastructure should deploy the application servers the same way as application components. Additionally, the components and the application servers need to configured according to the specialized functionality supported/required by them.

Let us understand the genesis of this runtime problem first. Component-based DRE applications are often installed across different target machines utilizing a deployment and configuration (D&C) infrastructure. Each target node (machine) has the necessary infrastructure machinery pre-installed to be able to host the application components. This runtime infrastructure is known as the middleware *Application Server* (AS), which hosts the application components as shown in Figure 34. The application server stack is comprised of multiple middleware framework layers. For example, in a CORBA Component Model-based application server, it comprises: (1) a communication substrate in the form of a Object Request Broker (ORB), (2) one or more component hosting entities known as containers, (3) a framework known as Portable Object Adapter (POA) that provides the

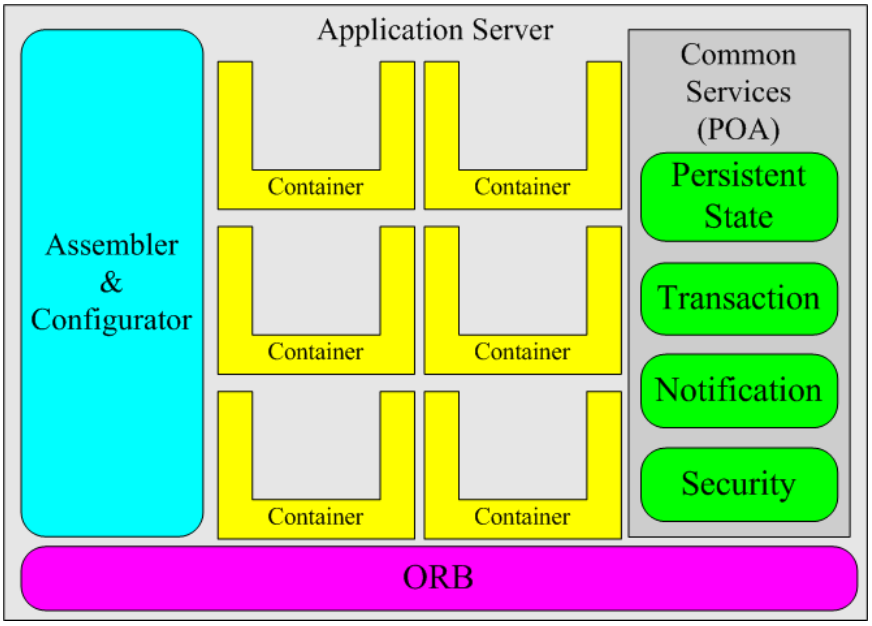


Figure 34: A Generic Application Server and its components

common services and, (4) a set of configuration handlers that provide hooks for configuring the application server entities.

Research to date on the middleware specialization process as shown in Chapter V has concentrated on specializing the individual application server framework layers thereby generating their multiple *forms,* which specifically cater to the feature requirements of the components hosted on that application server. However, individual frameworks alone do not provide a fully operational application server capability. In the traditional application server, these unspecialized frameworks seamlessly compose with each other. Hence, the next logical step is to how to compose individually specialized frameworks together to synthesize the application server stack? Assuming that these application servers are composed on a per-component basis, it yields a trivial component deployment scenario as shown in Figure 35. As shown in the figure, each of target nodes (1..N) have specialized application servers (1..N) deployed that host their respective components (1..N). Each specialized server is composed of individual specialized frameworks (ORB, POA, Container).

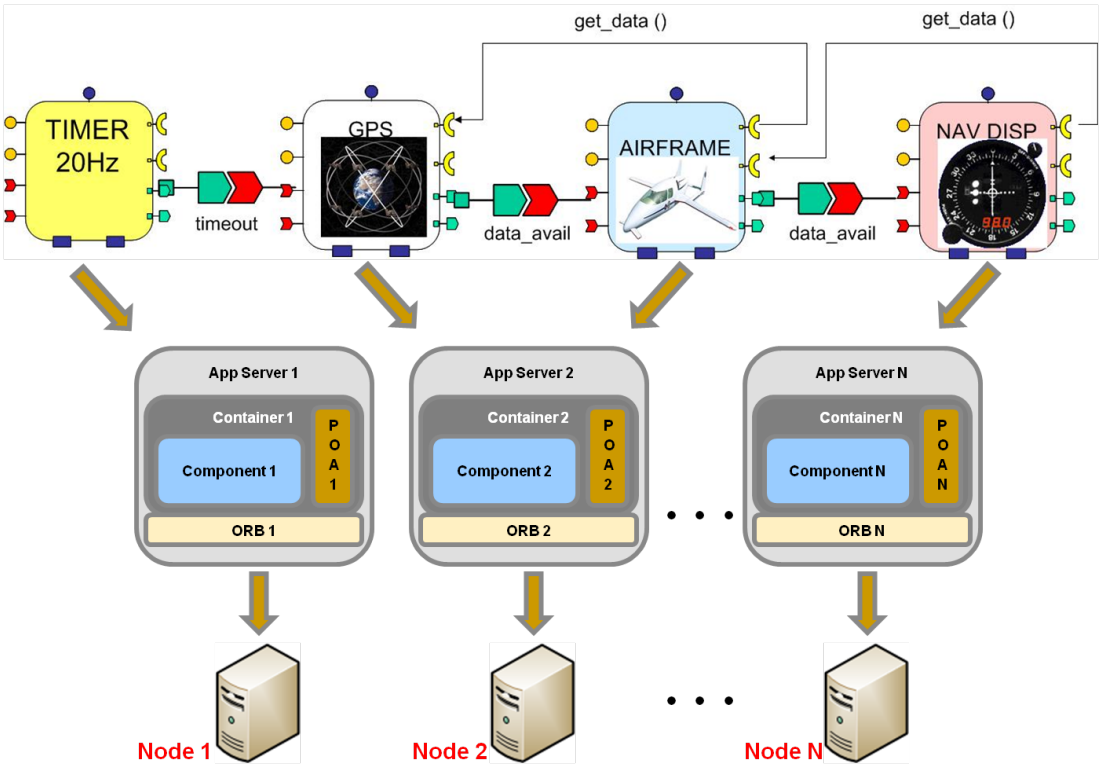


Figure 35: Deployment of Non-homogenous Specialized Application Servers

An important aspect of a component-based DRE system is that the D&C infrastructure should work coherently with the application server stack installed on each target node in order to be able to correctly provision the required resources and desired middleware features to the running DRE system components. The traditional D&C infrastructure assumes that the application server installed on each target node offers the same feature set i.e., they are homogenous. However, as shown in the figure 35, the application servers on each target node differ in the features provided. This is an undesirable side effect of middleware specializations, if it expected that the deployed system is to be able to operate correctly and predictably.

Thus middleware specialization of the application server stack directly impacts the overall runtime system composition and the deployment of its applications. In order to utilize the features of the specialized middleware frameworks correctly and seamlessly across the application server framework layers, the specialized frameworks need to be

composed and configured correctly within the existing application server runtime infrastructures. Hence, this gives rise to a new problem where the specialized middleware frameworks need to be safely composed within the application server fabric.

VII.2 Related Research

We survey the existing body of research on middleware composition in two categories. First, we look at works that involve developing compositional middleware designs that are highly modular and configurable. Second, we discuss the one-off middleware server customizations that were performed to satisfy a certain QoS requirement. Next we survey the past research on deployment optimizations.

