Towards a Holistic Approach for Integrating Middleware with Software Product Lines Research

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

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

Sumant Tambe
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
sutambe@dre.vanderbilt.edu

ABSTRACT

Prior research on software product lines (SPLs) in different domains (e.g., avionics mission computing, automotive, cellular phones) has focused primarily on managing the commonalities and variabilities among product variants at the level of application functionality. Despite the fact that the application-level SPL requirements drive the specializations (i.e., customizations and optimizations) to the middleware that host the SPL variants, middleware specialization is seldom the focus of SPL research. This results in substantial and ad hoc engineering efforts to specialize middleware in accordance with the functional and quality of service (QoS) requirements (e.g., latency, reliability) of the product lines. To overcome these problems, this paper highlights the need to unify middleware specialization issues within SPL research, and argues for new research directions in modularization and generative programming techniques that can account for the deployment and runtime issues, such as QoS and resource management. Preliminary ideas demonstrating how feature-oriented programming and model-driven development tools together can address these challenges are presented.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Product Lines, Middleware Specializations, Modularizations

Keywords

Generative programming + Product lines, FOP/AOP + MDD

1. INTRODUCTION

Research on software product lines (SPLs) [7] has focused primarily on managing the commonalities and variabilities [8] in application-level functionality of product variants. Generative programming [9] and modularization techniques, such as feature-oriented programming (FOP) [20] and aspect-oriented programming (AOP) [12], play an important role in composing and synthesizing product variants from modularized units called features and aspects.

Middleware is an important asset for SPLs across many domains, such as avionics (*e.g.*, Boeing's Bold Stroke architecture [23]), telecommunications (*e.g.*, Ericsson's family of carrier class switches [1]) and even cell phones (*e.g.*, Nokia or Motorola's family of cell phones). Middleware manages the quality of service (QoS) (*e.g.*, latency, reliability, security), and resource management (*e.g.*, bandwidth, CPU, memory) issues in product variants of a SPL. SPL developers tend to rely on standardized, general-purpose middleware, such as but not limited to J2EE, .NET Web Services, and CORBA, since these middleware provide a reliable, robust, low cost and low maintenance solution with the added benefit of feature-richness, flexibility, and high degree of configurability.

Although existing research in SPLs has significantly improved the quality of product lines, and reduced their development and maintenance costs, these research efforts have seldom addressed the challenges in effectively using middleware for SPLs. Addressing middleware challenges as part of SPL research is necessary since the feature-richness and flexibility of general-purpose middleware often becomes a source of excessive resource consumption and a lost opportunity to optimize for significant performance gains and/or energy savings in SPLs. Moreover, it is infeasible for general-purpose middleware to provide solutions to all possible domain-specific requirements since they are developed with the aim of broader applicability. Developing proprietary middleware for SPLs, however, is not a viable solution due to the excessively high development and maintenance costs.

In the current state of the art these limitations are addressed through significant but often *ad hoc* engineering efforts at specializing (*i.e.*, customizing and optimizing) general-purpose middleware. To overcome these deficiencies, there is a compelling need for SPL research to consider middleware platforms as an integral part of the SPL engineering processes and methodologies. This in turn argues for new research directions in modularization and generative programming techniques that account for QoS and resource management challenges, which are inherently deployment- and run-time problems, while most generative/modularization techniques are limited to design-time.

This paper proposes an integrated SPL methodology that incorporates capabilities for *middleware specialization*. Specialization is a process that manipulates general-purpose middleware in accordance with the commonalities and variabilities of an SPL by (a) adding custom features supplied by the application, (b) pruning unwanted features, and (c) optimizing the resulting middleware to address QoS and resource requirements of SPLs. Our approach is based on exploiting a hitherto before untapped algebraic structure of middleware by synergistically integrating (a) Origami ma-

McGPLE GPCE '08 Nashville, TN, USA

trices [4], which provide a formal representation for feature composition, interaction and refactoring [14], (b) Aspects [12], which modularize software that exhibits crosscutting characteristics into reusable features, and (c) Generative programming [9], which promotes automation in middleware specialization.

The remainder of this paper is organized to portray our vision of middleware specialization shown in Figure 1. Section 2 determines the problem space for middleware specialization; Section 3 describes the details of our holistic approach to combining middleware specializations with SPL research; and Section 4 provides concluding remarks and discusses open research issues.

