AutoMaS: An Automated Middleware Specialization Process for Distributed Real-time and Embedded Systems

AKSHAY DABHOLKAR and ANIRUDDHA GOKHALE
Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA

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 adapt to changing conditions. The first two challenges are addressed to a large extent by standardized, general-purpose middleware (e.g. CORBA, DCOM and Java RMI) through the use of a "black-box" approach, such as the object-oriented paradigm (frameworks and design patterns). The need to support a large variety and range of applications and application domains has resulted in very feature-rich implementations of these standardized middleware. However, such a feature-richness acts counteractive to resolving the remaining two challenges; instead it incurs excessive memory footprint and performance overhead, as well as increased cost of testing and maintenance. To address the four challenges all at once while leveraging the benefits of general-purpose middleware requires a scientific approach to specializing the middleware. Software engineering techniques, such as aspect-oriented programming (AOP), feature-oriented programming (FOP), and reflection make the specialization task simpler, albeit still requiring the DRE system developer to manually identify the system invariants, and sources of performance and memory footprint bottlenecks that drive the specialization techniques. Specialization reuse is also hampered due to a lack of common taxonomy to document the recurring specializations, and assess the strengths and weaknesses of these techniques.

To address these requirements, this paper presents a case for an automated, multi-stage, feature-oriented middleware specialization process that improves both middleware developer productivity and middleware performance. Three specific contributions are made in this paper. First, contemporary middleware specialization research is framed in terms of a three-dimensional taxonomy. Second, 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, a concrete implementation of the six stage, automated middleware specialization process is presented along with empirical data illustrating the benefits accrued using the framework.

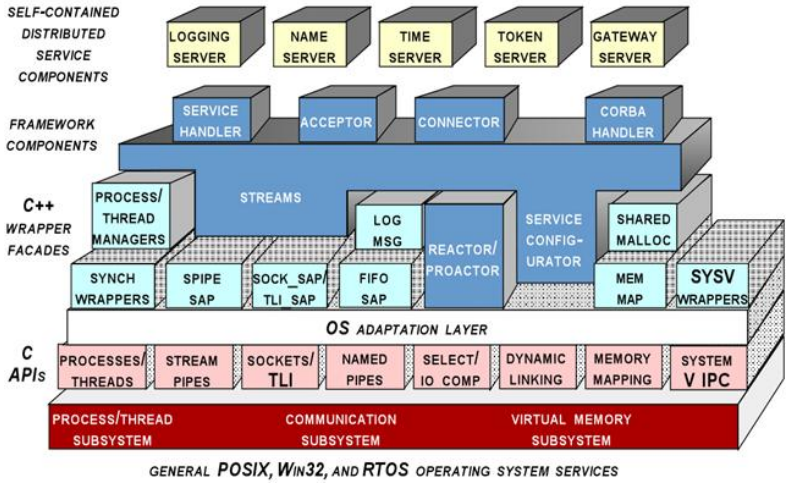
Keywords: Middleware, specialization, optimization, feature-oriented programming, aspect-oriented programming, design patterns

1. INTRODUCTION

1.1 Emerging Trends and Technologies

A large variety of applications and application product lines such as those found in avionics [Sharp and Roll 2003], telecommunication call processing, multimedia streaming video, industrial automation, multi-satellite missions [Suri et al. 2006], shipboard computing [Schmidt et al. 2001] 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 leverage general-purpose middleware in their design and implementation due to its many benefits such as lowering the time-to-market and hiding accidental complexities [Brooks 1987] 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.

This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation



Middleware Patterns and Frameworks (ACE [Schmidt 1997])

Despite these benefits, general-purpose middleware poses numerous challenges when developing DRE systems. First, the generality of these middleware make them very feature-rich. Yet, DRE applications are likely to use only a fraction of these features in their implementation. Moreover, DRE systems impose stringent demands 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). Thus, the feature-richness and flexibility of general-purpose middleware becomes a source of excessive memory footprint overhead where memory and other resources are already at a premium, 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 systems in two different domains, such as 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., make them general-purpose.

Consequently, developers are often faced with either developing proprietary and customized middleware solutions, or reinventing pieces of a middleware that are 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 impact of future revisions on all of the derived subsets. Moreover, adapting the middleware to changing requirements is hard to achieve without the right tools and techniques. 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.

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 their different systemic properties. We call this approach *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

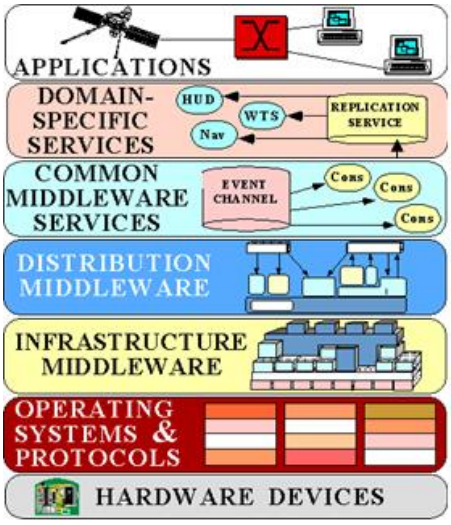
an issue that is distinct from specialization. This paper focuses on middleware specialization techniques.

1.2 Research Challenges in Middleware Specialization

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

Most prior efforts at specializing middleware (and other system artifacts) [Lohmann et al. 2006; Hunleth and Cytron 2002; Zhang et al. 2005a; Wohlstadler et al. 2003; Ömer Erdem Demir et al. 2007; Chakravarthy et al. 2008] 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 only 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:

1. **Spatial disparity between OO-based middleware design and 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 2. 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 [Gottlob et al. 1996] thereby necessitating *vertical decomposition*. For example, in OO-based middleware implementations of Real-time CORBA (RTCORBA) [Object Management Group 2005], 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_PRIORITY` 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.



Middleware Layers

2. **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 workflows made up of connected application components. 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 changes involving graceful degradation in their QoS as opposed to an abrupt denial. For instance, redundant DRE application workflows could be deployed in a surveillance system differing only in their QoS. A primary workflow and its underlying resources could have been configured for high-resolution, low-latency image processing whereas one or more alternate workflows could be configured to use gradually inferior QoS, which is to be used only if the primary workflow fails. The ability to switch between these workflows when failure occurs is necessary. Moreover, to fail over in a transparent fashion is also a challenge.
3. **Lack of apriori 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 from the previous phase 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.
4. **Lack of mechanisms for reusing specializations -** Unlike the years of efforts in documenting proven 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.

