An Approach to Middleware Specialization for Cyber Physical Systems

Akshay Dabholkar and Aniruddha Gokhale
Dept. of EECS, Vanderbilt University
Nashville, TN 37235, USA
{aky,gokhale}@dre.vanderbilt.edu

Abstract

Contemporary computing infrastructure, such as networking stacks, OS and middleware, are made up of layers of software functionality that have evolved over decades to support the broadest range of applications. The feature-richness and the layers of functionality, however, tend to be excessive and a source of performance overhead for Cyber-physical Systems (CPS). Yet it is necessary to leverage the decades of proven patterns and principles in these infrastructures. This paper presents an approach to systematically specialize general-purpose middleware used to host CPS. Our approach is based on the principles of Feature-Oriented Software Development (FOSD), which requires deducing an algebraic structure of contemporary middleware based on a higher level of abstraction of features. The paper showcase how Origami matrices and generative programming can play a key role in realizing the specializations. The paper concludes by delving in to future open areas of middleware specialization research.

Keywords: *Architecture and infrastructure, FOSD, middleware specialization, features, annotations.*

1 Introduction

Research in middleware over the past decade [3, 17, 21] has significantly advanced the quality and feature-richness of general-purpose middleware, such as J2EE, .NET, CORBA, and DDS. Middleware serves as the backbone for applications across many domains that have significant societal impact including electronic medical records in health care, air traffic control in transportation, industrial automation, among many others. The economic benefits of middleware are significant with up to 50% decrease reported in software development time and costs [15].

Despite these benefits, general-purpose middleware poses numerous challenges when developing Cyber-Physical Systems (CPS). First, owing to the stringent demands of CPS on quality of service (QoS) (*e.g.* real-time response in industrial automation) and/or constraints on resources (*e.g.* memory footprint of embedded medical devices monitoring a patient), the feature-richness and flexibility of general-purpose middleware becomes a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains and/or energy savings.

Second, general-purpose middleware lack *out of the box* support for modular extensibility of both domain-specific and domain-independent features within the middleware without unduly expending extensive manual efforts at retrofitting these capabilities. For example, CPS in two different domains as in industrial automation and automotive may require different forms of domain-specific fault tolerance and failover support. Arguably, it is not feasible for general-purpose middleware developers to have accounted for these domain-specific requirements ahead-of-time in their design. Doing so would in fact contradict the design goals of middleware, which aim to make them broadly applicable to a wide range of domains, i.e., general-purpose.

Developing proprietary and customized middleware solutions for individual CPS is not a feasible alternative due to the excessive costs of development, maintenance and testing. Moreover, such solutions often tend to reinvent many solutions that already exist in general-purpose middleware. Current trends and economies of scale in software development actually call for extensive reuse and rapid assembly of application functionality from off-the-shelf infrastructure and application components.

Addressing this dilemma requires an approach that will enable CPS developers to derive the benefits of general-purpose middleware, however, without incurring the overhead of unwanted features while seamlessly allowing domain-specific extensions. Such an approach must be rooted in scientific principles, which is

2 Motivating CPS Example for Middleware Specialization

We use a CPS case study to highlight the inherent structural and design constraints of middleware that makes the middleware specialization problem hard.

2.1 Reconfigurable Conveyor Belt System: A CPS Case Study

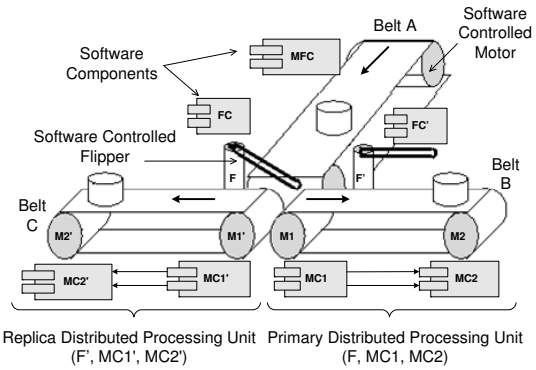

Figure 1: **Reconfigurable Conveyor Belt CPS**

Figure 1 represents a portion of a reconfigurable conveyor belt system found in CPS domains such as industrial automation. A material flow control (MFC) component directs a part using the route BELT A→BELT B or the route BELT A→BELT C. A flipper F and F′ assist in using BELT B and BELT C from BELT A, respectively. Further, hardware interface layer components, such as Motor Controllers (MC1, MC2) and the Flipper Controller (FC), control the belt motors and flippers, respectively. The MFC component instructs the Flipper Controller component to flip, which in turn instructs the Motor Controller components to start the motors and begin moving the parts. The MFC component uses the route BELT A→BELT C only if BELT B fails or gets jammed because of overloads or stuck parts.

