Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems

Arvind S. Krishna[†], Aniruddha Gokhale[†] and Douglas C. Schmidt[†],
Venkatesh Prasad Ranganath[‡] and John Hatcliff[‡]
[†]Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN
[‡]Dept. of Computing and Information Sciences, Kansas State University

Abstract

Product-line architectures (PLA)s are an emerging paradigm for developing software families for distributed real-time and embedded (DRE) systems by customizing reusable artifacts, rather than handcrafting software from scratch. To reduce the effort of developing software PLAs and product variants for DRE systems, it is common to leverage general-purpose – ideally standard – middleware platforms whose reusable services and mechanisms support a range of application quality of service (QoS) requirements, such as predictability and low end-to-end latency. While standard middleware provides generality and flexibility to support many types of PLAs and product variants, standard middleware implementations often incur unnecessary footprint overhead and lack optimizations needed to meet the QoS needs of PLAs and product variants for DRE systems. This paper describes systematic model driven development (MDD) techniques for specializing implementations of standards-based, general-purpose middleware to support the application-specific QoS needs of different product variants created atop a PLA. Our preliminary results show that implementations of standard middleware can be specialized transparently to better meet the QoS needs of PLAs and product variants, without compromising standards compliance.

1 Introduction

Software development organizations must innovate rapidly, provide capabilities that meet user needs, and sustain their competitive advantage in the face of time-to-market pressures and limited software resources. As a result, many organizations are trying to reuse existing artifacts and resources for a range of products, rather than handcrafting software for each product from scratch. A promising technology for systematically addressing the challenges of large-scale mission-critical software systems is *product-line architectures* (PLAs) [1].

In contrast to conventional software processes that produce separate point solutions, PLA-based processes develop families of *product variants* [2] that share a common set of capabilities, patterns, and architectural styles. PLAs can be characterized using *scope, commonality, and variabilities* (SCV) analysis [3], which is an engineering process that identifies the scope of the product families in an application domain and then determines the common and variable properties among them. Domain/systems engineers and software architects use SCV analysis to guide decisions about where and how to address possible variability and where the common development strategies can be used.

PLAs have been developed and applied to a variety of domains [4]; this paper focuses on applying PLAs to *distributed, real-time and embedded (DRE) systems* [5]. Examples of DRE systems include applications with hard real-time requirements, such as avionics mission computing [6], as well as those with softer real-time requirements, such as telecommunication call processing and streaming video [7]. These DRE systems are characterized by their stringent QoS requirements (such as low memory footprint, power consumption, latency, and predictability), which often makes them harder to develop, maintain, and evolve than conventional desktop and enterprise software.

The QoS challenges of DRE systems have hitherto led developers to (re)invent custom applications that are tightly coupled to specific hardware/software platforms, which is tedious, error-prone, and costly to evolve over product lifecycles. During the past decade, therefore, a key technology for alleviating the tight coupling between applications and their underlying platforms has been *middleware* [8], which (1) functionally bridges the gap between applications and platforms, (2) controls many aspects of end-to-end QoS, and (3) simplifies the integration of components developed by multiple technology suppliers.

Although middleware has been used successfully in DRE systems [5, 6], key challenges must be overcome before it can be applied seamlessly to support the QoS needs of *PLA-based* DRE systems. In particular, there is a need for R&D to resolve the tension between: (1) the *general-*

ity of standards-based middleware, which benefits from a reusable architecture designed to satisfy a broad range of application requirements and (2) *application-specific product variants*, which benefit from highly-optimized, custom PLA middleware implementations. In resolving this tension, solutions should retain the portability and interoperability capabilities provided by standard middleware.

The remainder of the paper is organized as follows: Section 2 identifies key middleware optimization challenges pertaining to PLA-based DRE systems; Section 3 explains how we plan to apply model-driven sepcialization to middleware to address these challenges; Section 4 compares our work with related research; and Section 5 presents concluding remarks.