VII.2.1 Flexible Middleware Composition

Middleware researchers have perennially tried to improve the modularity and flexibility of middleware designs. However, they are constantly faced with the design tension between generality and specificity while architecting middleware designs. In order to have generality they have used well known patterns and frameworks but to maintain performance have avoided using highly dynamic service composition techniques. Advantage of dynamic service composition is services are loaded into the memory only when needed thereby minimizing footprint but incur dynamic loading overhead.

ZEN [65] uses a flexible and extensible micro-ORB design (rather than monolithic-ORB design) for all CORBA services by generalizing TAO's pluggable protocol framework to other modular services within the ORB so that they need not be loaded until they are used. It identifies each core ORB service whose behavior may vary and moves it out of the ORB by applying the Virtual Component pattern [24] to make each service pluggable dynamically.

JBoss [41] is an extensible, reflective, and dynamically reconfigurable Java application server that is itself built in a component-based out of dynamically deployable components

that provide middleware services to application components. On such a server, extensible and dynamically reconfigurable, two general kinds of components can be deployed: deployed: middleware components and application components. Due to the differences between middleware components and application components, multiple component models are likely to coexist in a component-based application server: component-based application server: a model for middleware components, plus one or more models for application components.

PAM [9] provides a model-driven, deployment time optimization technique that tries to minimize the application footprint and invocation latency through a novel assembly fusion algorithm that composes the collocated application component's glue code (stubs and skeletons) into an single component assembly.

Modelware [142] advocates the use of models and views to separate intrinsic functionalities of middleware from extrinsic ones. Modelware considerably reduces coding efforts in supporting the functional evolution of middleware along different application domains. The authors use the term *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains. They use the term *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change.

FACET [51] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. To support functionality not found in the base code, FACET provides a set of features that can be enabled and combined subject to some dependency constraints. By using AOP techniques, the code for each of these features can be weaved at the appropriate place in the base code.

VII.2.2 QoS-specific Middleware Customizations

Wolf et al. [137] developed a custom component middleware server - CORFU which provides first class support for group failover and recovery based on component replication

along with support for real-time state dissemination among the group while providing automated deployment and configuration for group semantics. It addressing key challenges of component-based fault-tolerance, including the need for efficient synchronization of internal component state, failure correlation across groups of components, and configuration of fault-tolerance properties at the component granularity level.

Balasubramanian et al. [7] developed SwapCIAO, which is a QoS-enabled component middleware framework that enables application developers to create multiple implementations of a component and update (i.e. Şswap Ť) them dynamically. SwapCIAO provides techniques for updating component implementations dynamically and transparently (i.e., without incurring system downtime) to optimize system behavior under diverse operating contexts and mode changes. SwapCIAO extends CIAO, which is an open-source implementation of the OMG Lightweight CCM [93], Deployment and Configuration (D&C) [97], and Real-time CORBA [94] specifications The key capabilities that SwapCIAO adds to CIAO include (1) mechanisms for updating component implementations dynamically without incurring system downtime and (2) mechanisms that transparently redirect clients of an existing component to the new updated component implementation.

Wang et al. [134] describes how CIAO is augmenting the standard CCM specification to support static QoS provisioning that pre-allocates resources for DRE application and how dynamic QoS provisioning and adaptation can be addressed using middleware capabilities called Qoskets, which are collections of reusable software modules of the Quality Objects (QuO). They particularly focus on realtime QoS. They integrate CIAO and Qoskets to enable composition of both static QoS provisioning and dynamic adaptive QoS assurance in DRE applications. In particular, they focus on how CIAO uses Qoskets to weave in the software elements to create an integrated QoS-enabled component model that offers a total QoS provisioning solution for DRE applications.

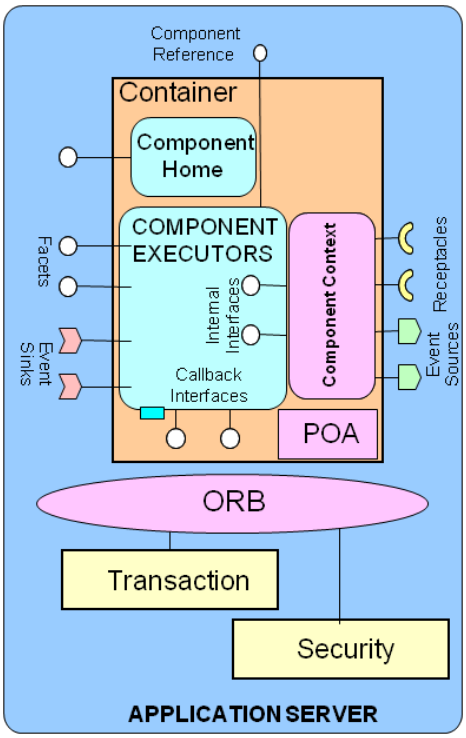


Figure 36: Application Server Stack

VII.3 Unresolved Challenges

We have identified three major challenges that impede the deployment and composition of specialized middleware frameworks layers within the application servers which arise mainly due to the large amount of general-purpose component middleware already being developed and not having the ability to support high pluggability of its own constituents.

VII.3.1 Challenge 1: Preserving Operational Correctness of Specialized Middleware Stack

The specialization of the individual framework layers of the application server stack results into inconsistencies in the features provided among the different layers. This creates a problem where it becomes difficult to compose the layers together to synthesize the application server stack as shown in the figure 36. It is important to ensure that the features are utilized consistently across the entire middleware stack. It is not only important to ensure

that the inter-layer interfaces are unchanged but their behaviors are also consistent. For example, an ORB specialized to support only `Select Reactor` cannot be composed with a POA configured with `Thread Pool` policy. Therefore, the framework composition techniques to ensure that the feature and configurations match across layers and are being consistently used to preserve the overall operational correctness of the specialized middleware stack.

VII.3.2 Challenge 2: Preserving Deployment Transparency during Middleware Composition

As the application servers are composed/specialized, the corresponding deployment infrastructure needs to be able to pre-install them onto the target nodes and prepared to be ready to host the application components. The design goals of a D&C infrastructure is to be transparent to the technology and feature composition of the application components being deployed. The application servers that host these components on the target nodes implement the standard hooks required to install, configure, run and uninstall the components. Therefore, application servers need to be composed from specialized middleware frameworks in a way that preserves these interfaces and their expected behavior in order to preserve the deployment transparency expected by the D&C infrastructures.

Preserving deployment transparency becomes even more paramount when the component allocation is controlled by an offline task allocation algorithm as shown in the figure 37. Depending upon the allocation decisions taken by the algorithm, the granularity of the composition may change from composing application server skeletons, to composing containers, and finally to composing component glue code [9]. The way the application server entities are composed should be transparent to the D&C infrastructure.