2.2 Need for Middleware Specialization in CPS Case Study

We view the middleware used to support our case study (and in general other CPS) as made up of layers of features targeted to perform specific activities. For our case study we use real-time CORBA (RTCORBA) [13], which is compliant with our layered middleware system model. CORBA is used here only for illustration purpose. RTCORBA features shown in Figure 2 define 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.

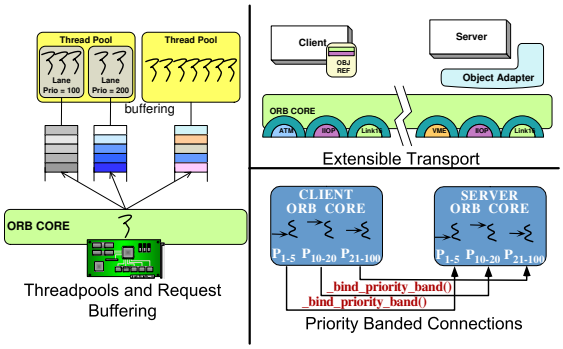

Figure 2: **RTCORBA Features**

The software components of the case study use multiple features of RTCORBA. For example, the extensible transport feature is used to leverage a proprietary signaling protocol for message communication instead of TCP/IP. The `CLIENT_PROPAGATED` priority propagation model is used for requests made between the software components. These requests comprise commands for motor start/stop, motor speed change, and flip activate, which must be handled at different priorities by the thread pool-with-lanes feature.

Despite the richness of the real-time feature set in RTCORBA, these features tend to be excessive for our case study. For example, we do not envision requests to get buffered. Similarly, we use a particular signaling protocol yet our software components must pay the price both in memory footprint and performance to use the extensible transport mechanism. These properties are known at design-time.

When the system is deployed we must ensure that the same middleware configurations exist across the software components. Based on *ahead-of-time* known properties, we can also determine different run-time overheads imposed by the middleware. For example, our system supports a known number of requests with known number and values for the parameters, *e.g.*, motor speeds. Similarly, the CORBA objects that implement the logic are also statically created and remain invariant over the lifecycle.

Despite knowing these properties, the layered architecture and design of the middleware forces requests to be dynamically created and deallocated, incur unwanted marshaling, and pass through several layers of request demultiplexing logic. On the other hand, some needed capabilities including a coordination layer between components (to stop all motors simultaneously) is not provided by RTCORBA requiring complex engineering efforts to integrate new functionality in the middleware.

2.3 Challenges in Middleware Specialization

Addressing these domain-imposed requirements is hard for the following limitations of middleware:
(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 price and is hard to break to satisfy the performance and footprint requirements of the domain.
(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 hard 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.

3 Related Research

Despite their often object-oriented approaches, different middleware are not designed [7] with the aim of allowing fine grained specializations. Thus, many existing specialization efforts discussed below rely on handcrafting these solutions. Handcrafted solutions, however, often tend to address only a specific need making it hard to generalize the solution approach.
• *Eliminating overhead of object-orientation:* Lohmann et. al. [12] argue that fine-grained and resource-efficient embedded system software requires a means for separation of concerns that does not lead to extra overhead in terms of memory and performance. Aspect-oriented programming (AOP) [9] is shown to eliminate this overhead.
• *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. For example, *FACET* [6] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects that can be woven in when needed.
• *Combining modeling and aspects for refinement:* The *Modelware* [19] project combines model-driven architecture (MDA) with AOP. Modelware advocates the use of models and views to separate intrinsic (*i.e.*, essential) functionalities of middleware from extrinsic ones (*i.e.*, optional). As in FACET, AOP is used to weave in different aspects, however, driven by a modeling tool.
• *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. 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 [14] have shown how AOP can be used to bypass middleware layers.
• *Just-in-time specializations:* Edicts [4] 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 [20] shows how middleware can be customized after application characteristics are known. Research, such as Edicts, 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.

