Assessing Contemporary Modularization Techniques for Middleware Specialization

Akshay Dabholkar
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
aky@dre.vanderbilt.edu

Aniruddha Gokhale
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
gokhale@dre.vanderbilt.edu

ABSTRACT

Middleware specialization is a technique to prune middleware features that are deemed unnecessary by the application domain, and to optimize and customize the relevant features to obtain domain-specific semantics within the middleware. Although contemporary modularization techniques, such as aspect-oriented programming (AOP) and feature-oriented programming (FOP), have been used in middleware specialization, there is a lack of a taxonomy that can assess the strengths and weaknesses of these techniques. To address these limitations, this paper develops a taxonomy that organizes contemporary modularization approaches applied to the problem of middleware specialization within a unified framework. The taxonomy helps assess the applicability of multiple modularization techniques used in concert for specializing system software such as middleware.

Categories and Subject Descriptors

H.4 [**Middleware**]: Specialization; D.2.8 [**Software Engineering**]: [complexity measures, performance measures]

1. INTRODUCTION

A number of applications based on distributed, general-purpose middleware platforms, such as J2EE/EJB, .NET Web Services and CORBA, must specialize these middleware to satisfy their functional and quality of service (QoS) requirements. Middleware specialization is a technique to prune middleware features that are deemed unnecessary by the application, and to optimize and customize the relevant features to obtain domain-specific semantics within the middleware.

Many prior research efforts in middleware specialization have used different modularization techniques, such as Aspect-oriented Programming (AOP) [10] and Feature-oriented Programming (FOP)-[23]. For example, AOP is used to develop resource-efficient system software [14] and context-specific specilized middleware [13], to bypass middleware layers [20], to define fine-grained middleware architectures that can be seamlessly refined [8], and even in dynamic specialization [4]. Modularization techniques have also been used in concert [35] with model-driven development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACoM OOPSLA '08 Nashville, TN, USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Although different modularization techniques have shown how middleware can be specialized, there is a general lack of a taxonomy that helps to assess the strengths and weaknesses of modularization techniques when used individually or in concert. To address these limitations, this paper develops such a taxonomy which is derived from a survey of many research efforts, and is broadly classified along three dimensions of application development: feature-dependent, paradigm-dependent and lifetime-dependent. This taxonomy assesses the applicability of a specific technique for a particular context in middleware specialization. Such a taxonomy can also provide guidelines in specializing other kinds of systems software including operating systems and databases.

To effectively present the taxonomy and assessment of contemporary modularization techniques, we have organized the remainder of this paper as follows: Section 2 proposes our three-dimensional taxonomy of middleware specialization techniques. Section 3 brings forth a brief discussion on the different middleware specialization techniques and provides an assessment of their combinations through qualitative evaluations and guidelines for applying them. Finally, Section 4 concludes the paper and suggests possible future research directions.

2. TAXONOMY OF MODULARIZATION TECHNIQUES FOR MIDDLEWARE SPECIALIZATION

In this section we develop a taxonomy for assessing the different modularization techniques used in middleware specialization. We surveyed multiple research efforts on middleware specialization that use different modularization techniques, which in turn can be categorized with respect to the type of specialization it provides.

2.1 Survey of Modularization techniques used in Middleware Specialization

Lohmann et. al. [14] argue that AOP is well suited for the development of fine-grained and resource-efficient system software product lines where overhead due to the dynamic binding and dispatch of object-orientation is not acceptable when aspects beat objects. *FACET* [8] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. The advantage of using AOP is that the hooks and callbacks that were needed using standard object oriented techniques for adding functionality to existing code are no longer required. This removes the need to preconceive where the variation points of the code are needed and also removes the need to refactor large amounts of existing code to insert these hooks after the fact leading to better modularization.

Modelware [35] advocates the use of models and role-based as-

pect views and aspect libraries to separate intrinsic functionalities of middleware (*i.e.,* a set of coherent components free of crosscutting concerns) from extrinsic ones (*i.e.,* domain variations). This separation effectively lowers the concern density per component and fosters the coherence and the reuse of the components of middleware architectures. Devanbu et. al. [20] adopted AOP to enable bypassing layers of middleware to avoid the rigid layer processing performed by middleware that can lower overall system throughput, and reduce availability and/or increase vulnerability to security attacks. Their focus is to enable developers of middleware-based applications to conveniently adopt a *bypassing design pattern* (or *bypassing style*) (with good tool and run-time support) to speed-up applications, without having to write intricate, low-level, inherently non-portable code.