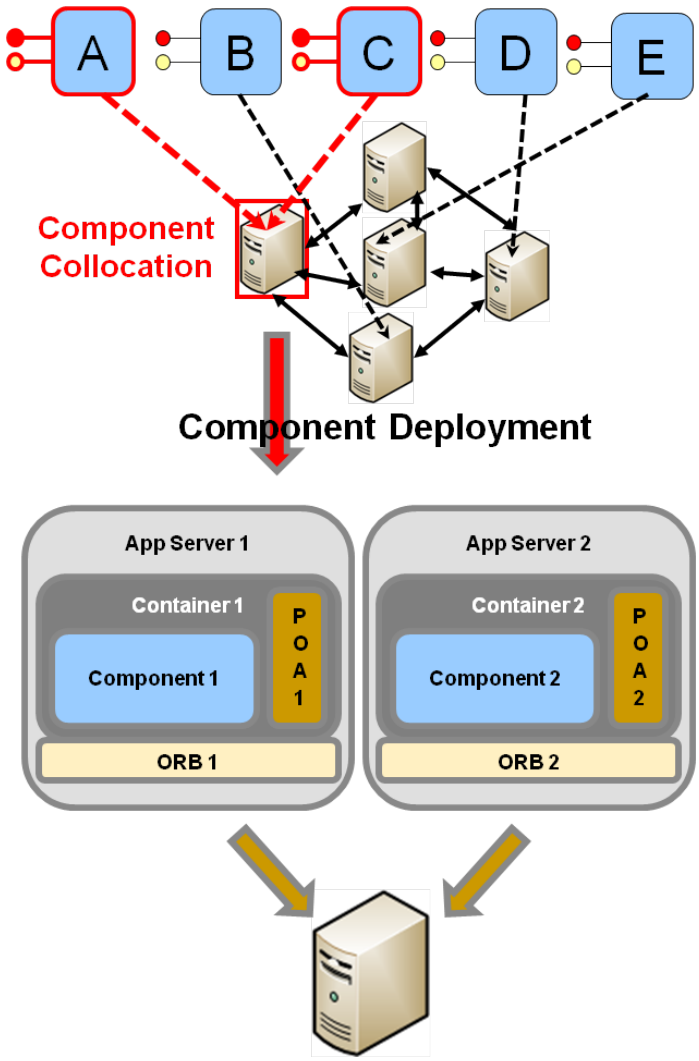


Figure 37: Component Allocation Example

VII.3.3 Challenge 3: Determining Middleware Composition Granularity

The figure 38 showcases the different application server composition granularity when any two components are collocated and hence deployed on the same target node. The challenge here is to determine what is the best composition granularity in order to keep footprint and invocation latencies at a minimum. Therefore, there is a need to devise optimization techniques that can help resolve these composition decisions.

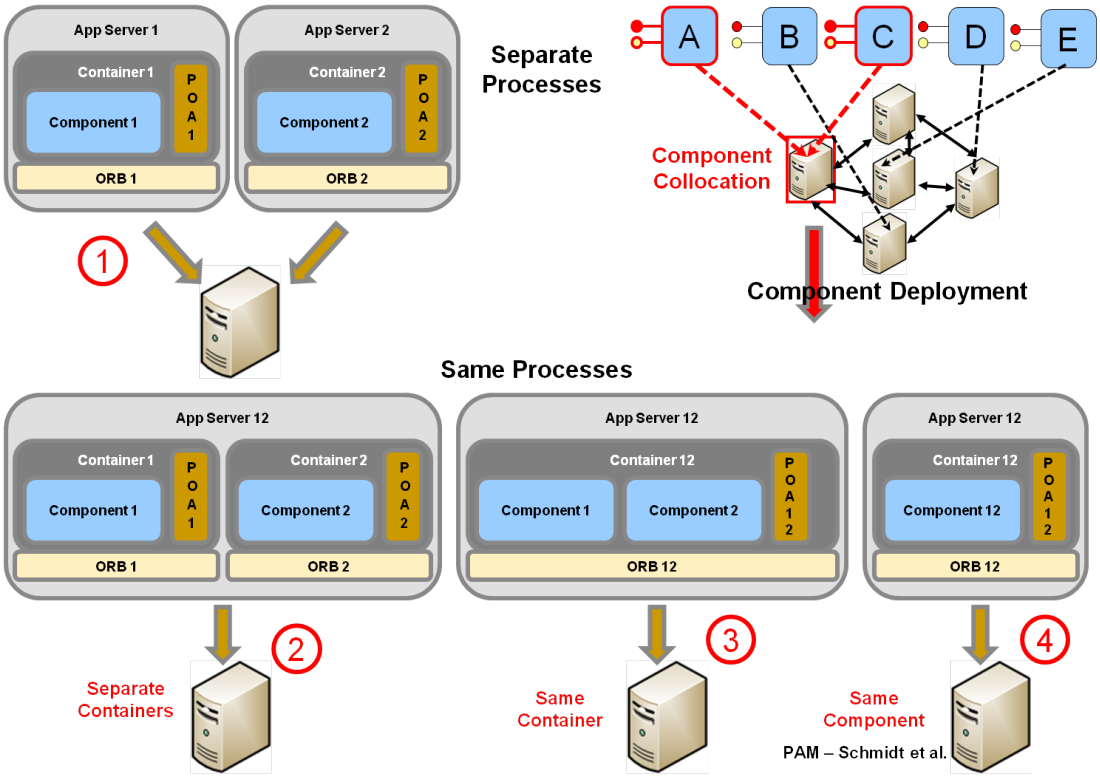


Figure 38: Composition Granularity

VII.4 Proposed Research: Safe Composition and Transparent Deployment of Specialized Middleware (DeCoM)

Addressing the challenges presented in section VII.3 requires solutions that satisfy two hypotheses. First, rethinking of alternatives to requiring application server redesign. Second, ensuring that the implementing the composition granularity due to collocation does not worsen per-component footprint and feature reduction by maximum 20

VII.4.1 Hypothesis 1: "Do No Harm"

Solution Approach: Safely Composing Application Server Frameworks Application servers frameworks are traditionally designed to be not only be syntactically composable through standard interfaces but also be semantically composable by providing expected behaviors to the layers above them. Therefore, as the constituent ORB and POA features are

pruned and specialized through the specialization process, it is not only necessary to ensure that the interfaces remain the same but it is also necessary to ensure that their expected behaviors match. It also necessary to ensure that the other server constituents that are dependent on the specialized entities function normally. However, redesigning the application server is not feasible.

In order to safely compose the specialized middleware frameworks (Container, POA, ORB, etc) within the application server stack, it is necessary to investigate new server specialization patterns. Dynamic patterns like Virtual Component can be leveraged to enable pluggability of the specialized constituents. It is also important to investigate techniques that can ensure end-to-end seamless operation of the entire operational string .Additionally, as described in section VII.2, to avoid the isolated, redundant and one-off customizations of the server for each QoS, it is necessary to develop a unified composition framework that will enable integration of different QoS in the same container (components swapping, realtime, fault tolerance, etc).

Even if the interfaces that compose the application server and the specialized middleware components are not modified, the reduced features provided by the middleware will finally impact the QoS provided by the container. Therefore, it is necessary to ensure the features and QoS provided by the specialized middleware is utilized consistently throughout the application server. Moreover, as multiple components are composed together it is necessary to ensure that the corresponding application servers on which the components are hosted work seamlessly and correctly. Not only the components need to work correctly between neighboring components but over the entire operational string that consists of a chain of application components.

VII.4.2 Hypothesis 2: "Whole does not exceed the Parts by 20%"

As shown in the figure 37, the task allocation [71] planning may mandate placing two or more components on the same target node. This will mandate composing the specialized

application server stacks for each component. The advantage of composition is that the overall footprint will be reduced drastically as there are no more dedicated middleware stacks running on a per-component basis.

Solution Approach: Transparent Augmentation of the Deployment Infrastructure Support Specialized Middleware The D&C infrastructure should deploy the application servers the same way as application components. Additionally, the components and the application servers need to configured according to the specialized functionality supported/required by them. Section VII.3.2 emphasizes how a deployment and configuration infrastructure should ideally be oblivious of the type of the application servers that host the components being deployed. Therefore, in order to support the transparent deployment of specialized application servers, the D&C infrastructure needs to be specialized with the necessary functionality and additional metadata. The D&C Plan Launcher and Execution Managers can be specialized with installation handlers that can transparently install the specialized application servers on the target nodes.