2 Middleware Optimization Challenges for PLA-based DRE Systems

This section uses a representative DRE system scenario to identify common types of excessive generality in middleware for PLAs and outlines how context-specific specialization techniques help to alleviate this generality without compromising standards compliance. Figure 1 illustrates key middleware optimization challenges associated with the tension between (1) *application-specific product variants*, which require highly-optimized and customized PLA middleware implementations and (2) *general-purpose, standards-based middleware*, which is multi-use and thus designed to satisfy a broad range of application requirements. Resolving this tension is essential to en-

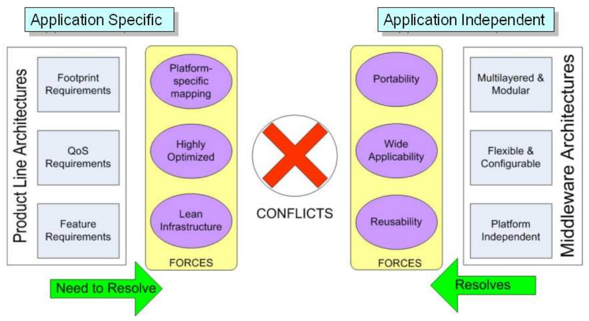


Figure 1. Application-specific vs. Application-independent Dimensions of PLAs and Middleware

sure that middleware can support the QoS requirements of PLA-based DRE systems. Unfortunately, implementations of standards-based QoS-enabled middleware technologies, such as Real-time CORBA [9] and Real-time Java [10], can be excessively general and thus poorly suited to support PLAs for DRE systems due to excessive time/space overhead for product variants. Removing unwanted generality, such as redundant functionality, navigation of multiple layers, and unnecessary checks, is therefore essential to optimize general-purpose middleware implementations to meet the QoS requirements of product variants. In turn, removing this generality helps identify opportunities for further optimizations, while still maintaining standard interfaces and interoperability protocols.

2.1 PLA DRE Scenario Case Study

The remainder of this section uses a concise, yet representative, DRE PLA scenario to (1) illustrate how the challenges outlined above occur in practice and (2) identify system invariants that drive our specialization approach. The scenario is based on the Boeing Bold Stroke avionics mission computing PLA [11], which supports the Boeing family of aircraft, including many product variants, such as F/A-18E, F/A-18F, F-15E, F-15K, etc. Bold Stroke is a component-based, publish/subscribe platform built atop *TAO* [12], which is a standards-based Real-time CORBA [9] implementation, and *Prism* [6], which is Qos-enabled component middleware influenced by the Lightweight CORBA Component Model (CCM) [13].

Figure 2 illustrates the *BasicSP* application scenario that is the focus of this paper and is representative of rate-based DRE systems in avionics, vetronics, and process control. This scenario involves four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays. Communication between components uses the event-push/data-pull model, with data producing components pushing an event to notify new data is available and data consuming components pulling data from the source. A `Timer` component pulses a `GPS` navigation sensor component at a certain rate, which in turn publishes `data_avail` events to an `Airframe` component. Aware that new data is available, this component then calls a method provided by the `Read_Data` interface of the `GPS` component to retrieve current location. After formatting the data, `Airframe` sends a `data_avail` event to the `Nav_Display` component, which then pulls the location and velocity data from the `Airframe` component and displays this information on the pilot's heads-up display.

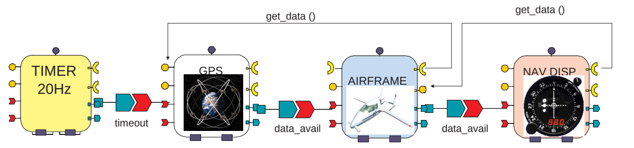


Figure 2. *BasicSP* Application Scenario

The *BasicSP* scenario illustrates the following commonalities and variabilities in the Bold Stroke PLA:

- **Commonalities** include the set of reusable components (such as `Display`, `Airframe`, and `GPS`) in Bold Stroke and middleware capabilities (such as connection management, data transfer, concurrency, synchronization, (de)marshaling, (de)multiplexing, and error-handling) that occur in all product variants and
- **Variabilities** include application-specific component connections (such as how `GPS` and `Airframe` components are connected in an F/A-18E vs. an F-15K), different implementations (such as whether GPS or inertial navigation algorithms are used), and components specific to particular customers (such as restrictions on this use of encryption in certain countries).