AOP, however, requires the specification of the middleware functionality which implies a broader knowledge of the structures and the application functionality. To support the runtime identification of the application needs and the *dynamic specialization* of the middleware according to the application requirements, *Aspect Open-Orb* [4] uses AOP to customize the reflective middleware. Aspects that are not in the application code can be dynamically inserted using the meta-object protocol of computational reflection.

FOCUS [13] describes how context-specific specializations can be automated and applied to optimize excessive generality in general-purpose middleware used for product-line architectures thereby improving throughput, average- and worst-case end-to-end latencies and predictability without affecting portability, APIs, or application software implementations while preserving interoperability.

The survey presented above enables us to develop a taxonomy as shown in Figure 1. The taxonomy can be broadly classified along three dimensions of application development: (1) feature-dependent, (2) paradigm-dependent, and (3) lifetime-dependent. The remainder of this section delves into the details of each dimension.

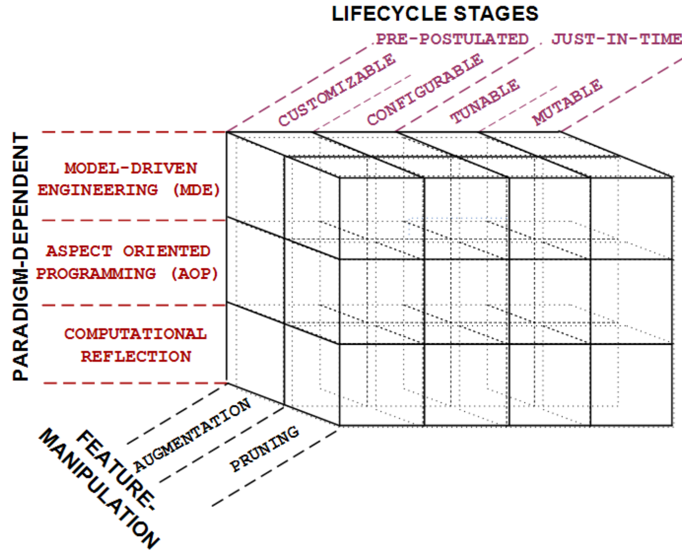

Figure 1: Three Dimensional Taxonomy of Middleware Specialization Research

2.2 Feature-Dependent Specialization

Feature-oriented programming (FOP) captures the variants of a base behavior though a layer of encapsulation of multiple abstractions and their respective increments that together pertain to the definition of a feature [16]. 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:

2.2.1 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 [13] where unnecessary features are automatically removed from general purpose middleware through techniques such as memoization to provide optimizations for DRE systems.

2.2.2 Feature Augmentation

Feature augmentation is a strategy applied when the specialization is grounded via the insertion of new features, either because the original middleware did not support it or the middleware is composed out of building blocks [1, 3, 30]. The latter variety of middleware platforms are designed to overcome the limitations of monolithic architectures. Their goal is to offer a small core and to use computational reflection to augment new functionalities.

In Section 2.4.2, AOP can be used to compose middleware platforms where the middleware core contains only the basic functionalities [8, 35]. 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.

2.3 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* [36]. Figure 2 shows this dimension of our taxonomy. If middleware specialization is performed during the application compile or startup time, we designate it *pre-postulated/static specialization*. For example, EmbeddedJava (`java.sun.com/products/em\-beddedjava`) minimizes the footprint of embedded applications during the application compile time. Similarly, if the middleware specialization is performed during the application run time, we designate it *just-in-time/dynamic specialization*. For example, MetaSockets [25] load adaptive specialization code during run time to adapt to wireless network loss rate changes. Notice that in Figure 2, dynamism increases from left to right.

2.3.1 Pre-postulated Specialization

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

- **Customizable specialization** enables adapting the middleware during the application compile/link-time so that a developer can generate specialized (adapted) versions of the ap-

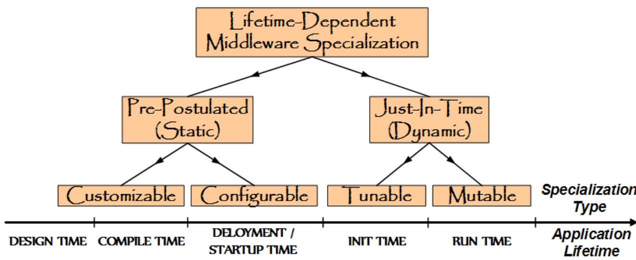

Figure 2: Lifetime-Dependent Middleware Specialization

plication. 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 [10], compiler flags, and precompiler directives [11]. QuO [38] 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 [19], optional command-line parameters, for example, to set socket buffer-size, and configuration files such as ORBacus configuration file (`www.orbacus.com`).