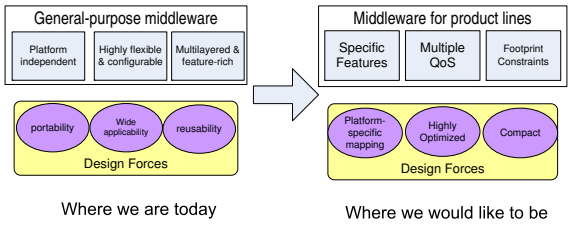

Figure 1: Middleware Specializations for SPLs

2. THE PROBLEM SPACE FOR MIDDLEWARE SPECIALIZATION

This section helps to define the problem space for middleware specialization in the context of SPLs.

2.1 Middleware System Model

The concept of middleware was born with the aim to shield applications from variabilities in lower-level artifacts, such as hardware, networks and compilers of programming languages. Years of middleware research resulted in a middleware model approximated by Figure 2. Middleware is made up of layers of software targeted to perform specific activities. At the bottom, the host infrastructure layer (*e.g.*, a Java virtual machine or the ACE [21] middleware) shields developers from the differences in operating systems and hardware. Next, the distribution layer (*e.g.*, CORBA or Java RMI) provides features for location transparency, request processing, and data marshaling, among others. The common services (*e.g.*, CORBA Naming or the UDDI discovery service) include features, such as naming, transaction, fault tolerance and real-timeliness, etc. At the top, the domain-specific middleware layer is tailored to a particular domain, such as avionics.

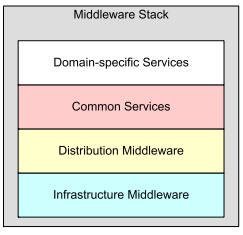

Figure 2: System Model for Middleware Specialization.

2.2 Survey of Related Research

Now we survey and organize related work along different dimensions that we observe to be prevalent in middleware specialization research.

• *Eliminating overhead of object-orientation:* Lohmann et. al. [15] argue that the development of fine-grained and resource-efficient system software product lines requires a means for separation of concerns [25] that does not lead to extra overhead in terms of memory and performance. The overhead of object-oriented programming (OOP), *e.g.*, due to dynamic binding and method dispatch, is not acceptable for some embedded systems. Aspect-oriented programming (AOP) [12] is shown to eliminate this overhead. Aspects are modularized pieces of code that traditionally are scattered across application code.

• *Aspects for footprint reduction:* AOP provides a novel mechanism to reduce footprint by enabling crosscutting concerns between software modules to be encapsulated into user selectable aspects. *FACET* [11] 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.

• *Combining modeling and aspects for refinement:* the *Modelware* [27] methodology adopts both the model-driven architecture (MDA) [17] and AOP. Borrowing terms from subject-oriented programming [10], 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.

• *Combining computational reflection and aspects:* computational reflection is an efficient and simple way of inserting new functionality into reflective middleware, such as *LOpenOrb* [5]. It uses a meta-object protocol to abstract away the implementation details so that it is necessary only to know the components and interfaces. To conserve resources and provide dynamic adaptation, AOP can be used to specialize the reflective middleware. Aspects that are not in the application code can be dynamically inserted using a meta-object protocol.

• *Layer collapsing and bypassing:* In a typical middleware platform every request passes through each layer, whether or not the services provided by that layer are needed for that specific request. This rigid layered processing can lower overall system throughput, and reduce availability and/or increase vulnerability to security attacks [19]. For use cases where the response is a simple function of the request input parameters, bypassing middleware layers may be permissible and highly advantageous. Devanbu et. al [26, 19] have shown how AOP can be used to bypass middleware layers.

• *Importance of lifecycle stages:* Traditionally, performance problems in middleware layers have been addressed by optimizing the source code and data structures. Edicts [6] is an approach that shows how optimizations are also feasible at other application lifecycle stages, such as deployment- and run-time. Just-in-time middleware customization [28] shows how middleware can be customized after application characteristics are known. These efforts discover the configuration of the target environment and compose only the necessary modules that are best suited among alternatives and configure them in the most optimal way.

3. INTEGRATING MIDDLEWARE SPECIALIZATIONS WITH SPL METHODOLOGIES

We now describe our proposed approach to integrate middleware specialization with SPL methodologies.