The *BasicSP* scenario shown in Figure 2 requests new inputs from the `GPS` component and updates the display outputs with new aircraft position data at a rate of 20 Hz. The time to process the inputs to the system and present the output to cockpit displays should therefore be less than a single 20 Hz frame. The rates at which these components interact is yet another variability that could change in different product variants.

2.2 Common Types of Excessive Generality in Middleware

We have thus far identified five common types of excessive generality in middleware relevant to PLA-based DRE system in general, and to the *BasicSP* scenario in particular, as shown in Figure 3. The challenges of each type of generality are discussed below:

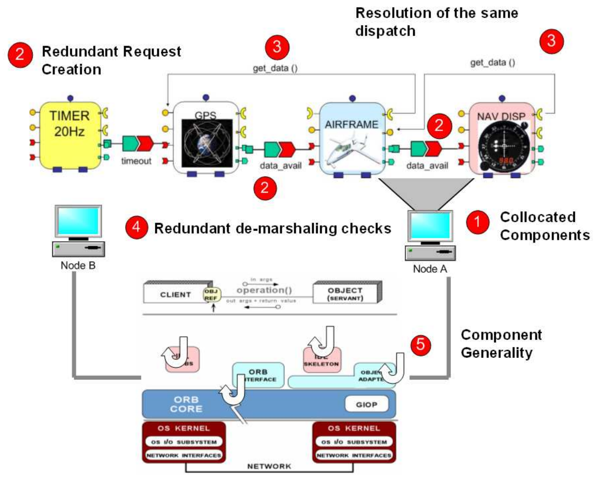


Figure 3. ***BasicSP*** **Specialization Points**

Challenge 1. Redundant remoting functionality for collocated objects. In PLA-based DRE systems, such as Bold Stroke, components are mapped to particular target nodes late in the design process, *e.g.*, during the deployment phase. Groups of components may be *collocated* (*i.e.*, placed on the same processor) to minimize network overhead in certain deployments. To optimize for this common case across different product variants or contexts within a single product, middleware implementations provide optimizations [14] that bypass the network and I/O subsystems when objects are collocated. Even for collocated deployments, however, standard middleware implementations often still contain code capable of performing *remoting*, which performs the (de)marshaling and framing logic used to send requests across a network.

A challenge is therefore to develop middleware specialization techniques that can identify redundant functionality (such as remoting code for collocated deployments) and remove this functionality from selected parts of the middleware for certain product variants or application-specific contexts.

Challenge 2. Redundant request creation and/or initialization. To send a request to the server, middleware implementations create a *request*, which contains buffer space to hold the header and payload information for each invocation. Rate-based DRE systems often repeatedly generate periodic events, such as timeouts that drive periodic system execution. Since most request information (such as message size, operation name, and service context) does not change across events, middleware implementations can use buffer caching [15] strategies to minimize request creation. This approach, however, can still incur the overhead of initializing the request header and payload for every request.

A challenge is therefore to develop middleware specialization techniques that can reuse pre-created requests (i.e., from previous invocations) partially and/or completely to avoid redundant initialization for certain product variants or application-specific contexts.

Challenge 3. Repeated resolution of the same request dispatch. To minimize the time/space overhead incurred by opening multiple connections to the same server, middleware often multiplexes requests on a single connection between client and server processes. Multiple client requests targeted for different request handlers in a server process are therefore received on the same multiplexed connection. Standard Real-time CORBA servers typically process a client request by navigating a series of middleware layers, *e.g.*, ORB core, object adapter(s), servant, and operation. To optimize request demultiplexing, Real-time CORBA implementations can combine active demultiplexing [16] and perfect-hashing [17] strategies to bound worst case lookup time to $O(1)$, irrespective of the nesting of the layers. This optimization, however, can still incur non-trivial overhead when navigating middleware layers and is redundant when the target request handler remains the same across different request invocations.