However, even if the overall system footprint reduces due to application server composition, the performance and feature overhead per component will be hampered. Therefore, it is necessary to investigate this trade off between composition and specialization. By determining the right composition granularity for the given component allocation, it maybe possible to alleviate this overhead and keep it within 20% for the composite compared to the individually specialized case. Constraint Optimization theory [49] can lend useful insights in this regard.

VII.5 Evaluation Criteria

To validate our hypotheses, our approach needs to be evaluated for safety and performance. While the application server is specialized, it is necessary to ensure that it's framework layers work seamlessly not only between themselves but also when the server

interacts with other application servers over the application operational string. The specialization approach needs to ensure application server composition and deployment are customized coherently while minimizing static and dynamic footprint while maintaining throughput and minimizing runtime overhead.

- Offer improved resource utilization
- Continue to support middleware design goals
- Extensive test cases to test inter-layer operations
- Minimizing static and dynamic footprint while maintaining throughput and minimizing runtime overhead
- Use RT-CCM as the specialization case study

CHAPTER VIII

CONCLUDING REMARKS

General-purpose middleware has been incrementally optimized over the period of time to efficiently handle the expected application functionality as well as provide the flexibility and adaptability to handle changing requirements and changing runtime conditions. However, the primary goal behind middleware design being generality and portability, it lacks finer customization and tunability to specific application requirements. To resolve this generality and specificity tension, middleware is usually specialized (customized and adapted) on a case-by-case basis. However this process becomes tedious and non-repeatable as the application requirements change as well as underlying platforms evolve. It is important that any modification to the middleware sources be retrofitted with minimal to no changes to the middleware portability, standard APIs interfaces, application software implementations, while preserving interoperability wherever possible. Otherwise such specialization approaches obviate the benefits accrued from using standards-based middleware. Additionally the accidental complexity from manually applying such approaches to mature middleware implementations renders the specializations tedious and error prone to implement.

In this PhD dissertation, we presented the research challenges involved in automating middleware specializations for component-based DRE systems. First, we discussed the challenge of tackling horizontal decomposition in traditional general-purpose middleware. Second, we motivated the need for a taxonomy for categorizing and reasoning about middleware specialization techniques. Third, we discussed the need for automating the middleware specialization process that reasons the application requirements in terms of middleware features and synthesizes the specializations directives using algorithms that transform the general-purpose middleware code into their optimized and specialized forms

with minimum developer intervention. We presented a multi-stage feature-oriented reasoning approach that infers middleware features from application requirements and determines the middleware specializations that are applicable. Next we presented an automated and generative process that uses novel and intuitive algorithms that generate the specialization directives to transform the middleware source and build configurations.

While an automated middleware specialization process addresses the traditional horizontal decomposition issues in general-purpose middleware and provides a systematic process of specializing middleware, run-time issues such as adapting the middleware safely and predictively to failures while improving resource utilization are not addressed adequately by current research. We sketched a solution in the form of a safe specialization methodology that would ensure safe adaptation of real-time middleware without adversely affecting other performance concerns such as application jitter and runtime processing overhead requirements of DRE systems while improving resource utilizations.

Table 8: Summary Of Research Contributions

Category	**Contributions**
Assessing Contemporary Middleware Specialization Techniques	**Taxonomy of Middleware Specializations:** A catalog that categorizes and reasons contemporary middleware specializing techniques along three dimensions of feature manipulation, development lifecycle and development paradigms.
Feature Oriented Reverse Engineering based Middleware Specializations	**FORMS:** A generic, feature oriented reasoning and specialization process for specializing middleware to reduce the footprint and amount of features being used. Can be adapted to work with any other kind of application and extended to work with other programming language platforms.
Generative Middleware Specializations	**GeMS:** A generative algorithm-based approach to automate the deduction of application invariants to infer specializations that are applicable and subsequently generating the specialization directives to transform the middleware sources.
Weaving Dependability Concerns in System Artifacts	**GRAFT:** An aspect-oriented transformation approach that generates the fault handling and masking code necessary for fault-tolerance provisioning
Safe Middleware Adaptation for Real-Time Fault-Tolerance	**SafeMAT:** A safe specialization methodology that would ensure safe adaptation of real-time middleware without adversely affecting other performance concerns such as application jitter and runtime processing overhead requirements of DRE systems while improving resource utilizations.

APPENDIX A

UNDERLYING TECHNOLOGIES

This appendix summarizes the various technologies that are used to build the middleware specialization techniques and the fault-tolerant middleware adaptation solutions that are described in this thesis.

A.1 Aspect Oriented Programming (AOP) Terminologies

Aspects modularize crosscutting concerns, coding concerns that are not localized, hence, not modularized. Aspect-oriented programming (AOP) allows the developer to cleanly encapsulated crosscutting concerns in separate modules [63]. Aspect-oriented languages, such as AspectJ, defines a set of new language constructs to support two kinds of crosscutting: *dynamic crosscutting* and *static crosscutting*. *Dynamic crosscutting* is defined by means of join points that denote well-defined points in the execution of a program. A *Pointcut* refers to a collection of join points and parameters associated with these *join points*. A method-like construct, referred to as an *advice*, is used to define aspect code executed before, after or in place of a join point. *Static crosscutting* affects the static structure of a program, such as classes, interfaces, and the type hierarchy whereas *dynamic crosscutting* affects the runtime behavior. *Inter-type declarations* are used to *introduce* new fields and methods into classes or interfaces. The *declare parents* construct is used to modify the existing type hierarchy. An aspect module includes pointcuts, the associated advices, inter-type declarations, and declare parents constructs.

A.2 Model-Driven Development (MDD)

Model-driven development refers to a software development process that is based on models of the software synthesized code. The *Model Driven Architecture process (MDA)*

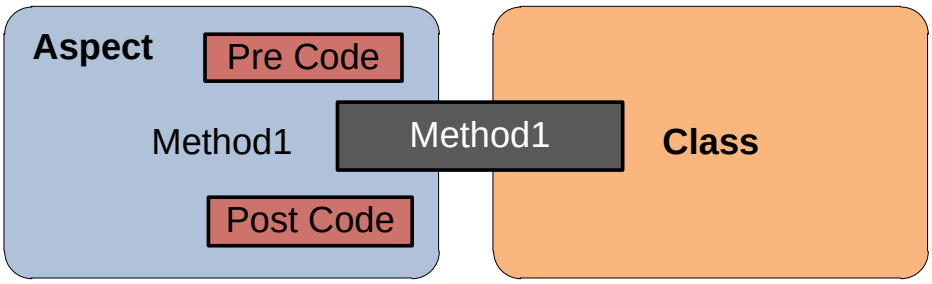


Figure 39: Aspect Oriented Programming (AOP)

is one prominent examples of a model-driven development approach. MDA advocates developing complex systems through multiple and hierarchical viewpoints. The *Platform Independent Viewpoint* and the associated *Platform Independent Model (PIM)* does not specify the details necessary for running the system on a particular platform, which makes it suitable for abstracting the essential functionality of a system across a number of middleware platforms. By combining the specifications of the PIM with the details of how to use a particular type of platform, a *Platform Specific Model (PSM)* is established. A set of mapping rules relate a PIM to its PSM that lays out the details with respect to a given middleware platform. How mappings can be effectively realized is still in question. The approach suggested in this paper is one possible realization for automating the mapping between different views and models.

A.3 Overview of Lightweight CCM

The OMG Lightweight CCM (LwCCM) [89] 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.

