FORMS: Feature-Oriented Reverse Engineering-based Middleware Specialization for Product-Lines

Akshay Dabholkar and Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville
Email: {aky,gokhale}@dre.vanderbilt.edu

***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 and feature-rich middleware platforms. However, each product variant now incurs memory footprint and performance overhead due to the feature-richness of the middleware along with the increased cost of its testing and maintenance. To address this tension, this paper presents FORMS (Feature-Oriented Reverse Engineering for Middleware Specialization), 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* resulting in vertical middleware decompositions. Empirical results evaluating memory footprint reductions (40%) and feature reductions (60-76%) are presented alo656ng with qualitative assessment of discrepancies in modularization of contemporary middleware.**

***Keywords*-Middleware; Specialization; Reverse Engineering; Closure; Footprint; Feature Oriented Programming; Product-line**

I. Introduction

Product-line engineering (PLE) [1] has 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 distinguishes individual product variants. The variability may manifest itself in terms of functionality or configurability or both.

To support these commonalities and variabilities, and to maximize reuse, middleware, such as CORBA, J2EE, and .NET, provides abstraction of complexity and heterogeneity. To be widely applicable across multiple domains, 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.

Despite the benefits of general-purpose middleware for a product line as a whole, individual product variants, however, incur the penalty of excessive memory footprint and potentially performance overhead due to the excessive set of middleware features – many of which may not be required 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 PLE management.

A promising solution to address the above-mentioned challenge is to specialize general-purpose middleware for product variants of the product line. Prior research on middleware specialization has focused on forward engineering techniques, such as Feature Oriented Programming (FOP) [2] and Aspect Oriented Programming (AOP) [3], which are based on composition and stepwise refinement. Some examples of these approaches include *FACET* [4], *Modelware* [5], *LOpenOrb* [6], and AHEAD [7], CIDE [8], and FOMDD [9], wherein specialized middleware is built by

Since middleware needs to cater to multiple domains (*i.e.*, be general-purpose and flexible), they are designed and modularized with a focus on extensible class hierarchies alone. Hence the middleware developer focuses more on *horizontal decomposition* of middleware into layers. In contrast, to support product variants, PLE requires the middleware code to be modularized along domain concerns. We call such a modularization as *vertical middleware decomposition* or *feature module specialization*.

We observe that much of the contemporary middleware available is still not developed using the top-down PLE techniques of domain engineering and application engineering but in fact built bottom-up based on a modularized design template. However, the PLE domain concerns (which we call features) are often tangled with each other, and are spread beyond the module (*i.e.*, class and package) boundaries across multiple modules within the middleware source. Hence, even if a middleware packager decides to compose a specialized middleware version based on the intended design modularity, the specialized version of the middleware still results in many excessive

This paper is based on "Middleware Specialization for Product-lines using Feature Oriented Reverse Engineering," by Akshay Dabholkar, and Aniruddha S. Gokhale, which appeared in Published in the Proceedings of the 7th International Conference on Information Technology : New Generations (ITNG 2010), Las Vegas, NV, USA, April 12-14, 2010.
© 2010 IEEE.
This work was supported by the NSF CAREER Grant 0845789.

features that are not necessary for the particular domain concern being tackled by the target application. As a consequence, prior research on middleware specialization does not address PLE issues.

An approach to resolve this challenge relies on reverse engineering techniques such as source code analysis since they are not restricted to module or layer boundaries imposed by traditional bottom-up composition techniques. Since reverse-engineering techniques rely more on top-down approaches using introspection and reflection, they address the PLE "application engineering" phase. Therefore, in this paper we primarily focus on PLE application engineering whereas we employ FOP-based reasoning that deduces domain engineering concerns to drive the overall process. Thus, reverse engineering driven by domain concerns enables the implicit analysis and decomposition along domain concerns.

To realize these goals, we present the **F**eature-**O**riented **R**everse Engineering for **M**iddleware **S**pecialization (FORMS) approach and the resulting framework for refactoring general-purpose middleware along individual domain concerns 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 individual product variants.