4 FOSD Approach to Middleware Specialization

Aspect-oriented programming (AOP) [9] is shown to be extensively used for middleware specialization [6, 19, 14]. However, AOP has a limited architectural model to define transformations to the structure of programs, particularly the ability to encapsulate new classes, which limits its suitability for middleware specialization. FOSD [16] which can represent single aspects or collections of aspects, complements model-based development (which is preferred in CPS development), as both paradigms stress the importance of transformations in automated program development. Moreover, FOSD supports bounded (*i.e.*, selective) quantification for feature manipulation in contrast to AOP techniques which provide unbounded quantification.

FOSD is thus a candidate approach for middleware specializations since it involves manipulation of middleware features. FOSD is best suited when the underlying construct on which it operates displays a well-defined algebraic structure. FOSD for middleware specializations is not straightforward, however, due to a lack of an explicit algebraic structure in the middleware design as explained in Section 2.

4.1 An Algebraic Structure for Contemporary Middleware

We ask ourselves whether it is possible to impose an algebraic structure on contemporary middleware so that it is amenable to FOSD-based specialization. A closer scrutiny of the middleware design reveals that if we raise the level of abstraction [18] to the level of features the middleware offers instead of focusing on source code-level details (which is the traditional view point of FOSD), then a strong alge-

braic structure unfolds wherein features can be manipulated using the FOSD paradigm subject to some constraints.

We have chosen the principles of AHEAD (Algebraic Hierarchical Equations for Application Design) [2], which is an implementation of FOSD that uses stepwise refinement to synthesize application and their families, as the basis of our approach. Static analysis and Partial Evaluation [5, 1] are potential alternatives to AHEAD though these alternatives are useful only in design-time specialization.

The notion of a feature in AHEAD is tied to basic object-oriented programming concepts, such as classes and methods. Although middleware also often uses object-oriented design principles, the level of abstraction for features that we are interested in is at a higher level of abstraction involving frameworks and class libraries that provide properties, such as real-timeliness and fault tolerance. Moreover, while AHEAD starts with a small set of base capabilities and refines them by incrementally adding features, our middleware specializations start with a much larger software base pruning unwanted features and customizing the needed ones with domain-specific properties.

4.2 Exploiting the Algebraic Structure

Once an algebraic structure is imposed on the middleware, the next step is to map the specialization problem into AHEAD's FOSD approach. AHEAD provides a tool called an Origami matrix to drive feature compositions. 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.

Table 1 depicts our initial attempt to capture the algebraic structure of RTCORBA capabilities we deduced as features within an Origami matrix. Note that our level of abstraction for features is different that the traditional view of AHEAD.

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 classic distributed computing properties and describe the RTCORBA mechanisms that support these.

The individual cells illustrate the feature interactions 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.

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

Table 2: **Origami Matrix for RTCORBA**

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 •.

$$\begin{aligned} RTCORBA \quad = \quad & MUTEX \bullet TPLane \bullet CLI\_PROP \bullet BandConn \\ & \bullet ExtXport \bullet TPReactor \bullet PriMapper \\ & \bullet RTPOA \bullet RTORB \end{aligned} \tag{1}$$

Now let us explore how such equations will help us. Since our case study uses RTCORBA, its middleware stack is characterized by Equation 1, which means that the software components use all the features, many of which are sources of excess generality. 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 representation of the specialized middleware.

The Origami matrix mechanism eliminates any errors in feature manipulations since 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. Origami matrices can be multi-dimensional with folding along any number of dimensions.

4.3 Mapping the Equations to Software

The final step in realizing the specializations is to map the equations that represent the specializations into middleware code-level program transformations and configuration management. Our earlier work [10, 8] in this problem space has developed program transformation capabilities using Perl-based and AOP-based techniques.