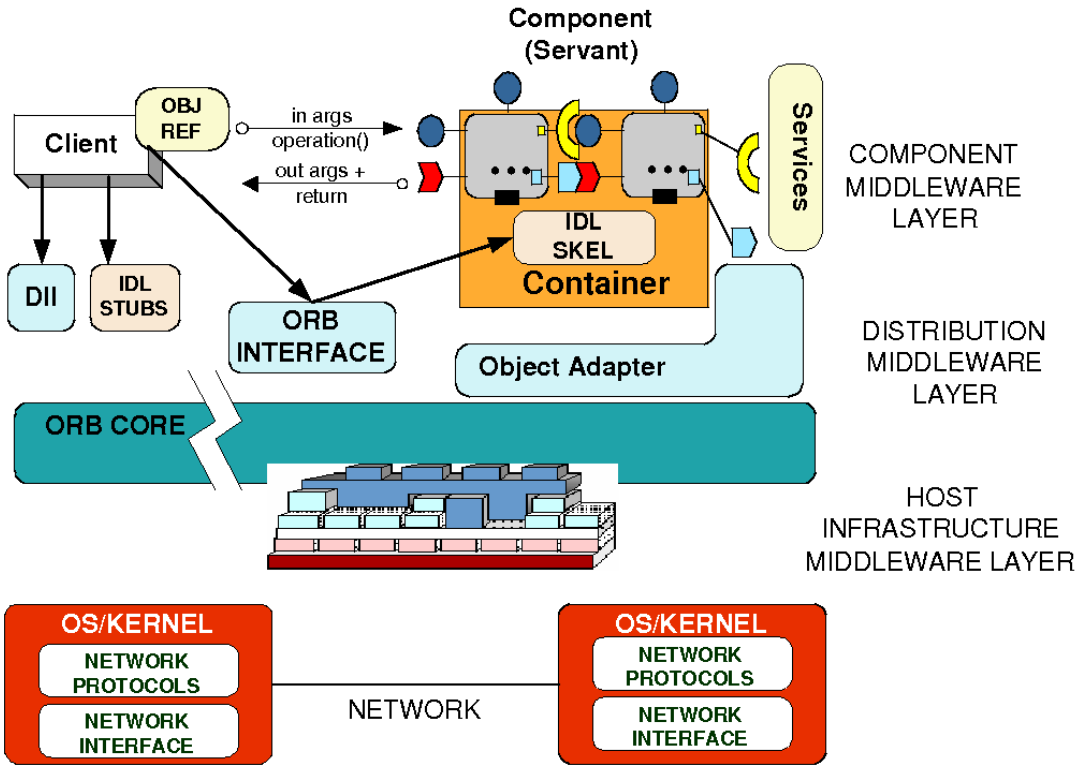


Figure 40: Layered LwCCM Architecture

Figure 40 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

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.4 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 41 gives an overview of the OMG D&C model, which is standardized by OMG through the Deployment and Configuration (D&C) [90] 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.

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.

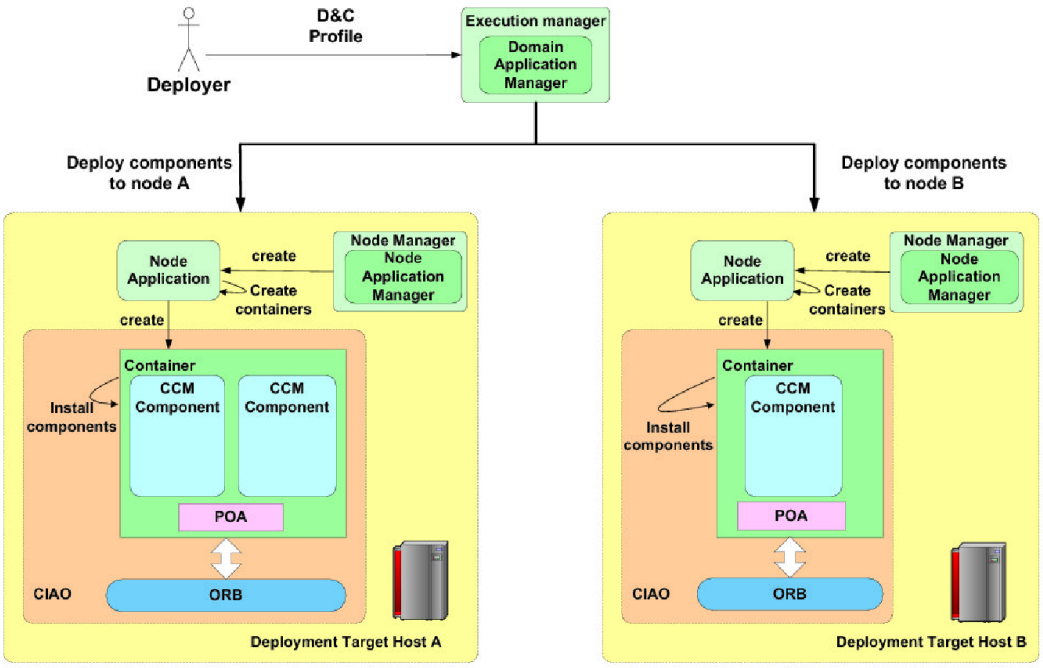


Figure 41: An Overview of OMG Deployment and Configuration Model

A.5 The ARINC-653 Component Model (ACM)

ACM combines the CORBA Component Model [133] with ARINC-653 [6]. ACM components interact with each other via well-defined patterns, facilitated by ports: asynchronous connections (event publishers & consumers) and/or synchronous provided/required interfaces (facets/receptacles). ACM allows the developers to group a number of ARINC-653 processes into a reusable component. Since this framework is geared for hard real-time systems, it is required that each port is statically allocated to an ARINC-653 process whereas every method of a facet interface is allocated to a separate process.

1. The ACM Modeling Environment

The ACM modeling environment captures (1) the component's interaction ports, conditions associated with the ports, (2) the real-time properties (priority, periodicity, deadline, worst case execution time etc.) and resource requirements (CPU, stack size) of the ports and the component, the data and control flow within the component, and (optionally) the local component level health management strategy (CLHM) part of the two-level Health

Management [33] using a domain specific modeling language and associated tools. The modeling tool allows the specification of the platform in terms of the modules (processors) and the partitions (processes) within each module. The integrator can specify the deployment of each component (group of threads) into an appropriate partition such that the *temporal partitioning* concerns are satisfied. Lastly, integrator can specify whether a Software Health Management (SHM) module should be generated for the assembly or not. Tools included with the modeling environment generate glue code that is responsible for implementing the ports, binding each port with an ARINC-653 process and the integration code and configuration files.

2. The ACM Middleware

The ACM middleware is composed of layers that are instantiated and configured for runtime. These layers are described next.

The Module Manager (MM) is the main controller responsible for providing *temporal partitioning* among partitions (i.e., Linux processes). For this purpose, each module is bound to a single core of the host processor. The module manager is configured with a fixed cyclic schedule computed from the specified partition periods and durations. It is specified as offsets from the start of the hyper period, duration and the partition to run in that window. Once configured and validated, the module manager implements the schedule using the `SCHED_FIFO` policy of the Linux kernel and manages the execution and preemption of the partitions. The module manager is also responsible for transferring the inter-partition messages across the configured channels. Figure 42 shows the example execution time line of a module with two partitions and a hyper period of 2 seconds.

In case of a distributed system, there can be multiple module managers each bound to a processor core that are controlled hierarchically by a system level module manager.