1.3 Research Contributions in Middleware Specialization

To address the middleware specialization challenges identified in Section 1.2, this paper describes (1) the contemporary research in specializing middleware in terms of a three dimensional taxonomy we have developed, (2) a feature-oriented approach to reason about (a) application requirements and their composition, (b) deployment and QoS models to determine the middleware features that are required by the application components, and (c) the specialization context. Together these help drive the specializations to be performed on the underlying middleware in order to support their QoS demands, and (3) the reverse-engineering, generative and aspect-oriented programming (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. The research contributions made in this paper are summarized below and explained in detail in the rest of the paper:

(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 as shown in our work. Section 2 describes the specialization techniques and their taxonomy in detail.

(2) **Process for Automated Middleware Specializations** illustrates a process we have developed to automate the middleware specialization process, which is described in Section 3.

(3) **Feature-oriented Reasoning Techniques to drive the Middleware Specializations** has been demonstrated using a decision tree-based reasoning approach that determines which middleware features 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. Section 4 describes the reasoning and deduction techniques for driving specializations in detail.

(4) **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. Section 5 describes this approach in detail.

1.4 Paper Organization

The remainder of this paper is organized as follows: Section 2 describes contemporary middleware specializations classifying them into a three-dimensional taxonomy, and compares and contrasts our solutions with related research. Section 3 describes a process for automated middleware specialization. Section 4 describes feature-oriented reasoning techniques for discovering opportunities to drive middleware specializations. Section 5 presents the automated and generative transformation approach and the corresponding algorithms for specializing middleware and its build system. Section 6 evaluates the claims made by AutoMaS. Finally, Section 7 presents the concluding remarks alluding to lessons learned and guidelines to apply specializations.

2. A TAXONOMY FOR MIDDLEWARE SPECIALIZATION

This section surveys and evaluates the existing body of research in middleware specializations and categorizes it into the three dimensions of lifecycle, paradigm and feature manipulation. Examples of each category are described and compared in detail. The section subsequently organizes the surveyed research into a taxonomy representation and proposes a multi-stage lifecycle for specializing general-purpose middleware. Such a taxonomy is essential to situate individual solutions we have developed for middleware specializaton. Finally, related work is compared with our solutions.

Contemporary research on middleware can be broadly classified along three dimensions of application development: (1) feature-dependent, (2) paradigm-dependent, and (3) lifetime-dependent.

2.1 Feature-Dependent Specialization

Feature-oriented programming (FOP) captures the variants of a base behavior through a layer of encapsulation of multiple abstractions and their respective increments that together pertain to the definition of a feature [Mezinia and Ostermann 2004]. 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:

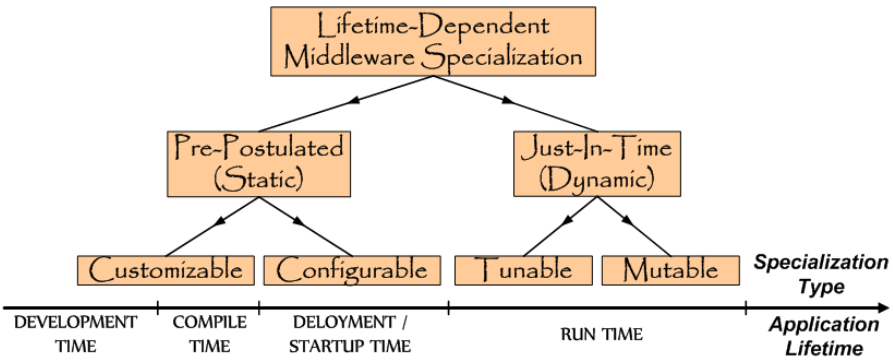
2.1.1 *Feature Pruning.* Feature pruning is a strategy applied to remove features of the middleware to customize it. In this case the original middleware provides a broad range of features but many are not needed for a given use case. These unwanted features are pruned from the original middleware. This approach is taken by FOCUS [Krishna et al. 2006] where unnecessary features are automatically removed from general-purpose middleware through techniques such as memoization to provide optimizations for DRE systems.

2.1.2 *Feature Augmentation.* Feature augmentation is a strategy applied when the specialization is realized via the insertion of new features, either because the original middleware did not support it or the middleware is composed out of building blocks [Agha 2002; Blair et al. 1998; Tripathi 2002]. 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.

As described in Section 2.3.2, AOP can be used to compose middleware platforms where the middleware core contains only the basic functionalities [Hunleth and Cytron 2002; Zhang et al. 2005a]. 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.

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* [Zhang et al. 2005b], which is illustrated in Figure 3. If middleware specialization is performed during the application compile or startup time, we designate it as *pre-postulated/static specialization.* For example, EmbeddedJava (`java.sun.com/products/embeddedjava`) 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 [Sadjadi et al. 2003] load adaptive specialization code during run time to adapt to wireless network loss rate changes. Notice that in Figure 3, dynamism increases from left to right.



Lifetime-Dependent Middleware Specialization

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 compilation or 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 [Kiczales

et al. 1997], compiler flags, and precompiler directives [Klefstad et al. 2002]. QuO [Zinky et al. 1997] 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 [Object Management Group 2000], optional command-line parameters, for example, to set socket buffer-size, and configuration files such as ORBacus configuration file (`www.orbacus.com`).

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 [David et al. 2001] and Yang et. al [Yang et al. 2002], the component configurator pattern [Schmidt et al. 2000] used in DynamicTAO [Kon et al. 2000], and the virtual component pattern [Corsaro et al. 2002] 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 [Blair et al. 1998], late composition of components [Klefstad et al. 2002], and dynamic weaving of aspects [Yang et al. 2002].

