Middleware Specialization for Product-Lines using Feature-Oriented Reverse Engineering

Akshay Dabholkar and Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville

Abstract

Supporting the varied software feature requirements of multiple variants of a software product-line while promoting reuse forces product line engineers to use general-purpose, feature-rich middleware. However, each product variant now incurs memory footprint and performance overhead due to the feature-richness in addition to the increased cost of its testing and maintenance. To address this tension, this paper presents FORMS *(**F**eature-**O**riented **R**everse Engineering for **M**iddleware **S**pecialization), which is a framework to automatically specialize general-purpose middleware for product-line variants.* FORMS *provides a novel model-based approach to map product-line variant-specific feature requirements to middleware-specific features, which in turn are used to reverse engineer middleware source code and transform it to specialized forms thus resulting into vertical decomposition. Empirical results evaluating memory footprint reductions (40%) are presented along with qualitative evaluations of reduced maintenance efforts and pointers to discrepancies in middleware modularization.*

Keywords: middleware, specialization, reverse engineering, domain-specific modeling languages, FOP, features, product-line

1 Introduction

Product-line architectures (PLA) have emerged to become one of the most widely used paradigms for software development in varied domains where commonality and variability plays a crucial role in determining the reusability, flexibility, adaptability, evolvability, maintainability and quality of service (QoS) provided by the product variants to the end users. The commonality is shared by different products of the product line whereas variability manifests itself into differentiating different product variants. The variability may manifest itself in the form of differences in functionality and configurability, among others.

To support these commonalities and variabilities, and to maximize reuse, middleware, such as CORBA, J2EE, and .NET, provides abstraction of complexity and heterogeneity. These middleware are designed to be general-purpose, highly flexible and very feature-rich *i.e.*, they provide rich set of capabilities along with their configurability to support a wide range of application classes in many domains.

Despite the benefits of general-purpose middleware for a PLA application as a whole, individual product variants must, however, incur the penalty of excessive memory footprint and potentially performance overhead due to the excessive set of features, many of which may not be needed by the product variant. Additionally, excess set of features results in unwanted testing and maintenance costs per variant, which is detrimental to a cost-effective PLA management.

A logical solution to the above-mentioned challenge is to automate the specialization of general-purpose middleware for product variants of the PLA application. PLA research to date has focused primarily on application-level details but ignored issues at the middleware level. Prior that deal with middleware specialization have focused on forward engineering based on composition and stepwise refinement, *e.g.*, AHEAD [2], CIDE [6], and FOMDD [7], wherein specialized middleware are built by incrementally refining the source based on domain requirements.

These prior efforts at specialization do not apply to contemporary middleware since these middleware are designed and modularized usually with concern for extensible class hierarchies alone. Unfortunately, satisfying the requirements of product variants of a PLA application require a modularization of code along domain concerns. We call the modularization according to domain concerns as vertical middleware decomposition or specialization into feature modules. Since middleware needs to cater to multiple domains so the middleware developer focuses more on horizontal decom-

position into layers.

Domain concerns (which we call features) are often entangled and spread beyond the module (*i.e.*, class and package) boundaries across multiple modules and even middleware source. So even if a middleware packager decides to compose a specialized middleware version based on the intended design modularity he/she still ends up with many excessive features that are not necessary for the target application.

A promising approach relies on reverse engineering techniques such as source code analysis since is is not restricted by module or layer boundaries. To realize this approach, we present the **F**eature-**O**riented **R**everse Engineering for **M**iddleware **S**pecialization approach and its resulting framework (*FORMS*) for refactoring general-purpose middleware that can be combined with application-level product line engineering. *FORMS* reverse-engineers existing middleware source code and synthesizes custom versions of middleware that are composed of only the features required by the product variants.

FORMS provides a multi-step process that (1) evaluates domain requirements using a wizard-driven reasoning that maps the platform-independent (PIM) domain requirements to a PIM middleware feature model, (2) subsequently prunes the PIM middleware feature model into the PLA or product variant specific feature model using the wizard interpreter tools, (3) determines which platform-specific (PSM) middleware features that are to be directly and indirectly included in the construction of the specialized middleware, (4) uses a sophisticated algorithm to synthesize independent feature modules corresponding to the pruned middleware feature model, (5) customizes the build system and synthesizes libraries for the individual specialized middleware variants corresponding to the individual product variants.

The rest of the paper is organized as follows: Section 2 discusses the related research efforts and classifies middleware specialization techniques; Section 3 describes the *FORMS* approach to middleware specialization; Section 4 evaluates the *FORMS* approach by checking correctness and calculating footprint reduction; and finally Section 5 provides concluding remarks alluding to future research issues and lessons learned.