For example, in [10], we developed a tool called FOCUS that requires annotation of source code using special markers placed within programming language comments. A set of directives that perform operations such as copy, replace, comment, etc are provided in a script file, which is then piped to the FOCUS tool. Using these directives FOCUS transforms the source code. The equation 1 translates to the following type of FOCUS specializations:
(1) Memoization for request creation: The equation already described the type of request to be created because of which request headers could be cached instead of being created and determined repeatedly for each request handshake.
(2) Layer Folding for request dispatch resolution: Since it was already determined that the way requests are to be handled at client priority using thread pool lanes and mutexes, request dispatch resolution could be easily bypassed thereby bypassing/folding middleware layers.
(3) Aspect Weaving for framework generalities such as POA and ORB: The equation specifies to use the ORB and POA specialized already for real time which are incorporated into the code through aspect weaving FOCUS directives that replace the generalized CORBA framework components which specialized ones i.e., RTORB and RTPOA.

5 Concluding Remarks and Open Research Issues

Current trends and market economies require that general-purpose middleware be used to develop Cyber-Physical Systems. However, the lack of fine granularity of modularization in their design make general-purpose middleware heavyweight solutions for CPS and are detrimental to their performance. This paper presents preliminary ideas on a systematic approach to specializing general-purpose middleware wherein middleware can be seamlessly manipulated by (a) addition of custom features, (b) pruning of unwanted features, and (c) optimizing the required infrastructure.

Our approach is based on the principles of feature-oriented software development (FOSD), which required us to deduce an algebraic structure for contemporary middleware based on a higher level of abstraction for features compared to traditional approaches that treat classes as features. We showed how Origami matrices and generative programming can play a key role in realizing the specializations.

A number of open research issues remain unresolved shown below:
(1) How are features manipulated across different stages of application lifecycle? Achieving the goals of specialization is challenging since the process must account for the design-, deployment- and run-time requirements imposed by a domain. For example, in our conveyor belt system case study, specializations must account for variations in domain-specific requirements, such as the number of components in a failover unit – a design-time issue that determines how many components will need the specialization; the topology of the failover unit – a deployment-time issue which requires that middleware be specialized in a similar manner across the failover unit; and the fault masking and failover strategy for the client components that lie outside the unit – a run-time issue that requires client-side middleware specialization to deal with faults.
(2) How do we handle feature interactions? Features are often known to interact [11] 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. Naturally, any *ad hoc* process will not produce the correct results nor will it work across different domains.
(3) Can we formalize the theory as a software engineering process? Theoretical underpinnings are a necessary but not sufficient condition to realize the benefits of research. Concrete mechanisms, such as tools and processes, that realize the theory are required. In our case, without the right tools and processes, specializations will continue to be carried out manually, which is detrimental towards verifying the correctness of CPS.
(4) How should next-generation midddleware be designed? Despite the object-oriented designs of contemporary middleware solutions, these are deemed too coarse-grained when fine-grained specializations are necessary, particularly for CPS. For our current approach, we were required to impose an approximate algebraic structure to contemporary middleware. New research directions are needed to explore how next generation middleware be designed such that they inherently maintain an algebraic structure that are seamlessly amenable to feature manipulations.
(4) How to efficiently annotate middleware source code

for feature identification and management? There is not only a need to systematically design middleware ground-up but also a need to refactor contemporary middleware for feature pruning/augmentation. This can only be achieved by devising efficient advanced annotations that identify middleware features, their dependencies and interactions. Moreover, semi-automated tools need to be developed that can leverage these feature annotations based on which the unnecessary features can be pruned in order to aid the construction of specialized middleware.

References

[1] Anne-Francoise Le Meur, Julia L. Lawall and Charles Consel. Specialization Scenarios: A Pragmatic Approach to Declaring Specialization Scenarios. *Higher-Order and Symbolic Computation*, 17(1), March 2004.

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

[3] 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.

[4] 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.

[5] J. Hatcliff. An Introduction to Online and Offline Partial Evaluation using a Simple Flowchart Language. *Partial Evaluation – Practice and Theory DIKU 1998 International Summer School, Springer Verlag*, 1706:20–82, June 1998.

[6] 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.

[7] 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.

[8] D. Kaul and A. Gokhale. *Automating Middleware Configurations and Specializations: A Modeling Language Approach*. VDM Verlag Dr. Muller, Jan. 2008.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] T. Pearson. Save time and money with COTS middleware for network equipment. `www.commsdesign.com/printableArticle/?articleID=174402378`, Nov. 2005.

[16] 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.

[17] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[18] 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.

[19] 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.

[20] 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.

[21] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.