2.3 Paradigms-Dependent Specialization

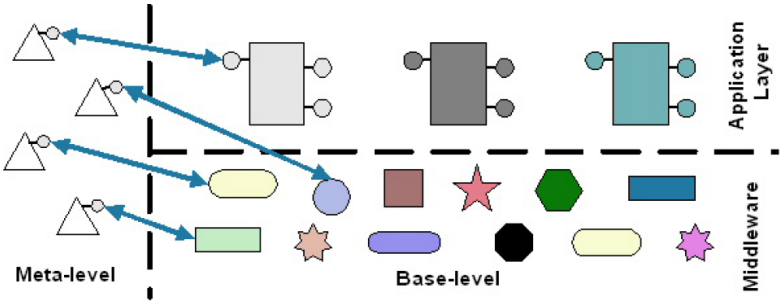
Numerous advances in programming paradigms have also contributed to middleware specialization techniques. Although many important contributions have been made in this area, a review of the literature shows that four paradigms, in addition to object-oriented paradigm, play key roles in supporting middleware specialization: computational reflection [Cacho and Batista 2005], component-based design [Szyperski 1997], aspect-oriented programming (AOP) [Kiczales et al. 1997], and feature-oriented programming (FOP) [Prehofer 1997].

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.

2.3.1 *Computational Reflection.* Computational reflection [Cacho and Batista 2005] 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. Thus computational reflection is an efficient and simple way of inserting new functionalities in a reflective middleware. A reflective middleware system is divided into two levels: a base-level and a self representative meta-level that is causally connected to the

system meaning that any modifications either to the system or to its representation are reflected in the other.

According to Figure 4, the middleware core is also represented by base-objects and new functionality is inserted by meta-objects. It also 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. Thus, it is necessary only to know components and interfaces. The next generation middleware [Blair et al. 1998; Fábio M. Costa and Gordon S. Blair 1999] exploits computational reflection to customize the middleware architecture. Reflection can be used to monitor the middleware internal (re)configuration [Roman et al. 2001].



Reflective Middleware

2.3.2 *Aspect Oriented Programming (AOP) Techniques.* Kiczales et al. [Kiczales et al. 1997] realized that complex programs are composed of different intervening crosscutting concerns (i.e., 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 [Kiczales et al. 1997] applies the principle of "separation of concerns" (SoC) [Parnas 1972] 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 [Sullivan 2001], 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. [Yang et al. 2002] and David et al. [David et al. 2001] 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`).

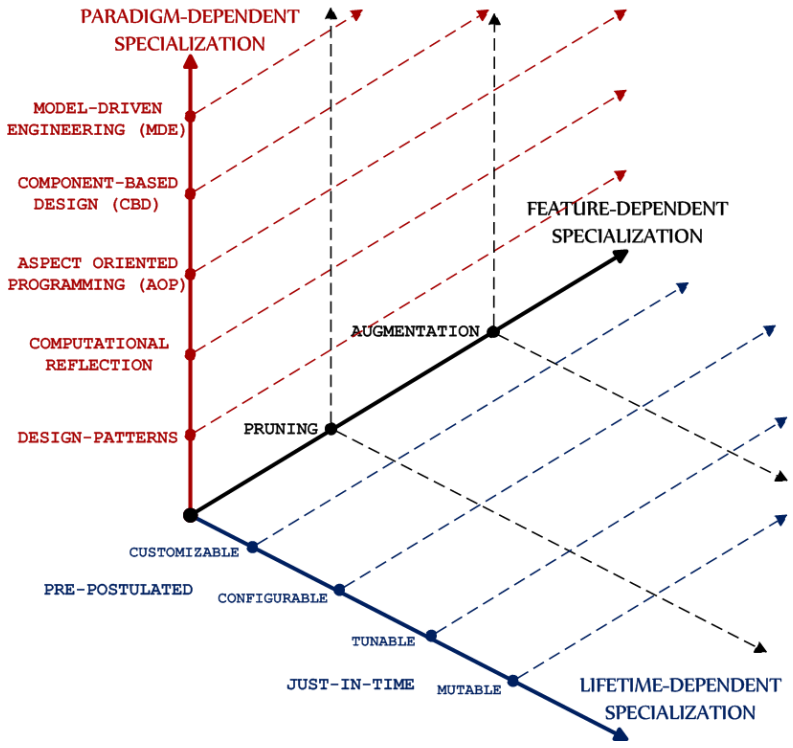
2.3.3 *Model-Driven Engineering (MDE).* MDE is a paradigm that integrates model-based software development techniques (including Model-Driven Development [Schmidt 2006] and the

OMG's Model Driven Architecture [Object Management Group 2001]) 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 inability to adapt in the face of functionality evolution and diversification. The insufficient module-level reusability stems from the non-modular interaction of the *"intrinsic"* and *"extrinsic"* properties in conventional middleware architectures. Conventional middleware architectures also lack effective means to reuse *"extrinsic"* properties, especially ones that are crosscutting [Kiczales et al. 1997] in nature, *i.e.*, not localized within modular boundaries as analyzed in [Zhang and Jacobsen 2004]. 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 properties 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). To tackle the aforementioned problems, Zhang et. al. [Zhang et al. 2005a] propose a new architectural paradigm called Modelware which embodies the *"multi-viewpoints"* [Nuseibeh et al. 1994] approach.

2.4 Specializations Taxonomy

Using these dimensions of specializations, we have developed a taxonomy for middleware specializations as shown in Figure 5.



Three Dimensional Taxonomy of Middleware Specialization Research

2.5 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.

2.5.1 *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 I: 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/correcting 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 [Trujillo et al. 2007]
AOP + Reflection	Composition based on application requirements	On-demand feature weaving	May cause conflicts	AspectOpenORB
AOP + MDE + FOP + Reflection	Design/Weave/Prune valid features combinations	Systematic, correct specialization process	Safe specializations is challenging	*Research Needed*

Table I 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 [Apel et al. 2008]. This weakness is the strength of aspects. Caesar [Mezini and Ostermann 2003], AFMs [Apel et al. 2008] 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) [Popovici et al. 2003]. JAsCo [Vanderperren et al. 2005] is an adaptive AOP language used to specialize Web Services implementations [Verheecke and Cibrn 2003] whereas PROSE [Nicoara et al. 2008] and Abacus [Zhang et al. 2005b] 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.

3. THE AUTOMATED MIDDLEWARE SPECIALIZATION PROCESS

The previous section realized a taxonomy for categorizing contemporary middleware specialization research. The taxonomy leads to insights in developing a multi-stage middleware specialization lifecycle process. This section presents the middleware specialization lifecycle and the resulting automated and generative framework for middleware specialization using feature-oriented, reverse engineering and, generative 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 section.

