A Generative Middleware Specialization Process for Distributed Real-time and Embedded Systems

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

***Abstract*—General-purpose middleware must often be specialized for resource-constrained, real-time and embedded systems to improve their response-times, reliability, memory footprint, and even power consumption. Software engineering techniques, such as aspect-oriented programming (AOP), feature-oriented programming (FOP), and reflection make the specialization task simpler, albeit still requiring the system developer to manually identify the system invariants, and sources of performance and memory footprint bottlenecks that determine the required specializations. Specialization reuse is also hampered due to a lack of common taxonomy to document the recurring specializations. This paper presents the GeMS (Generative Middleware Specialization) framework to address these challenges. We present results of applying GeMS to a Distributed Real-time and Embedded (DRE) system case study that depict a 21-35% reduction in footprint, and a $\tilde{3}6\%$ improvement in performance while simultaneously alleviating $\tilde{9}7\%$ of the developer efforts in specializing middleware.**

***Keywords* - Middleware, Specialization, Optimization, Generative, Models, Patterns, Frameworks.**

I. Introduction

A large variety of applications and application product lines, particularly those found in distributed, real-time and embedded (DRE) systems, such as avionics, telecommunication call processing, multimedia streaming video, industrial automation, shipboard computing and mission-critical computing environments, are leveraging general-purpose middleware in their design and implementation. Middleware enables these systems to realize long shelf lives by shielding these systems from the constant evolution in the underlying operating systems and hardware resources.

Despite the many advantages offered by middleware, the generality, flexibility, and configurability provided by contemporary general-purpose middleware platforms (which stem from the need to support a range of application domains) leads to over-general mechanisms that in turn result in performance bottlenecks, excessive memory footprint, and even extra power consumption issues for DRE systems. Developing proprietary middleware solutions to overcome these problems is not a viable solution due to cost and maintenance issues. A promising alternative thus is to transform general-purpose middleware into customized variants that are suited for each application domain and product variant – a process we call *middleware specialization.*

Most prior efforts at specializing middleware (and other system artifacts) [1]–[6] often require manual efforts in identifying opportunities for specialization and realizing them on the software artifacts. At first glance it may appear that these manual efforts are expended towards addressing problems that are purely accidental in nature. A close scrutiny, however, reveals that system developers face a number of inherent complexities as well, which stem from the following reasons:

1. Spatial disparity between OO-based middleware design and domain-level concerns - Middleware is traditionally designed using object-oriented (OO) principles, which enforce a *horizontal decomposition* of its capabilities into layers comprising class hierarchies. This design is, however, not suited for specializing middleware since domain concerns tend to map along the vertical dimension, which are shown to crosscut the OO class hierarchies [7] hence necessitating *vertical decomposition*.For example, in OO-based middleware implementations of Real-time CORBA (RTCORBA) [8], the implementation of features related to handling requests at a fixed priority (called the `SERVER_DECLARED` model) or allowing priorities to be propagated from task to task (called the `CLIENT_PROPAGATED_PRIORIY` models) crosscut multiple functional modules such as the object request broker (ORB), the portable object adapter (POA), and request demultiplexing and dispatching modules. Since the two priority models are mutually exclusive, only one configuration can be valid along the critical path between tasks of a DRE system. Thus, any transformation to prune the logic for the unused priority model must necessarily involve modifying several different classes that implement these different modules.

2. Lack of apriori knowledge of specialization requirements due to temporal separation of application lifecycle phases - DRE systems often involve a well-defined application development lifecycle comprising the design, composition, deployment, and configuration phases. Due to the temporal separation between these phases, and potentially a different set of developers operating at each phase, it is not feasible to identify specialization opportunities all at once. Instead, with each successive phase of the development lifecycle, system properties start becoming invariant one by one. For example, the system composition of an end-to-end task chain may reflect

This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

the need to differentiate priorities among multiple information flows across the tasks. However, whether the requests within a flow are handled at a fixed priority at each task or whether the priorities are propagated end-to-end will be evident only after the developers configure the system. Thus, any specialization will have to wait until the configuration of the system is known.

3. Lack of mechanisms for reusing specializations - Unlike the years of efforts in documenting good patterns of software design, there is a general lack of a knowledge base documenting reusable patterns for middleware specialization, which leads to reinventing specialization efforts in identifying what specializations are needed, and in realizing them. For example, if there is no approach to document how the specializations for a particular priority model are performed, then developers will be faced with similar challenges every time the same specialization is to be performed on a different DRE system.