A challenge is therefore to develop middleware specialization techniques that avoid the expense of navigating layers of middleware to resolve the same request dispatch for certain product variants or application-specific contexts.

Challenge 4. Redundant (de)marshaling checks. PLA-based DRE systems may be deployed on platforms with different instruction set byte orders. To support seamless request processing irrespective of byte ordering, general-purpose Real-time CORBA implementations therefore use the general inter-ORB protocol (GIOP), which performs byte order tests when (de)marshaling requests/responses. These tests incur unnecessary overhead, however, when the nodes a DRE system runs on have the same byte order.

A challenge is therefore to develop middleware specialization techniques that evaluate ahead-of-time deployment properties to remove redundant (de)marshaling checks for certain product variants or application-specific contexts.

Challenge 5. Overly extensible middleware frameworks. Middleware is often developed as a set of frameworks that can be extended and configured with alternative implementations of key components, such as different types of transport protocols (*e.g.*, TCP/IP, VME, or shared memory), event demultiplexing mechanisms (*e.g.*, reactive-, proactive-, or thread-based), request demultiplexing strategies (*e.g.*, dynamic hashing, perfect hashing, or active demuxing), and concurrency models (*e.g.*, thread-per-connection, thread pool, or thread-per-request). A particular DRE product variant, however, may only use a small subset of the potential framework alternatives. As a result, general-purpose middleware may be *overly* extensible, *i.e.*, contain unnecessary overhead for indirection and dynamic dispatching that is not needed for use cases in a particular context.

A challenge is therefore to develop middleware specialization techniques that can eliminate unnecessary overhead associated with overly extensible middleware framework implementations for certain product variants or application-specific contexts.

To resolve the challenges presented above successfully, it is crucial that (1) middleware implementations be specialized without compromising their standard interfaces and (2) the effort of specifying and applying these specializations to production middleware and PLAs for DRE systems be minimized. Section 3 describes our approach to applying model-driven middleware specializations that alleviate the common types of excessive generality described above.

3 Model-driven Middleware Specialization: Vision & Approach

Customizing middleware for different operating contexts requires a systematic approach to designing and implementing different context-specific specializations. Model-driven Development (MDD) offers the right choice to realize a systematic and scientific approach to resolving the PLA middleware challenges described in Section 2. This paper describes our R&D that explores MDD techniques, which leverage Ahead of Time (AOT) known PLA and product-specific system properties, to specialize middleware and thereby improve middleware performance and footprint. For the MDD techniques to succeed we require the refactoring of middleware and services so they are amenable to automatic and dynamic customization and optimization by using model-driven middleware specialization patterns [18]. Our vision of the MDD approach to middleware specialization is shown in Figure 4.

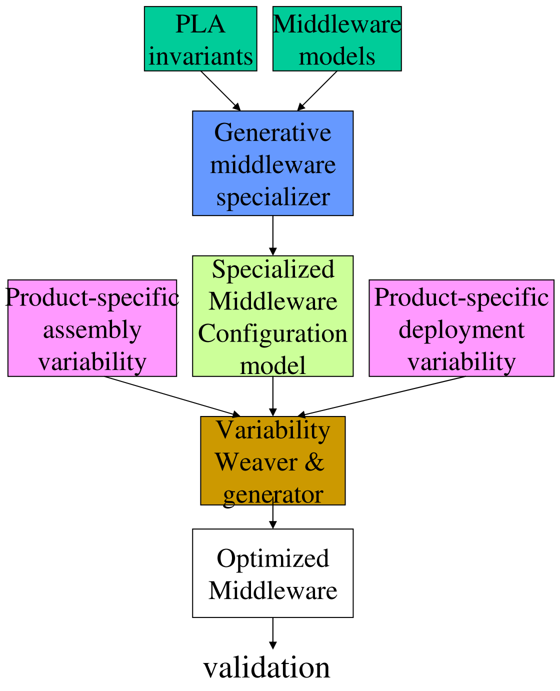

PLA invariants
Middleware models
Generative middleware specializer
Product-specific assembly variability
Specialized Middleware Configuration model
Product-specific deployment variability
Variability Weaver & generator
Optimized Middleware
validation

Figure 4. Model-driven Middleware Specialization

These specialization patterns are then mapped to middleware implementations via specialization tools, which are part of the generative tool chain of the MDD tools. The specialization process can be divided into two steps: (1) identification of the specialization points and transformations and (2) automating the delivery of the specializations. In the specialization identification phase, higher level models capture the system invariant information. This information captured is used in the second phase to automate the delivery of the specializations.