2.3.2 *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 [6] and Yang et. al [34], the component configurator pattern [27] used in DynamicTAO [12], and the virtual component pattern [5] 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 [3], late composition of components [11], and dynamic weaving of aspects [34].

2.4 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 [4], component-based design [29], aspect-oriented programming [10], and feature-oriented programming [23].

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.

2.4.1 *Computational Reflection*

Computational reflection [4] 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. As depicted in Figure 3, a reflective system (represented as base-level objects) has a self representation (represented as meta-level objects) that is causally connected to the system meaning that any modifications either to the system or to its representation are reflected in the other.

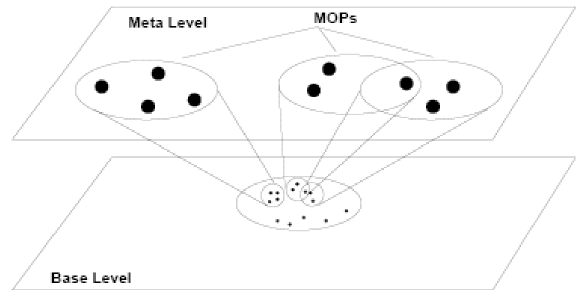

Figure 3: A Reflective System with Causally Connected Meta-level

The base-level part of a system deals with the *normal* (functional) aspects of the system whereas the meta-level part deals with the computation (implementation) aspects of the system. The meta-level contains the building blocks responsible for supporting reflection. The elements of the base-level and that of the meta-level are, respectively, represented by base-level objects and meta-level objects. A meta-object protocol (MOP) [9] is a meta-level interface that enables *systematic* (as opposed to ad hoc) inspection and modification of the base-level objects and abstraction of the implementation details.

Computational reflection is an efficient and simple way of inserting new functionalities in a reflective middleware. Thus, it is necessary only to know components and interfaces. The next generation middleware [3, 7] exploits computational reflection to customize the middleware architecture. Reflection can be used to monitor the middleware internal (re)configuration [24]. The middleware is divided in two levels: base-level and meta-level. According to Figure 3, the middleware core is also represented by base-objects and new functionality is inserted by meta-objects. Figure 4 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.

2.4.2 *Aspect Oriented Programming (AOP) Techniques*

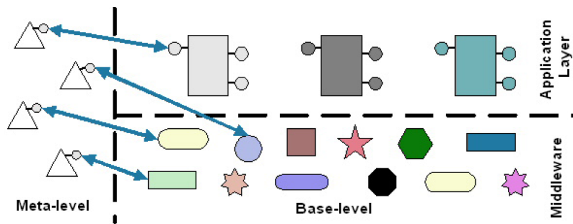

Figure 4: Reflective Middleware

Kiczales et al. [10] realized that complex programs are composed of different intervened crosscutting concerns (properties or areas of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are still scattered among different classes thereby complicating the development and maintenance of applications.

AOP [10] applies the principle of "separation of concerns" (SoC) [21] 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 [28], 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. [34] and David et al. [6] 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`).

2.4.3 *Model-Driven Engineering (MDE)*

MDE is an emerging paradigm that integrates model-based software development techniques (including Model-Driven Development [26] and the OMG's Model Driven Architecture) with QoS-enabled component middleware to help resolve key software development and validation challenges encountered by developers of large-scale distributed, real-time and embedded (DRE) middleware and applications. In particular, MDE tools can be used to specify requirements, compose DRE applications and their supporting infrastructure from the appropriate set of middleware components, synthesize the metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. These activities can be performed in a cyclic fashion until the QoS constraints are satisfied end-to-end.