2 Related Work

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

Forward Engineering Approaches:

• *Feature-oriented programming (FOP) for feature module construction:* Current PLA research is supported primarily through feature-oriented programming (FOP) techniques as advocated by AHEAD [2], CIDE [6], and FOMDD [7]. These are based on processes that annotate features in source code and creates feature modules that are essentially fragments of classes and their collaborations that belong to a feature. Some efforts in this direction stem from the identification feature interactions, their dependencies, granularity and their scope [1].

• *Aspect-oriented programming (AOP) for modularizing crosscutting concerns:* AOP provides a novel mechanism to reduce footprint by enabling crosscutting concerns between software modules to be encapsulated into user selectable aspects. *FACET* [4] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. To support functionality not found in the base code, FACET provides a set of features that can be enabled and combined subject to some dependency constraints. By using AOP techniques, the code for each of these features can be weaved at the appropriate place in the base code.

Reverse Engineering Approaches:

• *Design Pattern Mining from source:* Substantial research has been performed on discovering design and architectural patterns from source code [3]. However, most such techniques are informal and therefore lead to ambiguity, imprecision and misunderstanding, and can yield substandard results due to the variations in pattern implementations. In order to specialize middleware such design mining techniques need to be well supported by round-tripping techniques that will enable any specializations at design level to reflect back into the source code.

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

3 The *FORMS* approach

This section presents the *FORMS* approach and the resulting framework for middleware specialization. We assume that middleware developers often start devel-

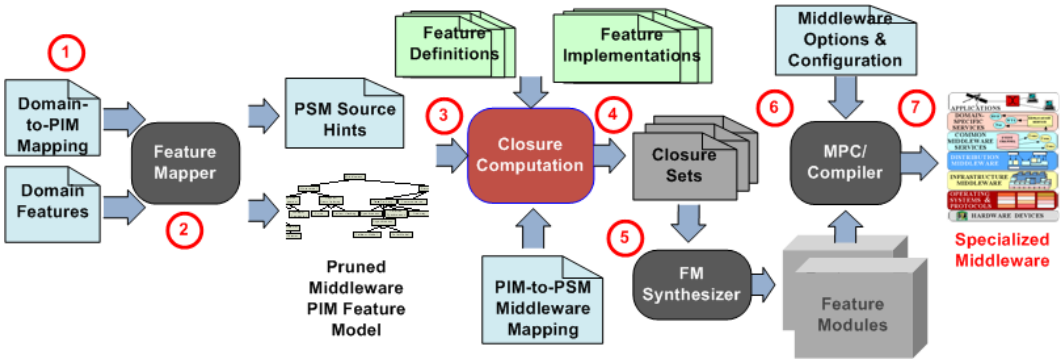

Figure 1. *FORMS* **Middleware Specialization Process**

oping 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.

FORMS is based on reverse engineering and takes a top-down approach where it identifies the feature modules within the code base, their dependencies, and then attempts to map the domain concerns to the feature modules. Subsequently, based on the selected domain concerns, it composes the corresponding implementation feature modules to synthesize the specialized middleware variant.

In *FORMS*, we view domain concerns to represent platform independent feature models (PIM) whereas middleware platform features represent platform-specific feature models (PSM). *FORMS* is built within a feature-oriented software development (FOSD) environment and has a host of associated tools that help interpretation of these feature models, their transformations from PIM to PSM and profiling the specialized middleware configurations synthesized using the *FORMS* tools.

Figure 1 shows an overview of the middleware specialization process that PLA developers use for their product variants. We briefly describe the steps in the *FORMS* process below:

1. *Feature Specification:* PLA application developer starts the middleware specialization wizard and begins describing the characteristics of the product to be developed specifying the domain-level features needed for the variant.
2. *Feature Mapping Wizard:* The Feature Mapping wizard performs maps the PIM product-line domain concerns to PIM middleware features. The wizard asks questions about the configuration characteristics and options of the product for which middleware is to be developed. These characteristics include distribution features such as client/server, concurrency features such as single/multi-threaded, in that order. The selected features are also configured along the way as they are selected for composition. The wizard can ask further fine-grained questions within each individual coarse-grained feature that is being selected to exactly configure that feature. The PLA developer response determines the next question that will be asked.
3. *Build Configuration:* The wizard then creates build configuration files that contain hints as to what source files to include in the middleware build. These files basically identify the starting points for creating the closure sets of source file dependencies where no file within a closure has dependencies on files outside the closure set.
4. *Closure Computation/Feature Module Composition:* 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).
5. *Product Variant Composition:* The feature modules are then composed into product variants which map to domain concerns directly.
6. *Build Configuration Specialization:* The build configuration is specialized by adding source files from individual feature modules to the build descriptor and thereby generating the build configuration file, such as a Makefile. For our evaluations, *FORMS* generates the Make Project Creator (MPC) file (`www.ociweb.com/products/MPC`). This MPC file represents the part of the specialized middleware that is to be built for the product variant.
7. *Specialized Middleware Synthesis:* This MPC file is then used to create platform-specific make files using the MPC scripts. The platform-specific make files are then used to build specialized mid-

dleware for the product line or product variant.