FORMS provides a multi-step process as follows: (1) it evaluates domain requirements using a wizard-driven reasoning that maps the platform-independent (PIM) domain requirements to a PIM middleware feature model, (2) it subsequently prunes the PIM middleware feature model into the PLE or product variant-specific feature model using the wizard interpreter tools, (3) it determines which platform-specific (PSM) middleware features are to be directly and indirectly included in the construction of the specialized middleware, (4) it uses a sophisticated algorithm to synthesize independent feature modules corresponding to the pruned middleware feature model, and (5) it 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 II presents the FORMS approach to middleware specialization; Section III evaluates the FORMS approach by checking correctness and calculating footprint reduction using a representative case study of networked logging server product line variants. It also discusses the insights gained for potential enhancements that can be incorporated within FORMS for addressing fine-grained feature composition and discovering further issues with respect to middleware modularity concerns; Section IV discusses the related research efforts and classifies middleware specialization techniques; and finally Section V provides 'concluding remarks alluding to future research issues and lessons learned.

II. The FORMS Middleware (De)composition Process

This section presents the FORMS approach and the resulting framework for middleware specialization for product lines using feature-oriented reverse engineering. We assume that middleware developers develop module code bottom-up based on a design template and subsequently create the corresponding build configurations for their modules through mechanisms, such as Makefiles or Visual Studio Project files.

FORMS is based on reverse engineering and takes a top-down approach where it identifies the feature modules within the middleware code base, and their dependencies based on the domain concerns that were identified in the PLE domain engineering phase. Subsequently, based on the selected domain concerns, it composes the corresponding implementation feature modules to synthesize the specialized middleware variant thereby vertically decomposing the middleware.

In FORMS, we view domain concerns to represent platform-independent feature models (PIM) whereas middleware platform features represent platform-specific feature models (PSM). FORMS provides a process to transform the PIM domain concerns to PIM middleware concerns and subsequently to PSM middleware implementation concerns, which finally drive the generation of specialized middleware for a given set of domain concerns. FORMS is built as a feature-oriented software development (FOSD) environment within a bigger toolchain called GAMMA (Generators and Aspects for Manipulating Middleware Architectures). GAMMA (http://www.dre.vanderbilt.edu/GAMMA/) has a host of associated tools that help the interpretation of these PIM feature models, their transformations from PIM to PSM, and profiling the specialized middleware configurations for performance and footprint metrics.

A. Overview of the FORM Process

Figure 1 illustrates the FORMS middleware specialization process that PLE developers use for their product variants. We briefly describe the steps of the FORMS process below:

1. *Feature Mapping Wizard:* The PLE application developer starts the middleware specialization wizard and begins describing the characteristics of the product to be developed specifying the PIM product-line, domain-level concerns needed for the variant. The Feature Mapping wizard maps the PIM product-line domain-level concerns to PIM middleware features. The wizard asks questions about the configuration requirements and options of the product for which middleware is to be developed. These requirements 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