The APEX Partition Scheduler is instantiated for each partition using the APEX services emulation library that implements a priority-driven preemptive scheduling algorithm using Linux `SCHED_FIFO` scheduler. It initializes and schedules the ARINC-653

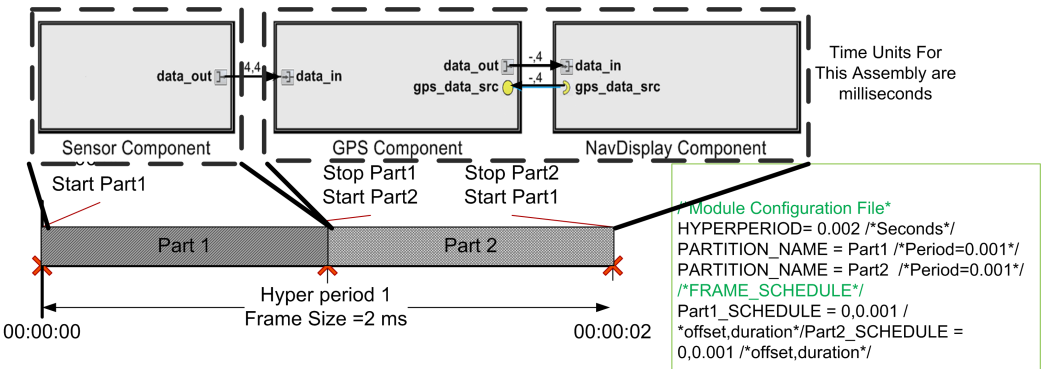


Figure 42: A module configuration and the time line of events as they occur.

processes inside the partition based on their periodicity and priority. It ensures that all processes, periodic as well as aperiodic, finish their execution within the specified deadline.

TAO Object Request Broker (ORB) The main TAO [56] ORB thread is executed as an aperiodic ARINC-653 process within the respective partition. For controllability, the ORB runs at a lower priority than the partition scheduler does. Since ARINC does not allow dynamic creation of processes at run-time, the ORB is configured to use a predefined number of worker threads (i.e. ARINC-653 Processes) that are created during initialization.

Component and Process Layers This layer provides the glue code, generated from the definitions of components and their interfaces specified in the modeling environment in order to map the concepts of component model into the concepts exposed by the ARINC Emulator layer and the TAO ORB layer. The system developer provides the functional code. This layer also consists of CLHMs that are special processes that can take mitigation actions, if required.

3. Software Health Management (SHM) in ACM

Software Health Management (SHM) in ACM happens at two levels. The first level of protection is provided by a component level health management (CLHM) strategy, which is implemented in all components. It provides a localized timed state machine with state

transitions triggered either by a local anomaly or by timeouts, and actions that perform the local mitigation. The System Level Health Manager (SLHM) is at the second, top level in our health management strategy. The deployment of the SLHM requires the addition of three special SLHM components to an ACM assembly: the *Alarm Aggregator*, The *Diagnosis Engine*, and the *Deliberative Reasoner*, as shown in Figure 43.

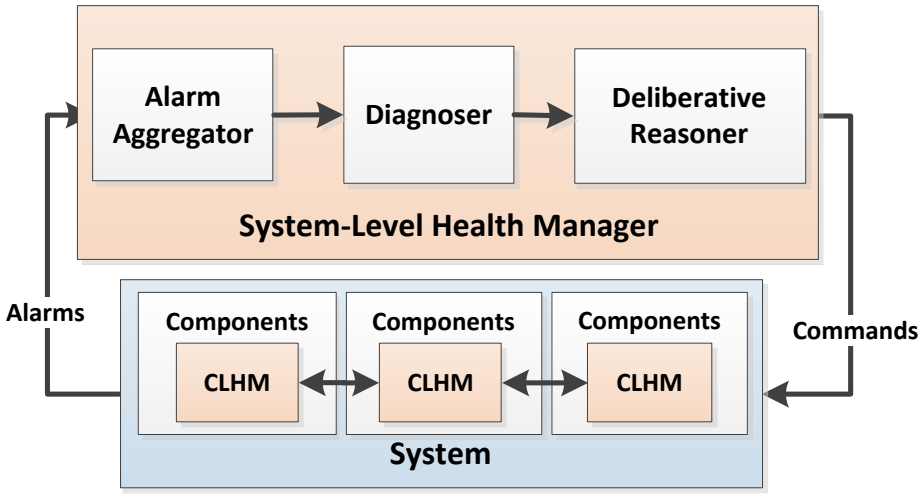


Figure 43: SHM architecture.

The *Alarm Aggregator* is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). This information is collected using a moving window two hyperperiods long. The events are sorted based on their time of occurrence and then sent to the *Diagnosis Engine*. The *Diagnosis Engine* is initialized by a Timed Failure Propagation Graph (TFPG) [1] model that captures the failure-modes, discrepancies (possibly indicated by the alarms), and the failure propagations from failure modes to discrepancies and from discrepancies to other discrepancies, across the entire system [35, 73]. The reasoner uses this model to isolate

the most plausible failure source: a software component that could explain the observations, i.e., the alarms triggered and the CLHM commands issued. The result, i.e., the list of faulty components is reported to the next component that provides the system level mitigation: the *Deliberative Reasoner*.

APPENDIX B

LIST OF PUBLICATIONS

Research on FORMS, GeMS, GRAFT, and SafeMAT has led to the following journal, conference, and workshop publications.

B.1 Refereed Journal Publications

1. Akshay Dabholkar, Abhishek Dubey, and Aniruddha Gokhale, "SafeMAT: Safe Middleware Adaptation for Predictable Fault-Tolerant Distributed Real-time and Embedded Systems," *(In submission), 2012.*

2. Akshay Dabholkar, and Aniruddha Gokhale, "AutoGeMS: An Automated and Generative Middleware Specializations Process for Distributed Real-time and Embedded Systems," *(Submitted to) Elsevier Journal of Software Architecture (JSA), 2012*

3. Akshay Dabholkar, and Aniruddha Gokhale, "FORMS: Feature-Oriented Reverse Engineering-based Middleware Specialization for Product-Lines," *Journal of Software Special Issue on Middleware and Network Application (JSW), Vol.6, No.4, 2011*

B.2 Refereed Conference Publications

1. Akshay Dabholkar, Abhishek Dubey, and Aniruddha Gokhale, "Reliable Distributed Real-time and Embedded Systems Through Safe Middleware Adaptation," *(In submission to) 31st International Symposium on Reliable Distributed Systems (SRDS), 2012.*

2. Akshay Dabholkar, and Aniruddha Gokhale, "A Generative Middleware Specialization Process for Distributed Real-time and Embedded Systems," *Proceedings of*

theă14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC), 2011.

3. Akshay Dabholkar, and Aniruddha Gokhale, "Middleware Specialization for Product-lines using Feature Oriented Reverse Engineering," *Proceedings of theă7th International Conference on Information Technology : New Generations (ITNG), 2010.*

4. Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale, "MoPED: A Model-based Provisioning Engine for Dependability in Component-based Distributed Real-time Embedded Systems," *Proceedings of theă18th IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS), 2011.*

5. 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.*

6. 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.*

7. Nilabja Roy,ăAkshay Dabholkar, Nathan Hamm, Larry Dowdy, and Douglas Schmidt, "Modeling Software Contention using Colored Petri Nets," *Proceedings of theă16th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2008.*

B.3 Refereed Workshop Publications

1. Akshay Dabholkar, and Aniruddha Gokhale, "Safe Specialization of the LwCCM Container for Simultaneous Provisioning of Multiple QoS," *Proceedings of OMGŠs*

Workshop on Real-time, Embedded and Enterprise-Scale Time-Critical Systems (OMG RTWS), 2011.