Solution approach — Domain-driven generative middleware specialization - To overcome the challenges outlined above, in this paper we present the GeMS (**Ge**nerative **M**iddleware **S**pecialization) approach for automating the middleware specializations process while promoting reuse. The GeMS framework supports a five step process as follows:

1. Deduce the context for specialization by leveraging models of application compositions, configurations and deployments.
2. Infer the set of specializations that can be performed based on the deduced context.
3. Identify the specialization points within the middleware code base where specializations must be realized,
4. Generate the directives that perform the specializations,
5. Stage the necessary workflow of back-end tools that will execute the specialization directives to result in the specialized middleware.

We use a representative DRE case study to validate the GeMS approach in terms of reductions in memory footprint, improvements in latency, and efforts saved over manual approaches to specialization.

Paper Organization - The rest of the paper is organized as follows: Section II compares GeMS to related work; Section III presents the GeMS process and the underlying techniques; Section IV empirically evaluates GeMS specialization process in the context of a DRE case study; and finally Section V provides concluding remarks identifying lessons learned and scope for improvements.

II. Related Work

We present related research in specialization classifying them along the inherent difficulties in middleware specialization that we outlined in Section I.

1. Addressing the spatial disparity between OO design and domain concerns - Both aspect-oriented programming (AOP) and feature-oriented programming (FOP) have been used extensively for specializing systems by addressing the disconnect between the vertical decomposition of OO design and horizontal decomposition of domain concerns. For example, Lohmann et. al. [1] argue that the development of fine-grained and resource-efficient system software product lines requires a means for separation of concerns [9] that does not lead to extra overhead in terms of memory and performance which they show using AspectC++.

The *FACET* [2] project identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects that represent domain concerns, which then can be woven into the core middleware. Our prior work used FOP-based reverse engineering in a tool called FORMS [10] that prunes unnecessary features from the middleware by deducing the necessary middleware features from high-level application requirements (*i.e.*, domain concerns).

2. Specialization in temporally distinct phases of application lifecycle - The *Modelware* [3] methodology adopts both the model-driven engineering (MDE) [11] and AOP. The authors use the modeling views – *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains, and *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change. 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 [12] shows how middleware can be customized after application characteristics are known.

3. Higher-level abstractions and generative mechanisms - The DADO project [4], [5] has shown how AOP can be used in a software development process to bypass the rigid layered processing by extending the middleware platform with new aspect-oriented modeling syntax and code generation tools. The FOCUS project [13] relies on manual identification of the application invariants, the specialization context and the specialization points within the middleware source, and manual writing of scripts to feed into a transformation tool that specializes the middleware sources.

Limitations in related research - Even if AOP is shown to be effective, it still suffers from the overhead of excessive memory footprint due to the additional code required for instrumenting the aspects within the source codes. Moreover, the learning curve required leads to additional complexity in maintaining and debugging AOP programs. The FACET like AOP approaches additionally require redesigning and refactoring the traditional middleware into aspects. Our work on FORMS does not address the vertical decomposition problem in its entirety since it only accounts for coarser-grained features. As shown later, however, tools like FORMS can be leveraged to add a systematic process to the higher-level requirements reasoning and to customize the middleware build configurations.

Although the FOCUS tool itself is reusable, the specializations required manual identification of opportunities for specialization within the middleware code. Naturally, these solutions are not maintainable, reusable and extensible, and therefore cannot be easily transitioned to apply to different middleware and are cumbersome to evolve with the middleware. Similarly, the manual writing of bypass IDL files required by DADO and refactoring of middleware mandated by FACET hampers reusability.

Modelware demonstrates an interesting approach to specializing middleware, however, its success hinges on generating the entire middleware code from model artifacts. On the contrary our work is focused on specializing existing middleware code. The GeMS framework presented in this paper incorporates the promising ideas from these related research while maximizing the opportunities for automation and reuse.

III. A Framework for Generative Middleware Specializations

Middleware Specialization for DRE systems is a process that manipulates general-purpose middleware by (a) integrating custom features supplied by the application, (b) pruning unwanted features, and (c) optimizing the resulting middleware to address DRE system QoS and resource constraint requirements. For this paper we focus only on the pruning and optimization dimensions. Identifying what features of middleware are excessive, and determining hidden optimization opportunities can be determined only from the invariant properties of DRE systems, and by instrumenting the middleware after excessive features are pruned.

As noted earlier, contemporary efforts at middleware specialization are often based on manual, point solutions. This section first brings out the requirements for an automated middleware specialization process. Subsequently it demonstrates how these requirements are realized within the GeMS framework.

A. Requirements for Generative Middleware Specializations

Detecting the system invariants manually on a case-by-case basis is infeasible, not to mention the subsequent manual efforts at specializing the middleware for each of the system under consideration. Many questions arise if automation is desired: How are the systems invariants to be identified automatically? Once these invariants are identified, how are they mapped to the underlying middleware-specific features that will indicate what parts of the middleware must be pruned and how the rest of the middleware be optimized? This problem is hard given that domain concerns crosscut class hierarchies of middleware design, and because system invariants become evident in different stages of the system lifecycle. We present the key requirements of an automated solution to middleware specialization.