3.1 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 an automated, generic, reusable, extensible and systematic mechanism for refining existing, and accommodating new specializations, as well as accounting for different middleware platforms. This research has identified 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.

However, automating middleware specializations for DRE applications with stringent 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.

3.2 Key Requirements for an Automated Solution

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.

3.2.1 *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.

3.2.2 *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 [Marlet et al. 1999], 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, and state synchronization tasks in stateful group failover [Tambe et al. 2009]. 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 [Krishna et al. 2006].

3.2.3 *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.

3.2.4 *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 [Kiczales et al. 1997]. 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 [Kiczales et al. 1997].

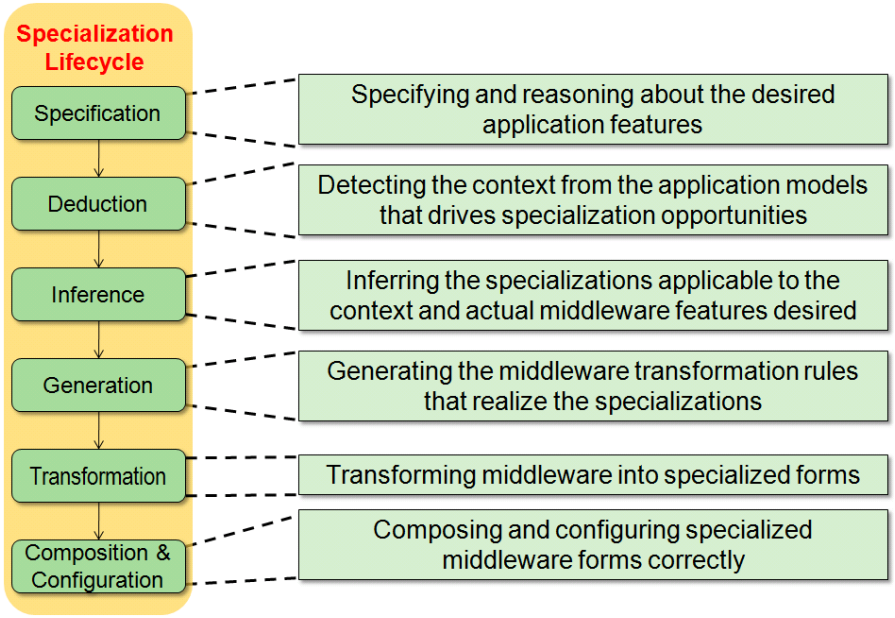
3.2.5 *Generating the Specialization Transformations.* Although the specialization points are determined, the exact nature of the transformation to be carried out at those points corresponding to the specialization patterns must be specified as a set of transformations, which we call *Specialization Advice.* Currently, these transformation rules are manually developed [Krishna et al. 2006] 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 may become problematic as the middleware frameworks and their respective sources evolve with changing application requirements.

3.2.6 *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.

3.3 Overview of the Middleware Specialization Lifecycle

Based on the specialization requirements described in the previous section, we have developed a systematic lifecycle for performing the middleware specializations. The related works surveyed address at a time only a subset of the specialization lifecycle stages. Therefore, there is a need to devise a systematic process methodology that addresses all these stages. Figure 6 illustrates and provides a brief overview of the middleware specialization process.

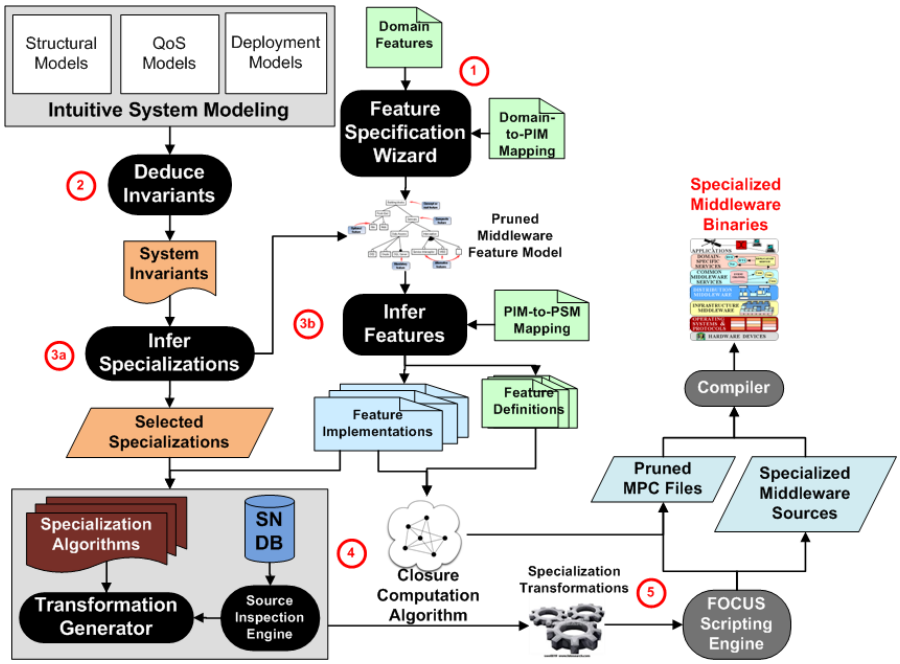


The Middleware Specialization Lifecycle

3.4 Automated Middleware Specialization Framework

In order to automate the middleware specialization process, we have developed a systematic, multi-staged, automated middleware specialization framework called AutoMaS. Figure 7 shows the AutoMaS middleware specialization framework and illustrates how the different stages of the middleware specialization lifecycle are addressed and automated by different framework software components.

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 platform-independent model (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



The Automated Middleware Specialization Framework

are also configured along the way as they are selected for composition. The 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 interpreters. Invariants are application properties which maybe structural or configuration or deployment that do not 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 platform-specific model (PSM) middleware features that implement the individual PIM features using the PIM-PSM mappings that are provided by the middleware developer.
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 hard coding 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) [Elliott 2007] 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 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.