Development of this two step process requires first the development of the delivery phase to automate the specialization and the development of MDD tools to drive the specialization process. In the remainder of this section, we describe the Feature Oriented Customizer (FOCUS) MDD tool we are developing to automate the PLA middleware specializations.

3.1 FOCUS MDD Approach

This section describes our vision of the FOCUS MDD tools that will be required to drive the FOCUS middleware specializers described in Section 3.2. An MDD tool like FOCUS must respond to the following key requirements:

- **Defining and capturing PLA invariants** – which will require the modeling tool to provide capabilities to capture the PLA-specific feature invariants. Modeling tools will need to map these invariants to the features in middleware that will be required to satisfy these invariants.
- **Represent middleware models and their configurability** – which will require the modeling tool to provide capabilities to represent middleware as building blocks that can be configured and customized according to the PLA and product-specific requirements. This capability is driven by the state of art in middleware, which comprises a composition and configuration of patterns-based building blocks.
- **Capturing product-specific functional and QoS variability** – which will require the ability to specify product-specific variability incurred due to functional and QoS requirements. This will also govern the variability in the assembly, configuration and deployment of the product variant and the associated middleware infrastructure.

The modeling capabilities described above represent a multi-dimensional separation of concerns [19]. Each concern could be represented using higher levels of abstractions instead of low level code or platform-specific artifacts. The mapping of these concerns by the FOCUS tool to platform-specific artifacts represent the selected features of the middleware in terms of the chosen building blocks. Concerns that crosscut layers of middleware and building blocks are akin to aspects. Finally, the QoS requirements of product variants map to specializations of the selected features of the middleware. The remainder of this section describes the specialization automations carried out by FOCUS.

3.2 FOCUS Approach for Specialization Automation

Figure 5 illustrates the actors, tools and workflow in FOCUS. The different phases in the FOCUS approach can be broken down as follows:

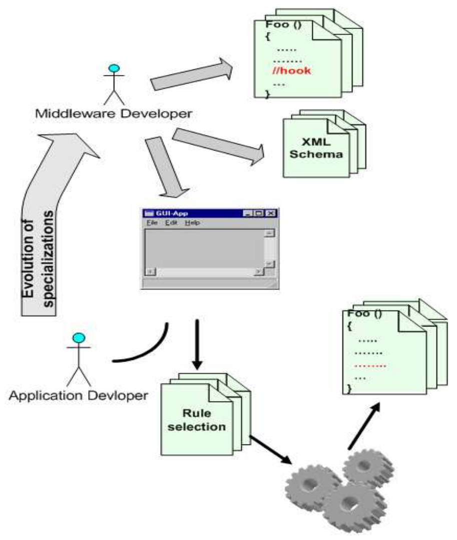


Figure 5. FOCUS Approach

- **Identification of specializations.** The most important step in the specialization process is the identification of specialization points to eliminate generality. Rather than selecting ad-hoc points, we examine the critical request/response processing path within middleware to systematically identify sources for specialization, as shown in Figure 6.