1. Deducing the Specialization Context - We define *specialization context* as the intent that drives the specialization process. Deriving the specialization context relies on detecting the system invariants [14], which become known over the application lifecycle stages.

2. Inferring the Specializations from the Specialization Context - DRE system developers must be able to map the specialization context to one or more known patterns of specialization. Inferring the set of specializations will require a repository of specialization patterns that can be queried using the context, which then returns a set of specializations applicable in that context. Such a repository must be extensible to include new patterns as they are discovered.

3. Identifying the Specialization Joinpoints within the Middleware - The inferred patterns of specialization manifest at a higher level of abstraction than the level of middleware source code that actually must be transformed. Thus, there is a need to identify the collection of *Specialization Joinpoints,* which are regions of code within the middleware where specialization patterns will apply [15].

4. Generating the Specialization Advice - Although the specialization joinpoints are determined, the exact nature of the transformation to be carried out at those joinpoints corresponding to the specialization patterns must be specified as a set of directives, which we call *Specialization Advice*.

5. Executing the Specialization Advice on Middleware Source - A final requirement to realize the specialized middleware code is to apply the specialization advice to the specialization joinpoints. Applying the advice requires a staging of backend tools, such as AspectJ and AspectC++, specific to the programming language in which the middleware is developed, or language-agnostic tools, such as Perl.

B. The GeMS Generative Middleware Specialization Approach

We have developed the Generative Middleware Specialization (GeMS) framework that satisfies the five requirements for middleware specialization highlighted in Section III-A. Figure 1 shows the GeMS approach and the different stages that form the overall process. At the core of GeMS is a model-driven engineering (MDE) approach [11]. Different model interpreter tools form the different stages and use algorithms in GeMS to perform their tasks. The rest of this section describes GeMS.

1) Deducing the Specialization Context from System Models:

Approach: System invariant properties provide an indication of what features from the underlying middleware will be utilized by the applications. Since system invariant properties become evident only with every successive phase of application lifecycle, we classify the system invariants as (1) structural invariants, which are obtained from the structural composition of the system; (2) configuration invariants, which are obtained from the QoS configuration parameters selected for the middleware hosting platforms that specify the performance constraints. These constraints include latency, throughput, timeliness and reliability that are placed on individual application components, and their connections as well as the end-to-end workflows of components (known as component assemblies); and (3) deployment invariants, which are obtained from the resource allocations including the mapping of application software components to processors, platform bindings, endianness, languages, compilers, and collocation of different application software components.

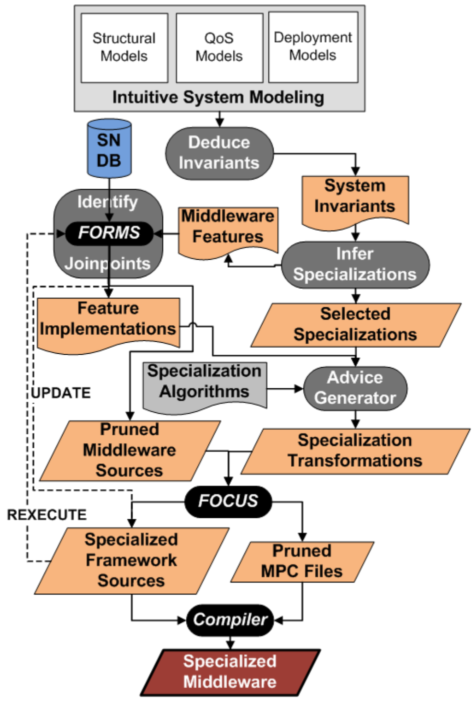

Fig. 1. **The *GeMS* Process**

An approach to identifying these invariants is through model interpreters that traverse the application models and establish the specialization context. Such a step eliminates the need for dedicated modeling annotations to identify the context within the application models. Most coarse-grained contexts can be detected automatically by examining the modeling structure and attributes but finer-grained contexts may need explicit identification.

Implementation: We have developed a model interpreter that traverses the system models to detect the invariants that provide the specialization context. The interpreter makes use of well-defined matching patterns that were specifically developed for the PICML component-based DRE system modeling language [16] to ease the traversal to specific granularity levels (assembly, component, connection, port, interface, methods, parameters, config properties, etc) of the system model. The interpreter proceeds by starting from the highest level of granularity (assembly) to the lowest (parameters, configuration properties). Once it discovers the invariants, it gathers the configuration data associated with them that will be further used to deduce the specialization context. The interpreter maintains an extensible catalog of these matching expressions that can be predefined by the model developer and if necessary can be further extended to accommodate the discovery of newer invariants.