2. Akshay Dabholkar, andăAniruddha Gokhale, "An Approach to Middleware Specialization for Cyber Physical Systems," *Proceedings ofăThe 2nd International Workshop on Cyber-Physical Systems (WCPS),ăCo-located withăICDCS pp. 73Ű79, 2009.*

3. Akshay Dabholkar, and Aniruddha Gokhale, "Developing and Evaluating a Taxonomy of Modularization Techniques for Middleware Specialization," *Proceedings of theă2nd OOPSLAăWorkshop on Assessment of Contemporary Modularization Techniques (ACoM), 2007.*

4. 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.*

5. 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.*

B.4 Technical Reports

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

B.5 Poster Publications

1. Akshay Dabholkar, and Aniruddha Gokhale, "Architecture-Driven Context-Specific Middleware Specializations for Distributed Real-time and Embedded Systems," *Proceedings of theăACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES-WIP-PS), 2010.*

2. Akshay Dabholkar, Sumant Tambe andăAniruddha Gokhale , "An Systematic Approach to Middleware Specialization for Cyber Physical Systems," *Published in theăProceedings of the Cyber Physical Systems Week (CPS-Week), 2009.*

3. Akshay Dabholkar, and Aniruddha Gokhale, "Towards Employing End-to-End Middleware Specialization Techniques," *Proceedings of OMGŠs Annual Real-time and Embedded Systems workshop (OMG RTWS), 2008.*

4. Joe Hoffert, Akshay Dabholkar, Aniruddha Gokhale, and Douglas Schmidt, 'Enhancing Security in Ultra-Large Scale (ULS) Systems using Domain-specific Modeling," *Spring Conference for Team for Research in Ubiquitous Secure Technology (TRUST), 2007.*

REFERENCES

[1] S. Abdelwahed, G. Karsai, N. Mahadevan, and S. C. Ofsthun. Practical considerations in systems diagnosis using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, February 2009.

[2] 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.

[3] Gul A. Agha. Introduction. *Communications of the ACM*, 45(6):30–32, 2002. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/508448.508469.

[4] 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.

[5] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *Software Engineering, IEEE Transactions on*, 34(2):162–180, March-April 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70770.

[6] ARINC. ARINC specification 653-2: Avionics application software standard interface part 1 - required services. Technical report, ARINC Incorporated, Annapolis, Maryland, USA, May 2010.

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

[8] 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.

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

[10] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: http://dx.doi.org/10.1109/RTAS.2005.4.

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

[12] BEA Systems, et al. *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.

[13] 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.

[14] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[15] U. Black. *OSI: A Model for Computer Communications Standards*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[16] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, London, 1998. Springer-Verlag.

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

[18] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[19] Nélio Cacho and Thaís Vasconcelos Batista. Using AOP to Customize a Reflective Middleware. In *OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 1133–1150. Springer, 2005. ISBN 3-540-29738-3.

[20] Zhongtang Cai, Vibhore Kumar, Brian F. Cooper, Greg Eisenhauer, Karsten Schwan, and Robert E. Strom. Utility-Driven Proactive Management of Availability in Enterprise-Scale Information Flows. In *Proceedings of ACM/Usenix/IFIP Middleware*, pages 382–403, 2006.

[21] 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.

[22] Venkat Chakravarthy, John Regehr, and Eric Eide. Edicts: Implementing Features with Flexible Binding Times. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, pages 108–119, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-044-9. doi: http://doi.acm.org/10.1145/1353482.1353496.

[23] 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.

[24] Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, and Carlos O'Ryan. Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications. In *Proceedings of the 9^{th} Annual Conference on the Pattern Languages of Programs*, Monticello, IL, September 2002.

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

[26] Akshay Dabholkar and Aniruddha Gokhale. FORMS: Feature-Oriented Reverse Engineering-based Middleware Specialization for Product-Lines. *Journal of Software (JSW) - Special Issue on Recent Advances in Middleware and Network Applications*, 6(4):519–527, 2011. ISSN 1796-217X.

[27] Pierre-Charles David, Thomas Ledoux, and Noury M.N. Bouraqadi-Saadani. Two-step Weaving with Reflection using AspectJ. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001.

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

[29] Ömer Erdem Demir, Premkumar T. Devanbu, Eric Wohlstadter, and Stefan Tai. An aspect-oriented approach to bypassing middleware layers. In Brian M. Barry and Oege de Moor, editors, *AOSD*, volume 208 of *ACM International Conference Proceeding Series*, pages 25–35. ACM, 2007. ISBN 1-59593-615-7.

[30] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet. Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 90, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0290-3.

[31] BerliOS Developer. The source-navigator^{TM} ide. http://sourcenav.sourceforge.net/, 2007.

[32] Jing Dong, Yajing Zhao, and Tu Peng. Architecture and design pattern discovery techniques - a review. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 621–627. CSREA Press, 2007. ISBN 1-60132-034-5.

[33] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. Towards model-based software health management for real-time systems. Technical Report ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University, August 2010. URL http://isis.vanderbilt.edu/node/4196.

[34] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. A component model for hard real-time systems: CCM with ARINC-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011. ISSN 1097-024X. doi: 10.1002/spe.1083. URL http://dx.doi.org/10.1002/spe.1083.

[35] Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. Model-based software health management for real-time systems. In *Aerospace Conference, 2011 IEEE*, march 2011. to appear. Draft available at http://isis.vanderbilt.edu/sites/default/files/PaperSubmission.pdf.

[36] Abhishek Dubey, Nagabhushan Mahadevan, and Gabor Karsai. A deliberative reasoner for model-based software health management. In *The Eighth International Conference on Autonomic and Autonomous Systems*, 2012. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2010.39. to appear.

[37] Abhishek Dubey, Nagabhushan Mahadevan, and Gabor Karsai. The inertial measurement unit example: A software health management case study. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 02/2012 2012.

[38] Chad Elliott. The makefile, project, and workspace creator (mpc). www.ociweb.com/products/mpc, Sep 2007.

[39] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 117–129, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336542.

[40] Fábio M. Costa and Gordon S. Blair. A Reflective Architecture for Middleware: Design and Implementation. In *ECOOP'99, Workshop for PhD Students in Object Oriented Systems*, June 1999.

[41] Marc Fleury and Francisco Reverbel. The JBoss Extensible Server. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware*

2003), Rio De Janeiro, Brazil, pages 344–373, 2003.

[42] Lorenz Froihofer, Karl M. Goeschka, and Johannes Osrael. Middleware support for adaptive dependability. In *Middleware*, pages 308–327, 2007.

[43] Aniruddha Gokhale, Dimple Kaul, Arundhati Kogekar, Jeff Gray, and Swapna Gokhale. POSAML: A Visual Modeling Language for Managing Variability in Middleware Provisioning. *Elsevier Journal of Visual Languages and Computing (JVLC) 2007*, 18(4):359–377, 2007.

[44] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *RTSS '97*, page 79, San Francisco, CA, USA, 1997. ISBN 0-8186-8268-X.

[45] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996. ISSN 1046-8188. doi: http://doi.acm.org/10.1145/230538.230540.

[46] Carl L. Hall. *Building client/server applications using TUXEDO*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-12958-5.

[47] William Harrison and Harold Ossher. Subject-oriented Programming: A Critique of Pure Objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: http://doi.acm.org/10.1145/165854.165932.

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

[49] J. Hooker, G. Ottosson, E.S. Thorsteinsson, and H.J. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review*, 15(1):11–30, 2000.

[50] Eugene S. Hudders. *CICS: a guide to internal structure*. Wiley-QED Publishing, Somerset, NJ, USA, 1994. ISBN 0-471-52172-8.