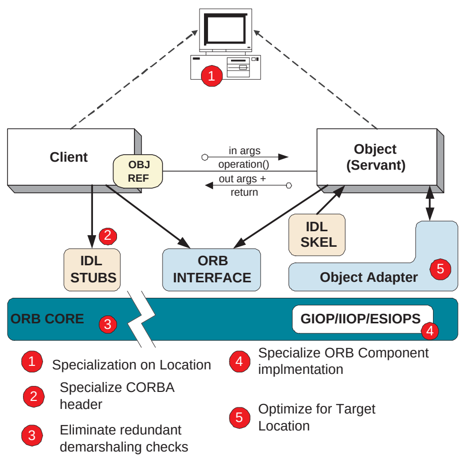
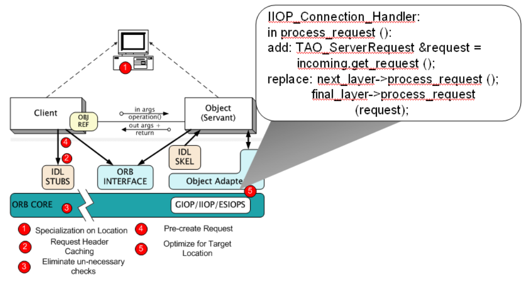

1
Client
OBJ REF
in args
operation()
out args + return
Object (Servant)
2
IDL STUBS
ORB INTERFACE
IDL SKEL
Object Adapter
5
ORB CORE
3
GIOP/IIOP/ESIOPS
4
1 Specialization on Location
2 Specialize CORBA header
3 Eliminate redundant demarshaling checks
4 Specialize ORB Component implmentation
5 Optimize for Target Location

Figure 6. FOCUS: Specialization Identification

Identification of these points have the greatest potential for increasing performance improvement. For example, middleware implementations run on top of multiple protocol implementations such as TCP/IP, UDP and shared memory. To seamlessly support and add new protocol implementations, middleware implementations are designed using the strategy [20] and template method pattern [20] that allow the different protocol implementations to be dynamically loaded within the middleware. These designs incur indirections and dynamic dispatching in middleware components along the critical request/response processing path.

• **Capturing specializations as rules.** Figure 7 illustrates how specializations are expressed as rules. In this phase, a middleware developer lays down the specialization rule required to transform general-purpose middleware into optimized middleware stack. These directly stem from the specialization points identified in the previous step.

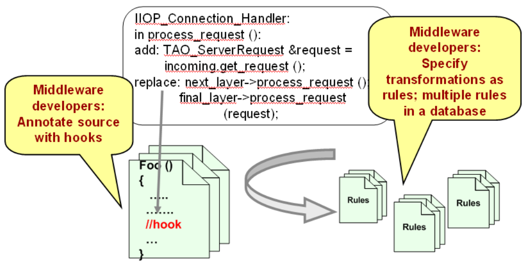


```
IIOP_Connection_Handler:
in process_request ():
add: TAO_ServerRequest &request =
        incoming.get_request ();
replace: next_layer->process_request ();
           final_layer->process_request
                     (request);
```

Figure 7. FOCUS: Capturing Specialization Rules

For example, challenge 3 in Section 2.2 describing the unnecessary demultiplexing of incoming requests for known target objects or challenge 2 describing unnecessary request creation can be resolved via capturing specializations as rules, which provide directives to the FOCUS tool to eliminate these sources of overhead from the source code as well as during runtime.

• **Middleware Annotation.** Figure 8 shows how the rules are used to annotate the middleware. In conjunction with capturing the specialization rules, the middleware developer annotates the middleware source with specialization hooks. These hooks are inserted as comments in the source code that do not interfere with the normal request/response processing. However, in the specialized mode, these hooks are used to weave in specialized code using a customizer engine.



```
IIOP_Connection_Handler:
in process_request ():
add: TAO_ServerRequest &request =
        incoming.get_request ();
replace: next_layer->process_request ();
           final_layer->process_request
                     (request);
```

```
Foo ()
{
  .....
  .......
  //hook
  ...
}
```

Middleware developers: Annotate source with hooks
Middleware developers: Specify transformations as rules: multiple rules in a database
Rules
Rules
Rules

Figure 8. FOCUS: Middleware Annotation

For example, challenge 1 in Section 2.2 describing the redundant remoting functionality for collocated objects can be resolved via annotating middleware thereby allowing the FOCUS transformation tools to strip unwanted functionality from the code.