4. FEATURE-ORIENTED REASONING TECHNIQUES TO DRIVE THE MIDDLEWARE SPECIALIZATIONS

The previous section realized a systematic and automated process for performing middleware specializations. 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 section addresses the first challenge outlined in Section 1.2 pertaining to feature-oriented reasoning to drive middleware specializations. We first list the challenges that are still unresolved. 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*.

4.1 Unresolved Challenges

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* [Marlet et al. 1999] from these high-level system models.

However, existing specialization techniques do not 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, read only operations, single interface operations that always get dispatched to the same server-side handlers, state synchronization tasks in stateful group failover [Tambe et al. 2009]. Such repetitive that can be potentially sped up by optimizing the underlying middleware through caching [Krishna et al. 2006], bypassing middleware layers [Demir et al. 2007], or fusion of layers [Krishna et al. 2006], 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

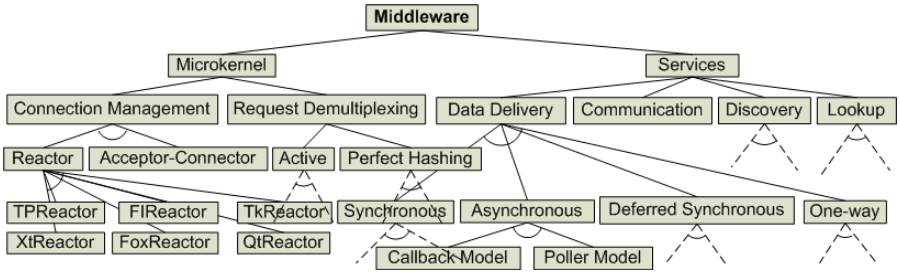
tedious task of identification of the middleware specialization context, the DRE system developer will still need to determine what specializations are applicable for a particular context. Current techniques of determining this mapping are still manual [Krishna et al. 2006].

4.2 Feature Oriented Reasoning

To address the outlined challenges, we describe our approach that uses feature-oriented reasoning.

4.2.1 *Feature Mapping Wizard.* In the development process, the automated specialization process' role is applicable in the packaging and assembly phases where the application variants along with the hosting middleware are 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 an application is supposed to perform, which includes dimensions of concurrency.

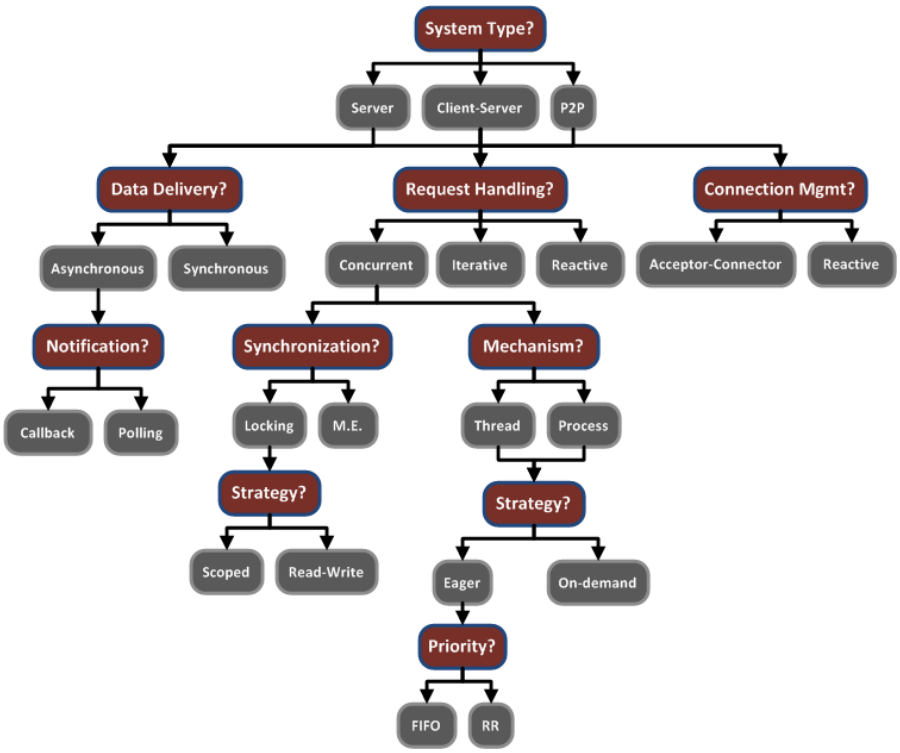
Normally, domain concerns and middleware features manifest themselves into separate hierarchial representations. Therefore, a mapping is required to transform domain concern hierarchies to middleware feature hierarchial models. In order to create a systematic mapping, this wizard makes use of model transformations 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.



Middleware PIM Feature Model

Feature models of the general-purpose middleware as shown in Figure 8 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 9 to ask different questions to the application developer to infer the application variant characteristics. These characteristics include distribution features, such as client/server, and 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 coarse-grained answer scopes down the application characteristics based upon which the next fine-grained questions are asked that configure the application behavior.



Decision Tree used by the Feature Mapper Wizard

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.

4.2.2 *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 [Balasubramanian et al. 2005] 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.

4.2.3 *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.

Table II: `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 III: Performance Optimization Principles [Varghese 2005]

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

The snippet of SP-KBASE knowledge base shown in Table II 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 are to be performed. It reports the incompatible set of specializations to the end-user or simply skips them if running in 'silent' mode.

5. AUTOMATED REALIZATION OF MIDDLEWARE SPECIALIZATIONS

The previous section 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 an 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 section addresses the second challenge outlined in Section 1.2 pertaining to automated realization of middleware specializations. First, a list of challenges that are still unresolved is presented. Next, a solution approach is presented that provides two automated techniques for generation of specialization transformation directives and for transformation of the middleware build configurations are presented.

5.1 Unresolved Challenges

5.1.1 *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:

1. ***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 [Institute for Software Integrated Systems sitya]. 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, then 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 must then 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.
2. ***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 script-

ing 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.

3. ***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 the 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 the 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.

5.1.2 *Lack of middleware support for domain-specific recovery semantics.* General purpose middleware have limitations in how many diverse *domain-specific* semantics can they readily support *out-of-the-box*. Since different application domains may impose different variations in fault tolerance (or for that matter, other forms of quality of service) requirements, these semantics cannot be supported out-of-the-box in general-purpose middleware since they are developed with an aim to be broadly applicable to a wide range of domains. Developing a proprietary middleware solution for each application domain is not a viable 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 the 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 be realized while leveraging low cost, general-purpose middleware without permanently modifying it? An approach based on aspect-oriented programming (AOP) [Kiczales et al. 1997] 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 [Mohapatra et al. 2005; **?**], 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* [Czarnecki and Eisenecker 2000] offers a promising choice to address this question.