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

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

[53] Software Engineering Institute. Ultra-Large-Scale Systems: Software Challenge of

the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, June 2006.

[54] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/, Vanderbilt University.

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

[56] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

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

[58] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic Scheduling of Distributed Method Invocations. In *21st IEEE Real-time Systems Symposium*, Orlando, FL, November 2000. IEEE.

[59] Vana Kalogeraki, P. M. Melliar-Smith, L. E. Moser, and Y. Drougas. Resource Management Using Multiple Feedback Loops in Soft Real-time Distributed Systems. *Journal of Systems and Software*, 2007.

[60] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 311–320, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368131.

[61] Panagiotis Katsaros and Constantine Lazos. Optimal object state transfer - recovery policies for fault tolerant distributed systems. In *Proc. of DSN. (2004)*.

[62] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-61074-4.

[63] 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 (ECOOP'97)*, pages 220–242, June 1997.

[64] K. H. (Kane) Kim and Chittur Subbaraman. The pstr/sns scheme for real-time fault tolerance via active object replication and network surveillance. *IEEE Trans. on Know. and Data Engg.*, 12(2), 2000. ISSN 1041-4347. doi: dx.doi.org/10.1109/69.842258.

[65] Raymond Klefstad, Arvind S. Krishna, and Douglas C. Schmidt. Design and Performance of a Modular Portable Object Adapter for Distributed, Real-time, and Embedded CORBA Applications. In *Proceedings of the 4th International Symposium on*

Distributed Objects and Applications, Irvine, CA, October/November 2002. OMG.

[66] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. Towards Highly Configurable Real-time Object Request Brokers. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2002. IEEE/IFIP.

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

[68] Arvind Krishna, Aniruddha Gokhale, Douglas C. Schmidt, John Hatcliff, and Venkatesh Ranganath. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*, pages 205–218, Leuven, Belgium, April 2006.

[69] Sudha Krishnamurthy, William H. Sanders, and Michel Cukier. An Adaptive Quality of Service Aware Middleware for Replicated Services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, 2003. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2003.1247672.

[70] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. *Transactions on AOSD II*, 4242:227–255, 2006.

[71] Perng-Yi Richard Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Trans. Comput.*, 31(1):41–47, January 1982. ISSN 0018-9340. doi: 10.1109/TC.1982.1675884. URL http://dx.doi.org/10.1109/TC.1982.1675884.

[72] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/38807.38821.

[73] Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai. Application of software health management techniques. In *Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, New York, NY, USA, 2011. ACM, ACM.

[74] Olivier Marin, Marin Bertier, and Pierre Sens. Darx: A framework for the fault-tolerant support of agent software. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 406, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3.

[75] Renaud Marlet, Scott Thibault, and Charles Consel. Efficient Implementations of Software Architectures via Partial Evaluation. *Automated Software Engineering:*

An International Journal, 6(4):411–440, October 1999. URL citeseer.csail.mit.edu/marlet99efficient.html.

[76] M.D.W. McIntyre and C.A. Gossett. The boeing 777 fault tolerant air data and inertial reference system-a new venture in working together. In *Digital Avionics Systems Conference, 1995., 14th DASC*, pages 178 –183, November 1995. doi: 10.1109/DASC.1995.482827.

[77] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643613.

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

[79] *Distributed Component Object Model Protocol (DCOM)*. Microsoft Corporation, 1.0 edition, January 1998.

[80] 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.

[81] 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.

[82] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Oper. Syst. Rev.*, 42(4):233–246, 2008. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1357010.1352617.

[83] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.328995.

[84] *Interceptors FTF Final Published Draft*. Object Management Group, OMG Document ptc/00-04-05 edition, April 2000.

[85] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.6*. Object Management Group, December 2001.

[86] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Version 3.0*. Object Management Group, July 2001.

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

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

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

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

[91] *Model Driven Architecture (MDA) Guide V1.0.1*. Object Management Group, OMG Document omg/03-06-01 edition, June 2003.

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

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

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

[95] 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.

[96] Ömer Erdem Demir, Prémkumar Dévanbu, Eric Wohlstadter, and Stefan Tai. An Aspect-oriented Approach to Bypassing Middleware Layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, Vancouver, British Columbia, Canada, 2007. ACM Press. ISBN 1-59593-615-7. doi: doi.acm.org/10.1145/1218563.1218567.

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

[98] openArchitectureWare. openArchitectureWare. www.openarchitectureware.org, 2007.

[99] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[100] Terry Pearson. Save time and money with COTS middleware for network equipment. www.commsdesign.com/printableArticle/?articleID=174402378, November 2005.

[101] Soila Pertet and Priya Narasimhan. Proactive recovery in distributed corba applications. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 357, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2052-9.

[102] 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.

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

[104] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, 14 (1):36–47, 1994. ISSN 0272-1732. doi: dx.doi.org/10.1109/40.259898.

[105] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer. ISBN ISBN 3-540-63089-9. URL citeseer.nj.nec.com/195556.html.

[106] Manuel Roman, Roy H. Campbell, and Fabio Kon. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), July 2001.

[107] Ward Rosenberry, David Kenney, and Gerry Fischer. *Understanding DCE*. O'Reilly and Associates, Inc., 1992.

[108] 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.

[109] S. Sadjadi, P. McKinley, and E. Kasten. Architecture and operation of an adaptable communication substrate, 2003. URL citeseer.ist.psu.edu/sadjadi03architecture.html.

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

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

2006.

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

[113] 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.

[114] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[115] 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.

[116] 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.

[117] 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.

[118] A.N. Srivastava and J. Schumann. The case for software health management. In *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*, pages 3–9. IEEE, 2011.

[119] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/383845.383865.

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

[121] *Java Remote Method Invocation Specification (RMI)*. Sun Microsystems, Inc, October 1998.

[122] 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.

[123] 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.

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

[125] 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.

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

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

[128] Anand Tripathi. Challenges Designing Next-Generation Middleware Systems. *Communications of the ACM*, 45(6):39–42, June 2002.

[129] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 44–53, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.36.

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

[131] George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers (Elsevier), San Francisco, CA, 2005.

[132] Bart Verheecke and María Agustina Cibrán. Aop for dynamic configuration and management of web services. In *In Proceedings of 2003 International Conference on Web Services*, page 2004, 2003.

[133] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. An Overview of the CORBA Component Model. In George Heineman and Bill Councill, editors, *Component-Based Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2000.

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

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

[136] Eric Wohlstadler, Stoney Jackson, and Premkumar Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems . In *Proceedings of the International Conference on Software Engineering*, Portland, OR, May 2003.

[137] 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*, 57(6):597–613, August 2010. ISSN 1383-7621. doi: DOI:10.1016/j.sysarc.2010.07.006. URL http://www.sciencedirect.com/science/article/B6V1F-50RP25M-1/2/ce8f29d70c51c80a123a38186dcd362c.

[138] W. Wolf. Cyber-Physical Systems. *Computer*, 42(3):88–89, 2009. ISSN 0018-9162.

[139] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 85–92, New York, NY, USA, 2002. ACM. ISBN 1-58113-609-9. doi: http://doi.acm.org/10.1145/582128.582144.

[140] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-9. doi: http://doi.acm.org/10.1145/1028976.1028992.

[141] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards Just-in-time Middleware Architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, Chicago, Illinois, 2005. ACM Press. ISBN 1-59593-042-6. doi: doi.acm.org/10.1145/1052898.1052904.

[142] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Generic Middleware Substrate Through Modelware. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*, pages 314–333, Grenoble, France, 2005.

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

[144] 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.