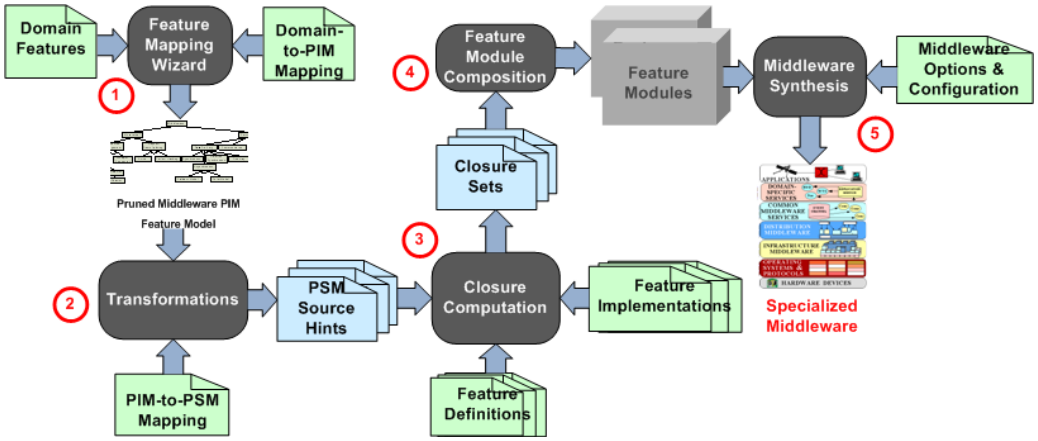

Fig. 1. **FORMS Middleware Specialization Process**

questions within each individual coarse-grained feature that is being selected to exactly configure that feature. The PLE developer response determines the next question that will be asked.

The wizard then creates initial build configuration files that contain hints as to what source files to include in the middleware build. These files 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. Note that the FORMS tool understands the middleware code organization including the organization of the source files.

2. *Transformations:* Once the pruned PIM middleware feature set is obtained from the wizard, it is then mapped to the actual PSM middleware features that implement the individual PIM features using the PIM-PSM mappings that are provided by the middleware developer.
3. *Closure Computation:* Once the hints are obtained, they are used to create closure sets using an algorithm that systematically composes the source code and files that are associated with each feature into a feature module (FM). The closure sets are essentially all the dependencies that are gathered by the tool.
4. *Feature Module Composition:* The feature modules for each product's desired feature are then composed into product variants which map to domain concerns directly.
5. *Specialized Middleware Synthesis:* The build configuration is specialized by adding source files from individual closure sets of feature modules to the build descriptor thereby generating the build configuration file, such as a Makefile. For our evaluations, FORMS generates the Make Project Creator (MPC) [10] build configuration file. This MPC file represents the part of the specialized middleware that is to be built for the product variant. The generated MPC file is then used to create PSM Makefiles by running the MPC-supplied perl-based scripts. The platform-specific Makefiles are then used to synthesize the specialized middleware 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 versions of the middleware since it can be synthesized through this process on demand. In the rest of the section we focus on some of the important building blocks of FORMS.

B. The FORMS Stages

1) Feature Mapping Wizard: In the PLE development process, FORMS role is applicable in the packaging and assembly phases where the PLE application variants along with the hosting middleware is configured and packaged. The requirements reasoning wizard performs the difficult job of mapping the PIM product-line domain concerns to PIM 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 includes dimensions of concurrency, event processing, protocols, etc.

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

Feature models of the general-purpose middleware as shown in Figure 2 tend to be very complex and huge mak-

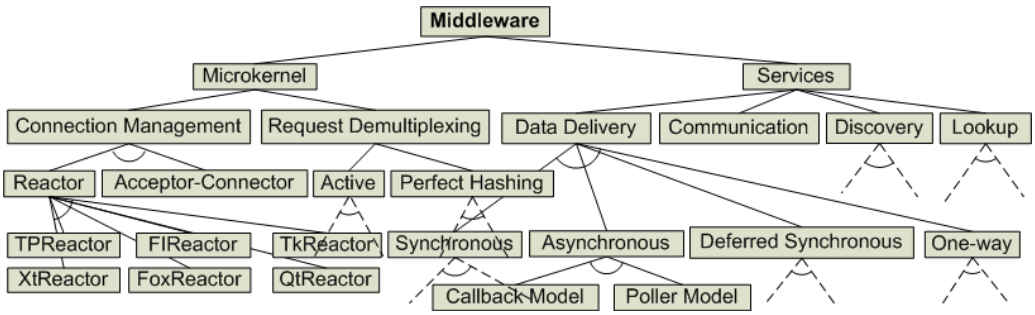

Fig. 2. **Middleware PIM Feature Model**

ing it very cumbersome to analyze for modularity. Fortunately, the feature sets for product variants are limited, which makes the mapping of concerns tangible within the middleware feature set. This helps us map known domain concerns to the middleware features in advance resulting in a $m : n$ correspondence between the domain concern model and middleware feature model. Thus, based on the domain concern model, the middleware feature model needs to be pruned to remove the unwanted features that do not map to the domain concerns. This is done through the feature model interpreters provided by FORMS.