5.2 Automated Realization of Middleware Specializations

We now describe our approach to overcome the outlined challenges.

5.2.1 *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 III) 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 II. 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 [Dabholkar and Gokhale 2011], 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.

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.

5.2.2 *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 [Gokhale et al. 2007]. 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 [openArchitectureWare 2007]. Another way is to annotate the framework and pattern source code to identify the specialization points and write source-to-source transformations (S2S) [Krishna et al. 2006]. 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*
return 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 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 we have not shown in this paper 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 [Krishna

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
return 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 for
return M_s
end

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 for
return M_s
end

et al. 2006] whose Perl scripts execute the transformations on the sources and subsequently the build specialization tools generate the specialized middleware source build configurations.

5.2.3 *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. 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.

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 [Developer 2007] 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 ∈ F_p
4: C_f : Closure set for feature f ∈ F_p
5: C_s : Closure set for source hint s ∈ 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 := ∅
11: for each feature f ∈ F_p do
12:     s := FIND feature definition from M_s for feature f
13:     C_f := ∅
14:     C_s := ∅
15:     C_s := COMPUTE closure for feature definition s
16:     C_f := C_f ∪ 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 := ∅
20:         C_s := COMPUTE closure for feature implementation file i ∈ P_i
21:         C_f := C_f ∪ C_s
22:         P_i := P_i ∪ FIND new feature definition & implementation files that were found in the closure computation
23:     end while
24:     C_p := C_p ∪ C_f
25: end for
26: return C_p
27: 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 5.2.3 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.

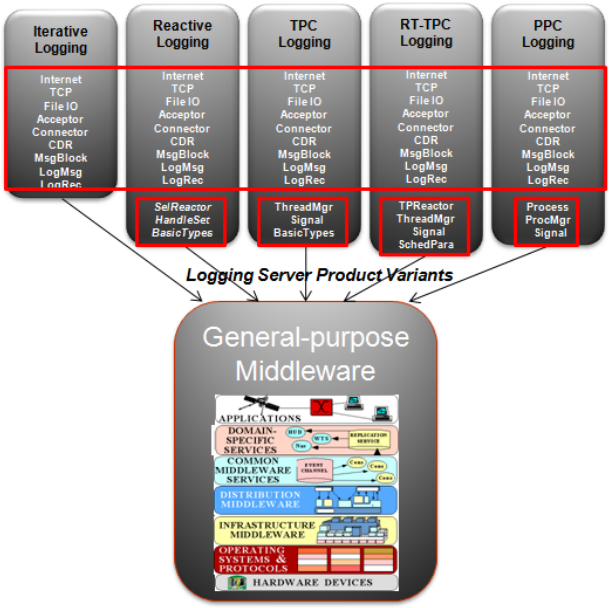
5.2.4 *Middleware Composition Synthesis through Build Specialization.* Different middleware use sophisticated techniques to compile its source code into shared libraries. Some of these techniques rely on straightforward scripting *e.g.*, shell script, batch files, Perl scripts, or ANT scripts while some of them rely on descriptor files such as make file system or advanced cross-compiler build facilities like MPC (Make Project Creator) [Elliott 2007]. 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.

6. EVALUATING THE AUTOMAS MIDDLEWARE SPECIALIZATION PROCESS

6.1 Logging Server Case Study



Logging Application Variant

In order to explain and evaluate the middleware specialization process, we use a motivating example of a application variant of networked logging servers as shown in Figure 10. 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 enterprise, 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. Enterprise 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 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 AutoMAS by modeling a application variant of networked logging applications based on contemporary, widely used communication middleware such as ACE [Institute for Software Integrated Systems sityb]. ACE is a free, open-source, platform-independent, highly con-

figurable, 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 10 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.

6.2 Evaluation of the Closure Computation Algorithm

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

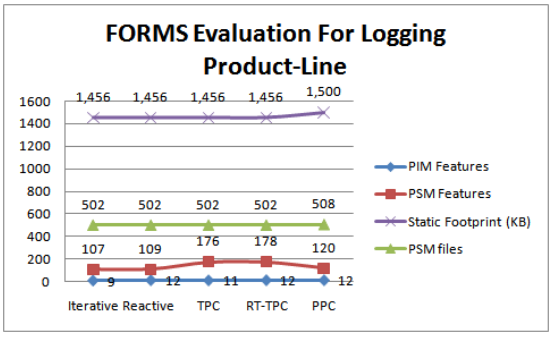
Networked Logging Applications Application Variant		**Outcome of Closure Computations**		**Synthesized Middleware**
application variant (described in Domain Concerns)	***# of Middleware PIM Features***	***# of Middleware PSM Features***	***Size of Closure Set (PSM files)***	***Static Footprint (KB)***
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 the 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.

6.2.1 *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 IV 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 IV also shows that the 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.

6.2.2 *Modularization Discrepancies.* On the other hand as shown in Figure 11, there is a wide disparity between the number of PSM middleware features required by the individual product



Modularization Disparities

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. 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.

6.3 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 6.2.2, 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.

6.4 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.

6.5 Evaluation of the Generative Middleware Specialization Algorithms

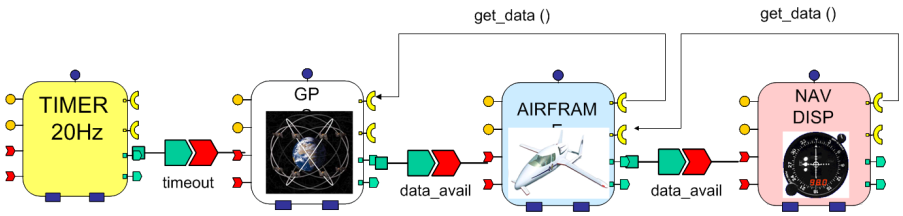
Since middleware specialization is a software engineering process, we demonstrate its applicability and evaluate its merits along the following dimensions: (1) We first show how the algorithms can be applied to specialize middleware for a representative DRE system case study; (2) We show the savings 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.