• **FOCUS Transformations.** Figure 9 illustrates the different steps in this phase. (1) A product-line application developer chooses the specializations that are suitable for the variant. This is done during the CV analysis phase. (2) A transformation engine, then uses the rules specified in the rules file (3) performs the transformations specified in the file using the hooks left in the middleware code and (4) Optimizing compiler then uses the modified source file to generate executable platform code. For example, challenge 5 in Section 2.2 describing the overly extensible nature of standard middleware can be resolved via annotations described earlier and transforming the original source code to specialized code, which eliminates the indirections and dispatching overhead.

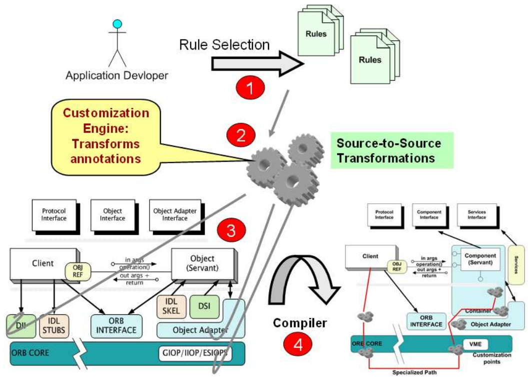


Figure 9. FOCUS: Transformation Process

The advantage of this approach is that the transformations are applied to the source directly enabling an optimizing compiler to generate optimized code. However, a disadvantage is that aspects provide a superior mechanism of weaving code. However, irrespective of the actual approach, our specialization rules are applicable across different mechanisms. The specializations in addition will not change the CORBA interfaces CORBA interfaces, for example, no addition of operation parameters to interface operations, which will avoid breaking CORBA compatibility.

4 Related Research

This section compares our work on context-specific specializations for middleware-based PLAs for DRE systems with related research in a range of system and application domains.

Operating systems. Specialization techniques have been applied to operating systems. For example, the Synthesis

Kernel [21] generated custom system calls for specific situations to collapse layers and eliminate unnecessary procedure calls. This approach has been extended to use incremental specialization techniques. For example, [22] have identified several invariants for an OS `read()` call on HP/UX. Based on these invariants, code is synthesized to adapt to different situations. Once the invariants fail, either re-plugging code is used to adapt to a different invariant or default unoptimized code is used. Our work is extending the catalog of specializations to encompass middleware invariants in the context of PLA-based DRE systems, which have some different constraints. For example, we do not consider re-plugging costs in many DRE systems since it would considerably increase jitter for a product variant.

Middleware. Specialization techniques have also been applied to various generations of middleware. [23] describes the use of the Tempo C program partial evaluator tool to automatically optimize common software architecture structures with respect to fixed application contexts. For instance, the authors show how partial evaluation can be applied to fold together and optimize layers in early generations of middleware, *i.e.*, a remote procedure call (RPC) implementation, by specializing RPC invocations to the size and type of remote procedure parameters (yielding speedups of 1.7x and 3.5x).

Other domains. Specialization mechanisms have been applied to computer graphics, database systems, and neural networks. In computer graphics, for example, ray tracing algorithms compute information on how light rays traverse a scene based on different origination. Specialization of these algorithms [24] for a given scene has yielded better performance rather than general purpose approaches. Similarly in databases [25], general-purpose queries have been transformed into specific programs optimized for a given input. Similarly, training neural networks [26] for a given scenario has improved its performance. The specializations described in this paper, in contrast, focus on specializations that customize standards-based, general-purpose middleware for particular product variants and application-specific contexts.

5 Concluding Remarks & Future Work

Product-line architectures (PLAs) enable organizations to reconfigure their software quickly to respond to new missions and new business opportunities [1]. PLAs are particularly applicable for large-scale distributed real-time and embedded (DRE) systems since reusable software families can be built for a domain (such as avionics, vetronics, or shipboard computing) where applications share many functional and architectural properties and then customized to meet the specific needs of product variants. Standards-based middleware is a key infrastructure technology for PLAs since it provides many reusable policies and mechanisms to simplify the development, customization, and deployment of product variants. The stringent demands of DRE systems, however, require conservation of resources, while simultaneously providing the desired QoS. It is therefore essential to optimize standards-based middleware implementations by eliminating extraneous functionality and extensibility not needed for particular product variants.