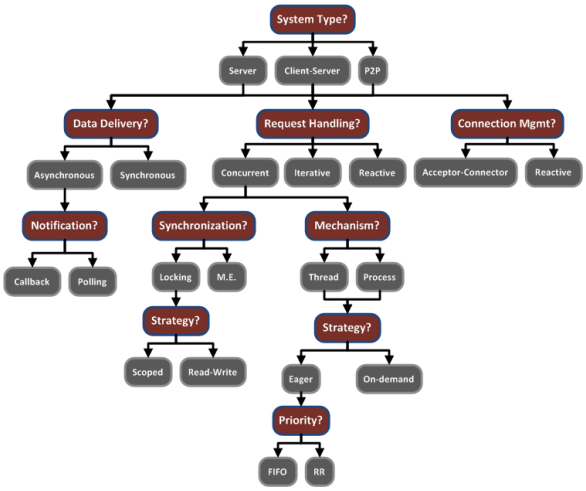

Fig. 3. **Decision Tree used by the Feature Mapper Wizard**

The feature mapping wizard traverses an internal decision tree as shown in figure 3 to ask different questions to the product-line developer to infer the product variant characteristics. It asks questions ranging from coarse-grained ones like whether the product variant is client-server or peer-to-peer, to fine-grained questions like what kind of thread-spawning strategy is desired. Each coarse-grained answer scopes down the product characteristics based upon which the next fine-grained questions are asked that configure the product behavior.

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

2) Discovering Closure Sets: Once the PSM source code hints that directly implement the domain concerns are determined, their dependencies on other code within the middleware needs to be determined. All such code that is interdependent on each other is what implements the domain concern. We call such a set of source files as a *closure set* in which there are no source file dependencies going out of the closure set. We differentiate between feature definition and feature implementation files. Feature definition makes it easier to identify and annotate features whereas feature implementations which capture the feature behavior may differ from one middleware implementation to another depending upon the language of implementation. Thus the closure computation identifies the set of dependent features definitions and their definitions, and composes them into a coherent and independent feature module.

We have designed a recursive closure computation algorithm that walks through the source code dependency tree and identifies the source that is dependent on the feature. However, opening each file on-the-fly and checking the dependencies is inefficient since it requires numerous I/O operations. Instead we run an external dependency walker tool like Doxygen or Redhat Source Navigator [11] to extract out the dependency tree.

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

Algorithm 1 Algorithm for Computing Closure Set for a product variant

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

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

The closure sets corresponding to the product variants that are discovered in Section II-B2 are different from cliques or maximally independent sets in graph theory. Closure sets, though transitive, are completely self-sufficient so they can also be called independent transitive closures.

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, or ANT scripts while some of them rely on descriptor files such as make file system or advanced cross-compiler build facilities like MPC (Make Project Creator) [10]. 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.