3.1 A Middleware Case Study

To make the description of our proposed approach concrete we use a middleware case study. Figure 3 illustrates the CORBA middleware architecture, which is compliant with our layered middleware system model. Also shown in the figure are CORBA services, real-time CORBA (RTCORBA) [18] enhancements, and component-based abstractions. CORBA is used here only for illustration purpose, however, our approach is general.

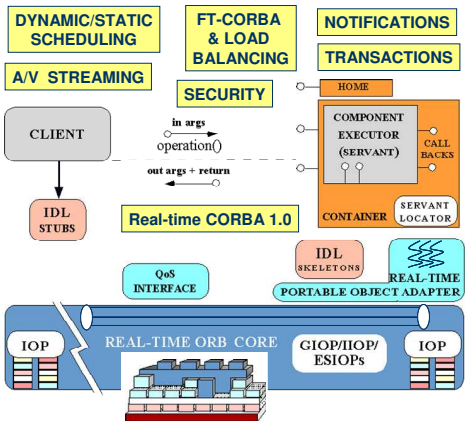

Figure 3: CORBA Architecture

The different RTCORBA features are shown in Figure 4. RTCORBA defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools. Applications typically specify these real-time QoS policies along with other policies when they call standard CORBA operations, such as `create_POA` or `validate_connection`. For example, the priority at which requests must be handled can be propagated from the client to the server (the `CLIENT_PROPAGATED` model) or declared by the server (the `SERVER_DECLARED` model).

3.2 Uncovering the Algebraic Structure of Middleware

Despite a rich repertoire of features, specializations including feature additions, pruning or customizations to general-purpose middleware is a hard problem due to the following challenges posed by their design and implementation:

a. fundamental restrictions and limited flexibility of programming languages such as C++ or Java do not allow interception of the control flow at arbitrary points in the control flow graph to inject required application-specific functionality or remove certain unnecessary functionality. This is currently feasible only at limited points in the code known as *interception points*, which is often not sufficient.

b. although object-oriented designs help develop modular middleware code, this modularity incurs a performance penalty.

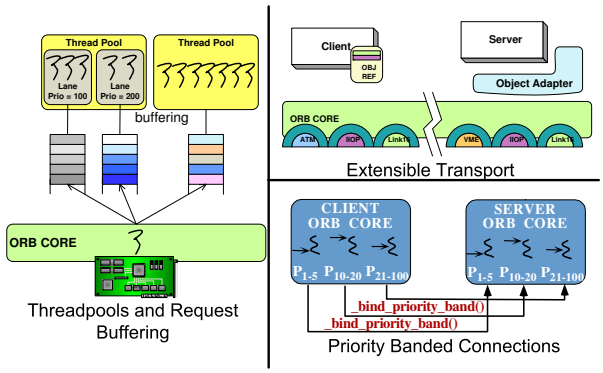

Figure 4: RTCORBA Features

Maintaining the modular design, which promotes longevity of the software, is desirable yet obtaining optimal performance is also required. There is a need to decouple specializations from the modular design, which is a hard problem.

c. the combinatorial complexity of the feature compositions makes it hard to find valid configurations manually because of the large number of middleware configuration options and complex semantic relationships between them.

d. deployment- and run-time specializations are even harder because feature removal and additions need to be considered simultaneously, systematically and in a semantically consistent and coordinated manner such that domain-specified requirements on performance and footprint are satisfied.

In Section 2.2 we discussed how Aspect-oriented programming (AOP) [12] is extensively used for middleware specialization (*e.g.*, [11, 27, 19]). AOP, however, does not support any architectural model to define transformations to the structure of programs, particularly the ability to encapsulate new classes, which limits its suitability for middleware specialization. Feature-oriented programming (FOP) [20] on the other hand can represent single aspects or collections of aspects, and also can complement model-based development since both paradigms stress the importance of transformations in automated program development [3]. Moreover, FOP has better support to provide bounded (*i.e.*, selective) quantification for feature manipulation in contrast to AOP techniques which often result in unbounded quantification.