6.6 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 II.

6.6.1 *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 [Sharp and Roll 2003], which is a component-based, publish/subscribe platform built atop the TAO Real-time CORBA Object Request Broker (ORB) [Schmidt et al. 2002]. Figure 12 illustrates the *BasicSP* application scenario, which is an assembly of avionics mission computing components reused in different Bold Stroke product variants.



The Basic Single Processor (*BasicSP*) Application Scenario

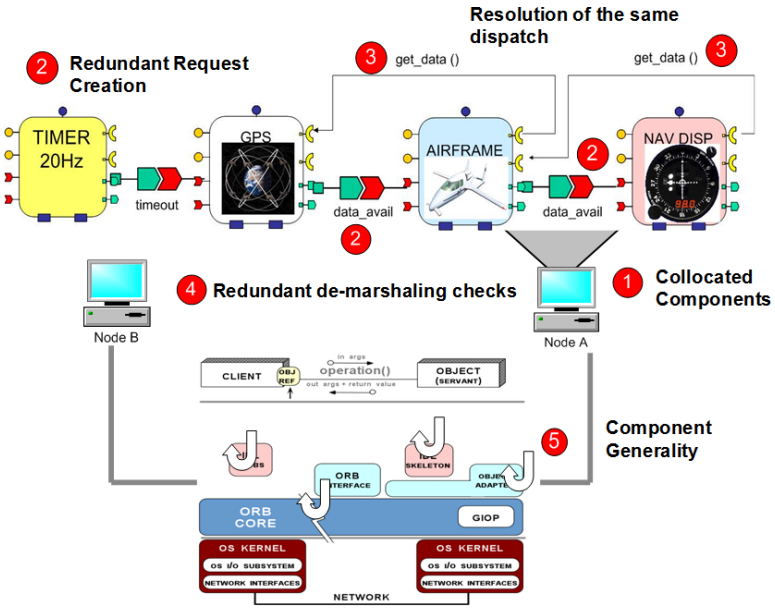
BasicSP involves four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays at a rate that is configurable. The time to process inputs to the system and present output to cockpit displays should thus be less than the rate, which as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

6.6.2 *Applying Generative Specializations to Specialize Middleware for BasicSP.* We show how the the model interpreters traverse the BasicSP model to realize the specializations.

Applying Step 1 *(Deducing the Context).* Figure 13 showcases the different application invariants that can be deduced.



The Basic Single Processor (*BasicSP*) Specialization Points

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 (endianess) 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 II.

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 [Schmidt et al. 2000] 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 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 5.2.1. 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.