Notice that this process is entirely repeatable and reusable. A repository of requirements for product variants can be maintained. There is no need to maintain the customized version of the middleware. In the rest of the section we focus on some of the important building blocks of *FORMS*.

3.1 Design of Feature Mapping Wizard

In the PLA development process, *FORMS* is applicable in the packaging and assembly phases where the PLA application and variant along with its middleware is configured and packaged. The requirements reasoning wizard performs the difficult job of mapping the PIM product-line domain concerns to PSM middleware features.

Domain concerns describe the characteristics of the product being developed. These characteristics may include functional concerns as well as non-functional (QoS) concerns. Functional concerns describe the way a particular application/product behaves and its configuration. Non-functional concerns usually describe the way a product is supposed to perform which include dimensions of concurrency, event processing, protocols, etc.

Normally, domain concerns and middleware features manifest themselves into a hierarchial representation. 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.

Feature models of general-purpose middleware as shown in figure 2 tend to be very complex and huge and hence very cumbersome to analyze for modularity. Fortunately, the product variants feature sets are limited, which makes the concerns mapping 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 concern model and middleware feature model.

After performing this mapping the pruned middleware PIM feature set is generated that is used to synthesize the specialized middleware for the particular product variant. We assume that the platform-specific middleware features to source code mapping is already performed beforehand by the middleware developer at design time enabling us to directly determine the source code that implements the middleware feature set and hence the domain concerns. The wizard outputs the source code hints that act as the starting point of the closure computation algorithm.

3.2 Discovering Closure Sets

Once the source code hints that directly implement the domain concerns are determined, their dependencies on other code within the middleware need 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.

However opening each file on-the-fly and checking the dependencies is inefficient. Instead we run an external dependency walker tool like Doxygen or Redhat Source Navigator to extract out the dependency tree. We have designed a recursive closure computation algorithm that walks through the source code dependency tree and collates the source that is dependent on the feature. We differentiate between feature definition and feature implementation files. Feature definition makes it easier to identify and annotate features where as feature implementations may differ from one middleware implementation to another depending upon the language of implementation.

Algorithm 1 Algorithm for Computing Closure Set for a product variant

```
1: M_s : Mapping of PSM middleware features to source
2: F_p : Feature Set for Product Variant p
3: C_p : Closure set for product p ∈ F_p
4: F_d : Set of features dependent on each feature in F
5: C_f : Closure set for feature f ∈ F_p
6: C_s : Closure set for source hint s ∈ M_s
7: P_i : Pending set of feature implementations whose closure set
   needs to be calculated
8: Input: F_p, M_s
9: Output: C_p (Initially empty)
10: begin
11: C_p := ∅
12: for each feature f ∈ F_p do
13:     s := FIND source hint from M_s for feature f
14:     C_f := ∅
15:     C_s := ∅
16:     C_s := COMPUTE closure for source hint s
17:     C_f := C_f ∪ C_s
18:     F_d := FIND features dependent on f ∈ C_f
19:     P_i := FIND new feature implementation files for each feature
        in F_d
20:     while P_i isnotempty do
21:         C_s := ∅
22:         C_s := COMPUTE closure for implementation file i ∈ P_i
23:         C_f := C_f ∪ C_s
24:         P_i := FIND new feature implementation files for new
            source files that were found in the closure computation
25:     end while
26:     C_p := C_p ∪ C_f
27: end for
28: return C_p
29: end
```

1. **Lines (1-7):** The middleware developer provides the mapping from the PSM middleware features to the feature definition files in which the features are

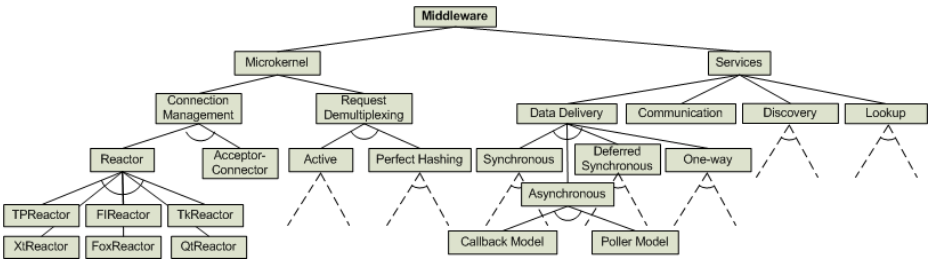

Figure 2. Middleware PIM Feature Model

mapped. Since this is a coarse-grained mapping it is simpler to designate a set of files for a feature definition.