FOP is thus a candidate approach for middleware specializations since it involves manipulation of middleware features. FOP is best suited when the underlying construct on which it operates displays a well-defined algebraic structure. FOP for middleware specializations is not straightforward, however, due to a lack of an explicit algebraic structure in the middleware design as explained above. We therefore ask ourselves whether it is possible to impose an algebraic structure on the middleware. A closer scrutiny of the middleware design reveals that if we raise the level of abstraction [24] to the level of features the middleware offers instead of focusing on source code-level details, then a strong algebraic structure unfolds wherein features can be manipulated using the FOP paradigm subject to some constraints.

We have therefore chosen the principles of AHEAD (Algebraic Hierarchical Equations for Application Design) [4], which is an implementation of FOP that uses stepwise refinement to synthesize application product lines, as the basis of the proposed approach. The notion of a feature in AHEAD is tied to basic object-oriented programming concepts, such as classes and methods. Al-

though middleware also often uses object-oriented design principles, our notion of features is at a higher level of abstraction involving patterns and frameworks that provide properties, such as real-timeliness and fault tolerance.

AHEAD starts with a small set of base capabilities and refines them by incrementally adding features. In contrast our middleware specializations start with a much larger software base pruning unwanted features and customizing the needed ones with domain-specific properties. Our goal is to enhance AHEAD and similar research to support design, deployment and run-time feature manipulation.

3.3 Exploiting the Algebraic Structure of Middleware

We now lay down our initial ideas on our proposed approach to middleware specializations based on recursive algebraic approaches such as AHEAD, however, by operating at a level of abstraction for features that is closer to patterns and frameworks, and across all stages of the application lifecycle.

Table 1 depicts our attempt to capture the algebraic structure of RTCORBA capabilities as features within an Origami matrix as proposed by the AHEAD approach [4] with the difference that our level of abstraction for features is different and we consider all stages of application lifecycle. Origami is a generalization of binary decision matrices, where matrix axes define different sets of features, and matrix entries define feature interactions. Origami matrices possess a special property in that they allow folding along the rows or columns or both. We discuss how this property will be used.

Base \ RT	BasicRT	Priority	Conc	Synch
ORB	RTORB	PriMapper	TPReactor	
POA	RTPOA			
Xport	ExtXport	BandConn		
ReqHndl		CLI_PROP	TPLane	MUTEX

Table 1: Origami Matrix for RTCORBA

We use rows to denote the basic CORBA features, such as the object request broker (ORB) that mediates requests and manages resources; the portable object adapter (POA) that manages object lifecycle; the Transport (shown as Xport) which handles communication; and ReqHandling which provides the data marshaling and handling of requests. The columns denote the real-time features that refine the basic features of CORBA with real-time capabilities. For example, BasicRT indicates the base capabilities that introduce real-time properties; Priority indicates the priority handling mechanisms; Concurrency and Synchronization are classical distributed computing properties and describe the RTCORBA mechanisms that support these.

The individual cells illustrate the feature interaction across the row and column. For example, the CLI_PROP cell indicates the priority model to be used in request handling. We assume that the RTORB shown in the top-left cell is the constant required by AHEAD. In reality, however, a single cell such as RTORB can itself be formed by its own nested Origami matrix where different features are composed to realize the notion of an RTORB. An empty cell indicates a composition *identity*, which does not change anything to the feature on which it is composed.

Now imagine a stepwise folding of columns onto each other, which in turn folds individual cells onto each other for all the rows. This cell-wise folding results in the composition of features of the folded cells. Table 2 depicts the folding of the third and fourth column in the original matrix. Continuing this folding along all columns and then rows (order does not matter) gives rise to a composition of features that constitutes the overall RTCORBA middleware and can be represented by Equation 1. Features are composed with each other using the composition operator •.

Base \ RT	BasicRT	Priority • Conc	Synch
ORB	RTORB	PriMapper • TPReactor	
POA	RTPOA		
Xport	ExtXport	BandConn	
ReqHndl		CLI_PROP • TPLane	MUTEX

Table 2: Folded Matrix for RTCORBA

$$RTCORBA = MUTEX \bullet TPLane \bullet CLI\_PROP \bullet BandConn \bullet ExtXport \bullet TPReactor \bullet PriMapper \bullet RTPOA \bullet RTORB \quad (1)$$

Now let us explore how such equations will help us. Our previous work [13] on handcrafted middleware specialization has showed how the RTCORBA middleware stack characterized by Equation 1, forces the software components of our avionics mission computing scenario to use all the features, many of which are sources of excess generality. We claim that an approach to prune unwanted features can follow a similar folding operations of the Origami matrix that produces an equation of features to be pruned (*e.g.*, bypassing the request demultiplexing logic) and customized (*e.g.*, caching requests). This can be attempted by the application developer or middleware developers who are given the requirements by domain experts. The algebraic difference between the RTCORBA equation and the equation describing the excess generality provides a formal approach to specializing middleware.