III. Evaluation

A. Logging Server Case Study

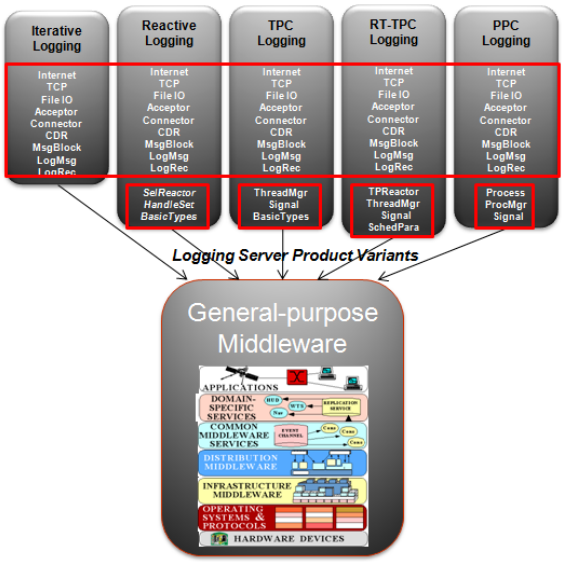

Fig. 4. **Logging Product Line**

In order to explain and evaluate the FORMS middleware specialization process, we use a motivating example of a product-line of networked logging servers as shown in Figure 4. We choose this particular product line since logging various status and error messages is a very frequent and widely used facility for monitoring the system performance as well as system survivability in different domains such as enterprize, or distributed real-time and embedded systems like shipboard computing and mission critical aviation software.

A logging server has different performance requirements depending upon the type of application that is using the logging facility. Depending upon the application domain the need for logging varies from sporadic to frequent logging. Enterprize applications may require sporadic logging where logging is restricted to mostly error and status messages whereas certain high security mission critical application that are susceptible to infiltrations may require more detailed logging traces of the system behavior in order to detect discrepancies and errors that may lead to discovering an impending or in-progress security attack. Hence sporadic logging may require iterative or reactive logging servers whereas frequent logging may require multithreaded or multiprocess logging servers.

We evaluate FORMS by modeling a product-line of networked logging applications based on contemporary, widely used communication middleware such as ACE [12]. ACE is a free, open-source, platform-independent, highly configurable, object-oriented (OO)

framework that implements many core patterns for concurrent communication in software. It enables developing product variants using various types of communication paradigms such as client-server, peer-to-peer, event-based, publish-subscribe, etc. Within each paradigm it supports various models of computation (MoC) which are highly configurable for different QoS requirements. We have designed the networked logging product-line servers based on the client-server paradigm with individual models conforming to various MoCs including iterative, reactive, thread-per-connection (TPC), real-time thread-per-connection (RT-TPC) and process-per-connection (PPC). Each 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.

Figure 4 shows the representation of the logging server product line in terms of commonality and variability of the features. We have showcased only those features that are required since we are not interested in how the individual logging server variant is implemented but rather what PIM features it desires from the underlying middleware platform.

B. Experimental Results of applying the FORMS process

By creating specialized variants of ACE middleware for different types of logging servers, FORMS profiling tools estimate the memory footprint savings, dependent middleware features, source files that implement the features, and exercise unit tests to determine whether the expected performance is met. We showcase the compile-time metrics that result from middleware specialization.

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

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

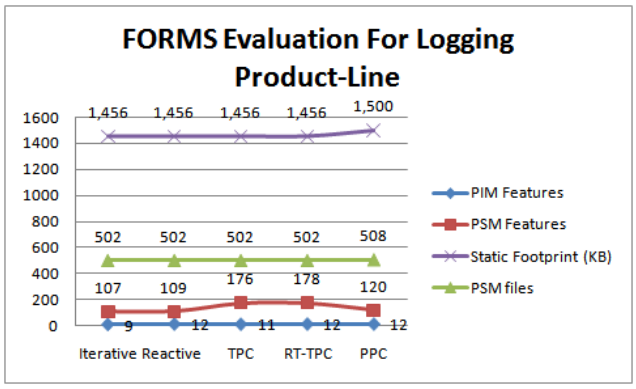

Fig. 5. Modularization Disparities

2) Modularization Discrepancies: On the other hand as shown in Figure 5, there is a wide disparity between the number of PSM middleware features required by the individual product variants (107-178) variants and the PSM source files (502-508) implementing them. More specifically after inspecting the individual product variant's generated MPC build configuration, there were some unused PSM features that percolated into the feature modules of a product variant. This means that there are several unused middleware features that find their way in the specialized middleware for the Iterative, Reactive and PPC product variants that originally required fewer features.