2. **Lines (10-17):** Once these source code hints are obtained the algorithm computes the closure set for each of the source code hints. This step produces even more dependent feature definition files which automatically form part of the closure set. Their closure need not be recalculated.
3. **Line (18):** The previous step gave rise to potentially more dependent feature sets that are not directly used by the product-line variant. The algorithm identifies the implementation files for the features in dependent feature sets.
4. **Line (19):** However 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 feature implementation file until all the pending implementation files are accounted for. The closure computation will always give rise to more feature implementation files as described in the 2nd step.

3.3 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, ANT scripts, etc. while some of them rely on descriptor files such as make file system, advanced cross-compiler build facilities like MPC (Make Project Creator), etc. 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 product-line variant requirements.

4 Evaluation & Future Work

We evaluate *FORMS* by modeling a product-line of networked logging applications based on contemporary, widely used communication middleware such as ACE [5]. ACE is a free, open-source, platform-independent, highly configurable, object-oriented (OO) framework that implements many core patterns for concurrent communication 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.

The candidate product-line we have chosen is based on the client-server paradigm with individual models conforming to various MoCs including simple, iterative, reactive, Thread-per-connection (TPC), real-time thread-per-connection (RT-TPC) and process-per-connection (PPC). Each product 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.

By creating specialized variants of ACE, *FORMS* profiling tools estimate the memory footprint savings, dependent features, source files that implement the features, and exercise unit tests to determine whether the expected performance is met. We showcase the

Networked Logging Applications PLA		Outcome of Closure Computations		Synthesized Middleware
Product 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	177	502	1,456
Reactive Logging	12	295	502	1,456
Thread Per Connection Logging	11	213	502	1,456
Real-Time Thread Per Connection Logging	12	215	502	1,456
Process Per Connection Logging	12	193	508	1,500

Table 1. Outcome of applying *FORMS* to a Product-line of Networked Logging Applications

compile-time metrics that result from middleware specialization. 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 core to be specialized is implemented in 1388 source files which result into a footprint of 2,456 KB. Table 1 shows that FORMS has achieved significant optimizations - a 64% reduction in the number of middleware source files used and a 41% reduction in the middleware footprint. Table 1 also shows that the PLA variants share many middleware PIM features as verified by the almost similar footprint measurements.

However, the implementing middleware PSM features that are being used have substantial variations. This means general-purpose middleware even though designed in a modular way, the modularity does not manifest exactly in the same way in their implementations in the middleware layers. Since each layer needs to cater to multiple domains so the middleware developer focusses more on horizontal decomposition (layers) rather than vertical decomposition (feature modules). More specifically after inspecting the individual product variant generated MPC build configuration, there were some unused PSM features that percolated into the feature modules of a product variant.

FORMS can advise middleware developers to correct their implementation mistakes by breaking unwanted dependencies with the middleware modules. This will help reduce the coupling between the modules within the middleware layers and minimize the percolation of unused features in feature modules. However this will not automatically decompose the middleware along domain concerns. The *FORMS* will be required to perform vertical decomposition of the middleware.

Furthermore, the lack of fine granularity of modularization in their design make general-purpose middleware heavyweight solutions and are a performance overhead. *FORMS* needs to tackle the fine-grained modularity by automatically annotating code and generating the middleware specialization directives. We intend to investigate such issues in our future work by further improving the *FORMS* tools based on the anomalies and discrepancies that *FORMS* can discover.

5 Concluding Remarks

Forward engineering, though systematic and elegant techniques for synthesizing specialized middleware, does not modularize middleware implementations along domain concerns that are often entangled and span the conventional modularization boundaries in middleware. *FORMS* has shown that reverse engineering techniques based on source code analysis offer a promising and viable alternative to modularize domain concerns within middleware code.

Source code analysis techniques tend to be coarse grained at their best but can provide crucial pointers to the lack of proper implementation methods by showcasing the difference between the intended module designs (PSM) and their code implementations (PIM).

References

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

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[3] J. Dong, Y. Zhao, and T. Peng. Architecture and design pattern discovery techniques - a review. In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 621–627. CSREA Press, 2007.

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

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

[6] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 311–320, New York, NY, USA, 2008. ACM.

[7] S. Trujillo, D. Batory, and O. 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.