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>
```

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.

Table V: 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 %

6.6.3 *Improvements in Developer Productivity through Auto-Generation.* We leverage FOCUS [Krishna et al. 2006] 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 5.2.2 alleviates the burden on the developers as well as makes them easy to extend and maintain. Table V 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.

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.

6.6.4 *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 VI 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 VI: 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%

7. CONCLUSIONS

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 paper, we presented an automatic, systematic and reusable process for specializing general-purpose middleware that enables the vertical decomposition of middleware along the domain concerns by deducing the invariant properties, inferring the specializations and generating the transformations required to specialize middleware sources. These specializations are based upon a comprehensive taxonomy for specializations that we developed based on a survey of the literature. Our AutoMAS approach is realized within the FORMS and GeMS tools. We also provided detailed evaluation of the process by quantifying the developer productivity improvements and reduction in latency, response time and memory footprint of the resulting specialized middleware.

7.1 Discussion and Lessons Learned

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 I 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.

7.2 Guidelines for Middleware Specialization

We now provide guidelines for middleware specialization based on our taxonomy that practitioners can adopt for their usecases. 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 run-time 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.

REFERENCES

Agha, G. A. 2002. Introduction. *Communications of the ACM 45,* 6, 30–32.

Apel, S., Leich, T., and Saake, G. 2008. Aspectual feature modules. *Software Engineering, IEEE Transactions on 34,* 2 (March-April), 162–180.

Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., and Schmidt, D. C. 2005. 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.* IEEE Computer Society, Washington, DC, USA, 190–199.

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

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

Cacho, N. and Batista, T. V. 2005. Using AOP to Customize a Reflective Middleware. In *OTM Conferences (2)*. Lecture Notes in Computer Science, vol. 3761. Springer, 1133–1150.

Chakravarthy, V., Regehr, J., and Eide, E. 2008. Edicts: Implementing Features with Flexible Binding Times. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA, 108–119.

Corsaro, A., Schmidt, D. C., Klefstad, R., and O'Ryan, C. 2002. 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.

Czarnecki, K. and Eisenecker, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts.

Dabholkar, A. and Gokhale, A. 2011. 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.

David, P.-C., Ledoux, T., and Bouraqadi-Saadani, N. M. 2001. Two-step Weaving with Reflection using AspectJ. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.

Demir, Ö. E., Devanbu, P. T., Wohlstadter, E., and Tai, S. 2007. An aspect-oriented approach to bypassing middleware layers. In *AOSD*, B. M. Barry and O. de Moor, Eds. ACM International Conference Proceeding Series, vol. 208. ACM, 25–35.

Developer, B. 2007. The source-navigator^{TM} ide. `http://sourcenav.sourceforge.net/`.

Elliott, C. 2007. The makefile, project, and workspace creator (mpc). `www.ociweb.com/products/mpc`.

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

Gokhale, A., Kaul, D., Kogekar, A., Gray, J., and Gokhale, S. 2007. POSAML: A Visual Modeling Language for Managing Variability in Middleware Provisioning. *Elsevier Journal of Visual Languages and Computing (JVLC) 2007 18,* 4, 359–377.

Gottlob, G., Schrefl, M., and Röck, B. 1996. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst. 14,* 3, 268–296.

Hunleth, F. and Cytron, R. K. 2002. 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)*. ACM Press, Berlin, Germany, 38–45.

Institute for Software Integrated Systems. Vanderbilt Universitya. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO.

Institute for Software Integrated Systems. Vanderbilt Universityb. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/.

Jin, J. and Nahrstedt, K. 2004. On Exploring Performance Optimizations in Web Service Composition. In *Middleware*. 115–134.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. 1997. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. 220–242.

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

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

Krishna, A., Gokhale, A., Schmidt, D. C., Hatcliff, J., and Ranganath, V. 2006. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*. Leuven, Belgium, 205–218.

Lohmann, D., Spinczyk, O., and Schröder-Preikschat, W. 2006. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. *Transactions on AOSD II 4242*, 227–255.

Marlet, R., Thibault, S., and Consel, C. 1999. Efficient Implementations of Software Architectures via Partial Evaluation. *Automated Software Engineering: An International Journal 6,* 4 (October), 411–440.

Mezini, M. and Ostermann, K. 2003. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM, New York, NY, USA, 90–99.

Mezinia, M. and Ostermann, K. 2004. Variability Management with Feature-oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes 29,* 6, 127–136.

Mohapatra, S., Cornea, R., Oh, H., Lee, K., Kim, M., Dutt, N. D., Gupta, R., Nicolau, A., Shukla, S. K., and Venkatasubramanian, N. 2005. A Cross-Layer Approach for Power-Performance Optimization in Distributed Mobile Systems. In *Proceedings of International Parallel and Distributed Processing Symposium.*

Nicoara, A., Alonso, G., and Roscoe, T. 2008. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Oper. Syst. Rev. 42,* 4, 233–246.

Nuseibeh, B., Kramer, J., and Finkelstein, A. 1994. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng. 20,* 10, 760–773.

Object Management Group 2000. *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 ed. Object Management Group.

Object Management Group 2001. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 ed. Object Management Group.

Object Management Group 2005. *Real-time CORBA Specification*, 1.2 ed. Object Management Group.

Ömer Erdem Demir, Dévanbu, P., Wohlstadter, E., and Tai, S. 2007. An Aspect-oriented Approach to Bypassing Middleware Layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development.* ACM Press, Vancouver, British Columbia, Canada, 25–35.

openArchitectureWare. 2007. openArchitectureWare. `www.openarchitectureware.org`.

Parnas, D. L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM 15,* 12 (Dec.).

Popovici, A., Alonso, G., and Gross, T. 2003. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development.* Boston, Massachusetts, 100–109.

Prehofer, C. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP'97—Object-Oriented Programming, 11th European Conference*, M. Aksit and S. Matsuoka, Eds. Vol. 1241. Springer, Jyväskylä, Finland, 419–443.

Roman, M., Campbell, R. H., and Kon, F. 2001. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online 2,* 5 (July).

Sadjadi, S., McKinley, P., and Kasten, E. 2003. Architecture and operation of an adaptable communication substrate.

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

Schmidt, D. C. 2006. Model-Driven Engineering. *IEEE Computer 39,* 2, 25–31.

Schmidt, D. C., Natarajan, B., Gokhale, A., Wang, N., and Gill, C. 2002. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online 3,* 2 (Feb.).

Schmidt, D. C., Schantz, R., Masters, M., Cross, J., Sharp, D., and DiPalma, L. 2001. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. In *CrossTalk - The Journal of Defense Software Engineering.* Software Technology Support Center, Hill AFB, Utah, USA, 10–16.

Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York.

Sharp, D. C. and Roll, W. C. 2003. Model-Based Integration of Reusable Component-Based Avionics System. Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003.

Sullivan, G. T. 2001. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM 44,* 10, 95–97.

Suri, D., Howell, A., Shankaran, N., Kinnebrew, J., Otte, W., Schmidt, D. C., and Biswas, G. 2006. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference.* College Park, MD.

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional.

Tambe, S., Dabholkar, A., and Gokhale, A. 2009. 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).* IEEE Computer Society, Tokyo, Japan.

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

Trujillo, S., Batory, D., and Diaz, O. 2007. Feature oriented model driven development: A case study for portlets. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, Washington, DC, USA, 44–53.

Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M. A., and Jonckers, V. 2005. Adaptive Programming in JAsCo. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development.* Chicago, Illinois, 75–86.

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

Verheecke, B. and Cibrn, M. A. 2003. Aop for dynamic configuration and management of web services. In *In Proceedings of 2003 International Conference on Web Services*. 2004.

Wohlstadler, E., Jackson, S., and Devanbu, P. 2003. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems . In *Proceedings of the International Conference on Software Engineering*. Portland, OR.

Yang, Z., Cheng, B. H. C., Stirewalt, R. E. K., Sowell, J., Sadjadi, S. M., and McKinley, P. K. 2002. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*. ACM, New York, NY, USA, 85–92.

Zhang, C., Gao, D., and Jacobsen, H.-A. 2005a. Generic Middleware Substrate Through Modelware. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*. Grenoble, France, 314–333.

Zhang, C., Gao, D., and Jacobsen, H.-A. 2005b. Towards Just-in-time Middleware Architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM Press, Chicago, Illinois, 63–74.

Zhang, C. and Jacobsen, H.-A. 2004. 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*. ACM, New York, NY, USA, 188–205.

Zinky, J. A., Bakken, D. E., and Schantz, R. 1997. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems 3,* 1, 1–20.

Dr. Akshay Dabholkar is currently a Principle Member of Technical Staff at Oracle Corporation, Santa Clara, CA, USA. Prior to that he was a Distributed Systems Engineer at Nimbula, Mountain View, CA, USA. Dr. Dabholkar obtained his PhD in Computer Science from Vanderbilt University in April 2012.

Dr. Aniruddha Gokhale is an Associate Professor in the Department of Electrical Engineering and Computer Science, and Senior Research Scientist at the Institute for Software Integrated Systems both at Vanderbilt University, Nashville, TN, USA. His current research focuses on developing novel solutions to emerging challenges in mobile cloud computing, real-time stream processing, publish/subscribe systems, and cyber physical systems. He is also working on using cloud computing technologies for STEM education. Dr. Gokhale obtained his B.E (Computer Engineering) from University of Pune, India, 1989; MS (Computer Science) from Arizona State University, 1992; and D.Sc (Computer Science) from Washington University in St. Louis, 1998. Prior to joining Vanderbilt, Dr. Gokhale was a member of technical staff at Lucent Bell Laboratories, NJ. Dr. Gokhale is a Senior member of both IEEE and ACM, and a member of ASEE. His research has been funded in the past by DARPA, US DoD and NSF including a NSF CAREER award.