Notice how this proposed approach is no longer *ad hoc* unlike handcrafted specializations. This desired property stems from the significant benefit of an Origami matrix in that it can realize only valid compositions of features. Notice that erroneous compositions (*e.g.* folding along the diagonal) or differences are impossible due to the constraints imposed by the folding capability of the Origami matrix. A model-based tool can provide an approach to collect all the domain requirements, which then can be used to drive the Origami folding and synthesis of the different equations.

3.4 Feature Manipulations across Application Lifecycle Stages

The composition operator • is part of a well-defined algebra [2], which to our knowledge works only for design-time feature composition. AHEAD (and hence Origami) does not support deployment- and run-time feature manipulation. We argue for new research in enhancing existing SPL research, such as AHEAD, to include deployment- and run-time phases of application lifecycle. Applying AHEAD principles to cover all the stages of application lifecycle is hard however because the level of abstraction it operates at (*e.g.*, code level) is not suitable for feature manipulations in the deployment- and run-time stages, and it is conceivable that the existing feature algebra will be incompatible at these stages.

We make an initial attempt to enhance this theory. Imagine a third dimension added to Table 1, which defines the deployment dimension. We can visualize this scenario as comprising multiple planes each having its own Table 1, where each table corresponds to the middleware specialization for the hosted component of the product variant. Suppose that the deployment of the product variant must ensure that the middleware is specialized for the CLIENT_PROPAGATED priority model. Now suppose that one

such table uses a SERVER_DECLARED priority model for request handling instead. We need an approach by which the folding operation along the third dimension should throw an exception due to a misconfiguration in one of the matrices. Run-time issues such as adding features for, say, a coordination layer for fault management can be handled by extending the Origami matrix in the fourth dimension to cover these run-time issues.

3.5 Feature Interactions across Application Lifecycle Stages

Our discussions so far have assumed that features are independent of each other and that they can be seamlessly added or pruned. However, we cannot make such simplified assumptions in all cases. For example, Narasimhan et. al. [16] have illustrated how real-time and fault-tolerance properties of applications conflict with each other thereby requiring tradeoffs.

Figures 5 and 6 illustrates how features can interact [14] with each other at the framework level (which is our level of abstraction for features). We show two design possibilities for an RTORB that supports thread pools with lanes. The thread pool serves as an additive refinement to the RTORB (*i.e.*, an Introduction). However, as shown in Figures 5 and 6, the request handling strategy interacts with the RTORB in different ways each with its benefits and consequences on performance.

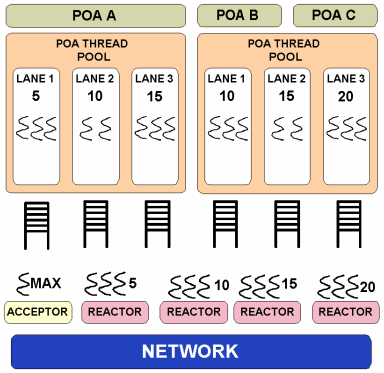

Figure 5: Queue-per-lane Design for Threadpool-with-Lane

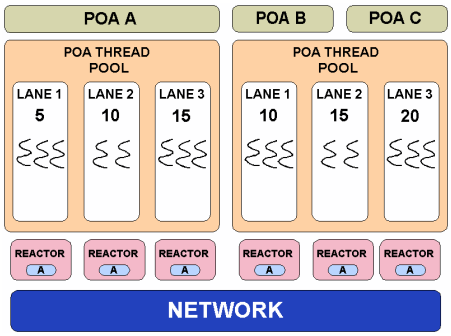

Figure 6: Reactor-per-lane Design for Threadpool-with-Lane

In the `queue_per_lane` strategy, a separate thread listens for requests over the network and hands over the request to a worker thread, which is the Half-Sync/Half-Async architectural pattern [22]. This model simplifies the design but incurs message queuing and thread synchronization overhead. In the `reactor_per_lane` approach, the thread that receives the request also handles the request, which is the Leader-Follower architectural pattern [22]. This model is difficult to implement and debug for race conditions.