Conventional middleware architectures suffer from insufficient module-level reusability and the ability to adapt in the face of functionality evolution and diversification. As reported in [35], *"intrinsic"* and *"extrinsic"* properties interact non-modularly in conventional middleware architectures. Consequently, middleware architects are faced with immense architectural complexities because the concern density per module is high. The code-level reusability of the *"common abstractions"* is also drastically reduced because the generality of intrinsic components is restricted by the *"extrinsic"* properties in the face of domain variations. A contributing factor to this complexity, is that the code-level design reusability in conventional middleware architectures is incapable of adequately dealing with *"change"* in two dimensions: time (functional evolution) and space (functional diversification).

The reusability in conventionally developed software components is insufficient due to the lack of explicit means to effectively distinguish intrinsic and extrinsic architectural elements. Conventional middleware architectures also lack effective means to reuse *"extrinsic"* properties, especially ones that are crosscutting [10] in nature, *i.e.*, not localized within modular boundaries. Conventional architectures have fallen short of doing so because they are incapable of componentizing and reusing crosscutting concerns as analyzed in [37]. Being able to componentize and to reuse these functionalities tremendously facilitates the construction of middleware systems. To tackle the aforementioned problems, Zhang et. al. [35] propose a new architectural paradigm called Modelware which embodies the *"multi-viewpoints"* [18] approach.

3. ASSESSMENT OF MODULARIZATION TECHNIQUES FOR MIDDLEWARE SPECIALIZATION

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

3.1 Qualitative Evaluation of the Middleware Specialization Taxonomy

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

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

1. **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 [2]. This weakness is the strength of aspects. Caesar [15], AFMs [2] 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.

Table 1: Evaluation of the Combinations of Dimensions

COMBINATIONS	USE CASES	STRENGTHS	WEAKNESSES	RELATED WORK
Pre-postulated + AOP	Weave/Prune at compile-time	Transparency without affecting core	Code Bloating	FACET, CLA, FOCUS, Bypassing Layers, AspectOpenORB
Pre-postulated + MDE	Weave/Prune only known features	Elegant design	Runtime specializations not possible	DTO, CLA, Modelware
Pre-postulated + Reflection	Inspect target platform features	Useful during deployment	Difficult to predict runtime conditions	AspectOpenORB, DTO
Just-in-time + AOP	Dynamic weaving of features	Dynamic Adaptation	Requires native platform support	JAsCo, PROSE, Abacus
Just-in-time + MDE	Self-healing/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 [31]
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*

2. **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) [22]. JAsCo [32] is an adaptive AOP language used to specialize Web Services implementations [33] whereas PROSE [17] and Abacus [36] 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 runtime. 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.

3.2 Guidelines for Middleware Specialization

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

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.

7. **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.

4. CONCLUDING REMARKS

Prior research has shown the usefulness of different modularization techniques to handle middleware specialization challenges. Yet there does not exist a common vocabulary that unifies these efforts. This paper addresses this challenge by developing a three-dimensional taxonomy for middleware specialization. We use this taxonomy to provide a qualitative assessment of the strengths and weaknesses of the modularization techniques. We also provide guidelines for application or middleware developers who are interested in specializing the middleware on how best to use this taxonomy in their project.

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 which can be assimilated into a catalog of specialization patterns 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. Moreover, validating the safety of specialization approaches is hard. Our current work focuses on this dimension of the research.

5. REFERENCES

[1] Gul A. Agha. Introduction. *Communications of the ACM*, 45(6):30–32, 2002.

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

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

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

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

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

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

[8] Frank Hunleth and Ron K. Cytron. Footprint and Feature Management Using Aspect-oriented Programming Techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 02)*, pages 38–45. ACM Press, 2002.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

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

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

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

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

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

[16] Mira Mezinia and Klaus Ostermann. Variability Management with Feature-oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[17] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Oper. Syst. Rev.*, 42(4):233–246, 2008.

[18] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994.

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

April 2000.

[20] Ömer Erdem Demir, Prémkumar Dévanbu, Eric Wohlstadter, and Stefan Tai. An Aspect-oriented Approach to Bypassing Middleware Layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, New York, NY, USA, 2007. ACM Press.

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

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

[23] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

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

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

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

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

[28] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001.

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

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

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

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

[33] Bart Verheecke and MarÃ■a Agustina CibrÃąn. Aop for dynamic configuration and management of web services. In *In Proceedings of 2003 International Conference on Web Services*, page 2004, 2003.

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

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

[36] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards Just-in-time Middleware Architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2005. ACM Press.

[37] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205, New York, NY, USA, 2004. ACM.

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