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 specialized 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.

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. FACT [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 functionality which implies a broader knowledge of the structures and non-portable code.

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:

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. **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/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 [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-
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.

2.4.2 Aspect Oriented Programming (AOP) Techniques

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.
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 runtime, 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

<table>
<thead>
<tr>
<th>COMBINATIONS</th>
<th>USE CASES</th>
<th>STRENGTHS</th>
<th>WEAKNESSES</th>
<th>RELATED WORK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-postulated + AOP</td>
<td>Weave/Prune at compile-time</td>
<td>Transparency without affecting core</td>
<td>Code Bloating</td>
<td>FACET, CLA, FOCUS, Bypassing Layers, AspectOpenORB</td>
</tr>
<tr>
<td>Pre-postulated + MDE</td>
<td>Weave/Prune only known features</td>
<td>Elegant design</td>
<td>Runtime specializations not possible</td>
<td>DTO, CLA, Modelware</td>
</tr>
<tr>
<td>Pre-postulated + AOP</td>
<td>Inspect target platform features</td>
<td>Useful during deployment</td>
<td>Difficult to predict runtime conditions</td>
<td>AspectOpenORB, DTO</td>
</tr>
<tr>
<td>Just-in-time + AOP</td>
<td>Dynamic weaving of features</td>
<td>Dynamic Adaptation</td>
<td>Requires native platform support</td>
<td>JAsCo, PROSE, Abacus</td>
</tr>
<tr>
<td>Just-in-time + MDE</td>
<td>Self-healing/correcting systems</td>
<td>Validation of Specializations</td>
<td>Incur runtime overhead</td>
<td>Models@Runtime</td>
</tr>
<tr>
<td>Just-in-time + Reflection</td>
<td>Introspect runtime application features</td>
<td>Dynamic Adaptation &amp; reconfiguration</td>
<td>Can cause unpredictable behavior</td>
<td>AspectOpenORB</td>
</tr>
<tr>
<td>AOP + FOP</td>
<td>ISD and SPLs</td>
<td>Better modularization of crosscutting features</td>
<td>Runtime specializations not possible, cause conflicts</td>
<td>AFMs, Caesar</td>
</tr>
<tr>
<td>FOP + MDE</td>
<td>SPLs</td>
<td>Better composition of features</td>
<td>Run-time specializations not possible, cause conflicts</td>
<td>FOMDD [31]</td>
</tr>
<tr>
<td>AOP + Reflection</td>
<td>Composition based on application requirements</td>
<td>On-demand feature weaving</td>
<td>May cause conflicts</td>
<td>AspectOpenORB</td>
</tr>
<tr>
<td>AOP + MDE + FOP + Reflection</td>
<td>Design/Weave/Prune valid features combinations</td>
<td>Systematic, correct specialization process</td>
<td>Safe specializations is challenging</td>
<td>Research Needed</td>
</tr>
</tbody>
</table>

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 pre-determined system feature requirements. However, it can configure dynamic augmentation or pruning of features at runtime. Recently models at run-time have 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-
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**


April 2000.