We showed design-time feature interactions above, which is a hard problem since our features represent patterns and frameworks. Feature interactions at other lifecycle stages are even harder to address. Simple foldings of Origami columns and rows may not suffice since the foldings have no capabilities to tradeoff one feature over the other as in the case of fault tolerance and real-time. Traditionally the tradeoff problems have been mapped to combinatorial optimization problems where heuristics are developed to find near-optimal solutions.

4. CONCLUDING REMARKS AND OPEN ISSUES

Middleware is an important asset of SPLs that operate in a distributed computing environment. In this paper we argued for extending SPL research to incorporate middleware specializations. We showed how an algebraic structure can be imposed on the middleware which in turn makes it suitable for feature manipulation. We then explored the use of generative programming and modularization techniques based on AHEAD for middleware specialization outlining how they can be extended to address deployment- and run-time issues in middleware.

A number of open issues remain unresolved as explained below.

- *Mapping higher-level feature abstractions to code:* Since the algebraic structure we consider is at a higher level of abstraction, we require a mappping from the high level artifacts to low level details such as code. Naturally, such a mapping cannot break existing code. Hence we will need out-of-band mechanisms such as source code annotations including those we developed in our preliminary work [13] or aspect definitions to refactor existing middleware into the algebraic form we require.
- *Semantics of the composition and difference operator for deployment- and run-time phases of the application lifecycle:* Is a single equation feasible that can capture the specializations to middleware by accounting all three phases of application lifecycle. An important open issue points to the algebra of these operators across the lifecycle. A number of questions must be answered: What is the associativity and precedence relationship of the operators along the lifecycle stages? Do the semantics of Origami folding change in different lifecycle stages? How are features represented at the other lifecycle stages? How can Origami folding handle distributed coordination at run-time? Can Origami capture system schedulability and performance optimizations?
- *Runtime Tradeoffs via Origami foldings:* Adaptive systems must make runtime tradeoffs among inherently conflicting system properties such as real-timeliness and fault-tolerance. Many questions must be answered if Origami abstractions are used to solve these challenges: Do individual cells encode constraints? Do foldings give rise to cost functions? How do constraints get refined during folding? What does the final equation represent? Does the composition operator encode a heuristic to solve the optimization problem? How can feature manipulations be considered simultaneously, systematically and in a semantically consistent and coordinated manner such that domain-specified requirements on QoS and footprint are satisfied?

5. REFERENCES

[1] G. Ahlforn and E. Örnulf. Ericsson's Family of Carrier-class Technologies. *Ericsson Review*, 4:190–195, Apr. 2001.

[2] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 36–50. LNCS vol. 5140, Springer-Verlag, 2008.

[3] D. Batory. Using Modern Mathematics as an FOSD Modeling Language. In *To Appear in the Proceedings of the Generative Programming and Component Engineering (GPCE 08)*, New York, NY, USA, 2008. ACM.

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

[5] 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*, pages 191–206, London, 1998. Springer-Verlag.

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

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[8] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

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

[10] 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*, pages 411–428, New York, NY, USA, 1993. ACM.

[11] 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. ACM Press, 2002.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[13] 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*, pages 205–218, Leuven, Belgium, Apr. 2006.

[14] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121. ACM Press New York, NY, USA, 2006.

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

[16] P. Narasimhan. Trade-Offs Between Real-Time and Fault Tolerance for Middleware Applications. Workshop on Foundations of Middleware Technologies, Nov. 2002.

[17] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

[18] Object Management Group. *Real-time CORBA Specification*, 1.2 edition, Jan. 2005.

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

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

[21] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, Dec. 1993. SUG.

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

[23] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions*, volume 576, pages 353–370, Aug 2000.

[24] J. A. Stankovic, P. Nagaraddi, Z. Yu, Z. He, and B. Ellis. Exploiting Prescriptive Aspects: A Design time Capability. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 165–174, New York, NY, USA, 2004. ACM Press.

[25] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the International Conference on Software Engineering*, pages 107–119, May 1999.

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

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

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