C. Additional Insights provided by FORMS

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

1) **Discovering Modularization Discrepancies:** The reason for the modularization discrepancies described in section III-B2, are due to the physical implementation dependencies between the logical feature modules. These results from the conflicts between the design goals envisioned by the middleware designers and the implementation goals of the middleware developers. This happens if a single PSM implementation source file implements more than one PIM feature or vice versa. Thus the logical PIM feature independence does not always translate to their actual physical PSM implementation independence. Thus even though general-purpose middleware is designed in a modular way, the modularity does not manifest exactly in the same way in their implementations of the middleware layers. FORMS can provide a guideline to the middleware developers to detect and break unnecessary dependencies within their source code and thereby reduce the tight coupling between the modules within the middleware layers.

TABLE I
Outcome of applying FORMS to a Product-line of Networked Logging Applications

Networked Logging Applications Product Line		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	107	502	1,456
Reactive Logging	12	109	502	1,456
Thread Per Connection Logging	11	176	502	1,456
Real-Time Thread Per Connection Logging	12	178	502	1,456
Process Per Connection Logging	12	120	508	1,500

2) **Automated Test Case Selection:** FORMS reduces the amount of features, in turn reduces the functionalities that are expected from the middleware. Thus it can enable automatic test case selection of functional unit tests in order to alleviate the testing and maintenance overhead for the middleware developers
3) **Discovering Middleware Core:** FORMS helps in identifying the core middleware features needed by the product-line. FORMS can take a multiset intersection of all the closure sets that are generated for the different product-line variants. This intersection represents the commonality whereas the rest of the features represent the variability. Thus, FORMS can potentially figure out the differences between the logical middleware core as designed and envisioned by the middleware architect and physical middleware core estimated by the closure computation.

D. Validation of the FORMS approach

As middleware is statically specialized, checking the correctness of its functionalities becomes paramount. In this case a simple successful compilation of the specialized middleware and shared library generation are not sufficient. It becomes necessary to verify the runtime correctness of the specialized middleware through exhaustive testing processes. We validated the FORMS methodology by re-executing the tests on the specialized middleware that were originally designed for the general-purpose middleware. However, we also ensured that the tests that have been invalidated due to the missing features from the specialized middleware are pruned away and not re-executed.

IV. 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.

A. Forward Engineering Approaches

1) Feature-oriented programming (FOP) for feature module construction: Current PLE research is supported primarily through feature-oriented programming (FOP) techniques as advocated by AHEAD [7], CIDE [8], and FOMDD [9]. These approaches are based on processes that annotate features in source code and compose feature modules that are essentially fragments of classes and their collaborations that belong to a feature. Being forward engineering techniques that they rely on clear identification of features, their dependencies and their interactions right from the requirements gathering stage of the PLE software lifecycle. Some efforts in this direction stem from the identification of feature interactions, their dependencies, granularity and their scope [13].

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

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

3) Combining modeling and aspects for refinement: The *Modelware* [5] methodology adopts both the model-driven architecture (MDA) [14] and AOP. Borrowing terms from subject-oriented programming [15], the authors use the term *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains. They use the term *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change.

Modelware advocates the use of models and views to separate intrinsic functionalities of middleware from extrinsic ones. Modelware considerably reduces coding

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

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

B. Reverse Engineering Approaches

1) Design Pattern Mining from source: Substantial research has been conducted on discovering design and architectural patterns from source code [16]. However, most such techniques are informal and therefore lead to ambiguity, imprecision and misunderstanding, and can yield substandard results due to the variations in pattern implementations. In order to specialize middleware such design pattern mining techniques need to be well supported by round-tripping techniques provided by FORMS that will enable any specializations at design level to reflect back into the source code. We are investigating the application of such techniques to automate feature annotation in source code.