This paper systematically identifies the generality within middleware implementations and describes our FOCUS MDD approach for automating the specializations. Currently, we have handcrafted the middleware specializations that resolve the challenges described in Section 2. Preliminary results show that application of our specializations improves end-to-end throughput and latency measures by $\sim$25% over general-purpose middleware. Our future work on FOCUS, will involve:

- Identifying the different types transformations necessary to specialize middleware implementations from our handcrafted approach. For example, examination of our handcrafted approach showed that transformations require capabilities of weaving/removing code at specific points within middleware,
- Specifying a XML DTD that will provide a structure for capturing the different specializations. Simultaneously, the middleware implementation will be annotated with markers to aid in the automation process,
- Development of scripting tools that will parse the XML specialization document and automate the specializations, and
- Devlopment of Modeling paradigm that will determine transformations required to specialize middleware from system models and generate specialization transformations that will be executed by the FOCUS tool. We plan to integrate the modeling paradigm with CoSMIC [27][1] which is an integrated collection of domain-specific modeling languages (DSMLs) and generative tools that support the development, configuration, deployment, and validation of component-based DRE systems.

References

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

[2] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[3] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, vol. 15, November/December 1998.

[1]CoSMIC's open-source MDD tools are available for download at `www.dre.vanderbilt.edu/cosmic`.

[4] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. New York: John Wiley & Sons, 2004.

[5] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[6] W. Roll, "Towards Model-Based and CCM-Based Applications for Real-Time Systems," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Hakodate, Hokkaido, Japan), IEEE/IFIP, May 2003.

[7] D. C. Schmidt, V. Kachroo, Y. Krishnamurthy, and F. Kuhns, "Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications," *IEEE Communications Magazine*, vol. 38, pp. 112–123, Oct. 2000.

[8] G. Blair, A. Campbell, and D. C. Schmidt, "Middleware Technologies for Future Communication Networks," *IEEE Network*, vol. 18, Jan. 2004.

[9] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.

[10] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[11] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[13] Object Management Group, *Lightweight CCM RFP*, realtime/02-11-27 ed., Nov. 2002.

[14] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, pp. 47–52, November/December 1999.

[15] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), pp. 145–159, USENIX, May 1999.

[16] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design 3* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, Massachusetts: Addison-Wesley, 1997.

[17] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, Apr. 1990.

[18] G. Daugherty, "A proposal for the specialization of ha/dre systems," in *Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM 04)*, (Verona, Italy), ACM, Aug. 2004.

[19] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," in *Proceedings of the International Conference on Software Engineering*, pp. 107–119, May 1999.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[21] C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis kernel," *Computing Systems*, vol. 1, pp. 11–32, Winter 1988.

[22] C. Pu, T. Autery, A. Black, C. Consel, C. Cowan, J. W. Jon Inouye, Lakshmi Kethana, and K. Zhang, "Optimistic incremental specialization: Streamlining a commercial operating system," in *Symposium of Operating System Principles*, (Copper Mountain Resort, Colorado), Dec. 1995.

[23] R. Marlet, S. Thibault, and C. Consel, "Efficient Implementations of Software Architectures via Partial Evaluation," *Automated Software Engineering: An International Journal*, vol. 6, pp. 411–440, October 1999.

[24] P. Andersen, "Partial Evaluation Applied to Ray Tracing," DIKU Research Report 95/2, DIKU, 1995.

[25] C. Sakama and H. Itoh, "Partial Evaluation of Queries in Deductive Databases," *New Generation Computing*, vol. 6, no. 2,3, pp. 249–258, 1988.

[26] L. Lei, G.-H. Moll, and J. Kouloumdjian, "A Deductive Database Architecture Based on Partial Evaluation," *SIGMOD Record*, vol. 19, pp. 24–29, September 1990.

[27] Institute for Software Integrated Systems, "Component Synthesis using Model Integrated Computing (CoSMIC)." www.dre.vanderbilt.edu/cosmic, Vanderbilt University, Nashville, TN.