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

V. Concluding Remarks

Although forward engineering provides systematic and elegant techniques for synthesizing specialized middleware, it does not modularize middleware implementations along domain concerns that are often entangled and crosscut conventional horizontal modularization boundaries in middleware. In this paper we present FORMS which is a reverse engineering techniques based on source code analysis that offers a promising and viable alternative to modularize domain concerns within middleware code. Source code analysis techniques tend to be coarse grained at best but can provide crucial pointers to the lack of proper implementation methods by showcasing the difference between the intended PIM module designs and their PSM code implementations.

Future Work and Open Issues: Following are the open issues not handled by FORMS that impede fine-grained modularization of middleware:

- **How do we handle feature interactions?** Features are often known to interact [17] with each other. For example, when performance and resource constraints are also to be addressed across the lifecycle, it is conceivable that specializations that satisfy one requirement may interact in unforeseen ways with other kinds of specializations.
- **How to efficiently annotate middleware source code for feature identification and management?** There a need to systematically refactor contemporary middleware for feature pruning/augmentation that can be a monumental task for middleware developers. This can only be achieved by devising efficient advanced annotation mechanisms that identify middleware features, their dependencies and interactions, which can then be leveraged by tools like FORMS.
- **How to tackle fine-grained modularization?** lack of fine granularity of modularization in their design make general-purpose middleware heavyweight solutions and 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 and by integrating contemporary tools like CIDE, AHEAD, and FOCUS [18] to support fine-grained feature composition.

References

[1] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society Washington, DC, USA, 2004, pp. 702–703.

[3] G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols," *Commun. ACM*, vol. 44, no. 10, pp. 95–97, 2001.

[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)*. Berlin, Germany: ACM Press, 2002, pp. 38–45.

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

[6] G. 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*. London: Springer-Verlag, 1998, pp. 191–206.

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

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

[9] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 44–53.

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

[11] B. Developer, "The source-navigator^{TM} ide," http://sourcenav.sourceforge.net/.

[12] Institute for Software Integrated Systems, "The ADAPTIVE Communication Environment (ACE)," www.dre.vanderbilt.edu/ACE/, Vanderbilt University.
[13] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 162–180, March-April 2008.
[14] *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 ed., Object Management Group, Jul. 2001.
[15] W. Harrison and H. Ossher, "Subject-oriented Programming: A Critique of Pure Objects," in *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 1993, pp. 411–428.
[16] J. Dong, Y. Zhao, and T. Peng, "Architecture and design pattern discovery techniques - a review," in *Software Engineering Research and Practice*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2007, pp. 621–627.
[17] J. Liu, D. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications," in *Proceedings of the International Conference on Software Engineering*. ACM Press New York, NY, USA, 2006, pp. 112–121.
[18] A. Krishna, A. Gokhale, D. C. Schmidt, J. Hatcliff, and V. Ranganath, "Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures," in *Proceedings of EuroSys 2006*, Leuven, Belgium, Apr. 2006, pp. 205–218.

Akshay Dabholkar is a PhD candidate in the Department of Electrical Engineering and Computer Science (EECS) in Vanderbilt University.

His research interests are broadly in the area of Middleware Specializations with focus on AOP, FOP, reverse engineering, reflective, and generative techniques to improve the real-time and fault tolerance quality of service (QoS) provisioning.

Mr. Dabholkar has a MS in Computer Science from Vanderbilt University. Contact him at aky@dre.vanderbilt.edu.

Dr. Aniruddha S. Gokhale is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University, Nashville, TN, USA.

His primary research interests are in investigating novel model-driven engineering (MDE) solutions for systems problems pertaining to distributed real-time and embedded systems. He is the project lead on the CoSMIC MDE framework. Dr. Gokhale has published over 100 technical papers.

Dr. Gokhale obtained his B.E (Computer Engineering) from University of Pune, 1989; MS (Computer Science) from Arizona State University, 1992; and D.Sc (Computer Science) from Washington University in St. Louis, 1998. Prior to joining Vanderbilt, Dr. Gokhale was a member of technical staff at Lucent Bell Laboratories, NJ, USA. Dr. Gokhale is a member of IEEE and ACM.