2) *Inferring Specializations from Specialization Context:*

Approach: Depending upon where they occur in the application model, the invariants that form the specialization context have certain semantics that implicitly determine the specializations that can be performed. For instance, application invariants such as repetitive tasks can provide a different specialization context based on the semantics they have, *e.g.*, periodic tasks can manifest in terms of periodic invocations that have synchronous request-response semantics which provide opportunities to optimize the redundant processing along the middleware call processing path. Since the specialization contexts map to different patterns of specialization, an extensible repository that can be queried for the right specializations is needed.

Implementation: We have synthesized an extensible and intuitive repository called `SP-KBASE`, which serves as a knowledge base and is implemented as a complex multidimensional hashmap that stores the specialization patterns corresponding to the specialization. Note that a pattern also encodes the ordering in which individual specializations must be executed. Such an ordering is useful to the specialization staging algorithm that can correctly determine the next specialization to be performed. Another important piece of information that is stored is the incompatibilities or conflicts with other specializations in terms of common code paths or features being manipulated by them.

TABLE II
Performance Optimization Principles [17]

Principle	Description
P1	Avoid obvious waste
P2	Avoid unnecessary generality
P3	Don't confuse specification and implementation
P4	Optimize the expected case

The snippet of `SP-KBASE` knowledge base shown in Table I has been developed based on the intuition of the middleware developers who have expert-level knowledge of the middleware design and implementation. The model interpreter from Step 1 parses the `SP-KBASE` using the uniquely inferred specialization contexts for each invariant and obtains the set of specializations. It then orders them based on the dependency information extracted from the dependency fields and emits out an ordered set of specializations that are to be performed. It reports the incompatible set of specializations to the end-user or simply skips them if running in 'silent' mode.

3) *Identifying Specialization Joinpoints:*

Approach: To identify the specialization joinpoints within the middleware we rely on the fact that most standards-based middleware implementations use frameworks that are based on well-known design patterns. Therefore it is possible to optimize the frameworks by specializing their constituent design patterns. Rather than relying on the source code annotation alone to specify the specialization joinpoints, other techniques like code profiling and inspection, and feature identification and composition can also be leveraged. Specialization joinpoints for functional artifacts can be identified by examining the design patterns in the middleware frameworks whereas the joinpoints for the execution threads of control can be identified by examining the middleware call paths. We leverage well-known optimization patterns (shown in Table II) to specialize traditional middleware frameworks. A preliminary catalog identifying the middleware specialization joinpoints and the specialization techniques that apply to these joinpoints

TABLE I
SP-KBASE: Extensible Catalog of Specialization Techniques

#	System Invariants	Optimization Principles	Specialization Techniques	Specialization Joinpoints	Depends On	Conflicts
S1	**Periodic Invocations**	P1, P4	Memoization	Request Creation	–	S3
S2	**Fixed Priorities**	P1, P4	Aspect Weaving	Concurrency	–	S5
S3	**Homogenous Nodes**	P1	Constant Propagation	Demarshaling Checks	–	S1
S4	**Same Call Handler**	P1, P4,	Memoization + layer-folding	Dispatch Resolution	S2	–
S5	**Known Implementation**	P2	Aspect weaving	Framework Generality	–	S2
S6	**Fixed Platform**	P2	autoconf	Deployment Generality	S2, S5	–

is shown in Table I. We expect this catalog to be extended as new joinpoints are discovered.

Implementation: To specify the specialization joinpoints, in GeMS we first figure out the source code files that need to be transformed. To that end we have leveraged and extended our previous work, FORMS [10], to figure out the file dependency structure for the framework/pattern that needs to be specialized. FORMS can take the required features as input and compute the closure set of source file dependencies that are independent of other closures. This gives us the files we need to process to perform the required source transformations.

We have developed a *generic inspection engine* that uses source code inspection to identify the various individual components of a class such as header includes, forward declarations, scopes, methods, and data members. This pre-processing implicitly helps to identify the specialization joinpoints. Once the pre-processing is done, it provides the necessary information for the following operations – method removal, class movement, scope section replacement, checking for already defined methods, checking the order of typedefs and forward declarations needed for ensuring clean compilations – which form the basis of the specialization advice GeMS generates.

4) Generation and Execution of Specialization Advice:

Approach: Once the specialization joinpoints are identified, to specialize the frameworks into their optimized equivalents, we require rules needed to perform the corresponding source-to-source transformations on the frameworks sources by using the available tools and scripts. One way of performing this is to represent these middleware and patterns in terms of high-level domain-specific architectural models [18]. Then perform model-to-model (M2M) transformations to convert these models into their optimal equivalents and later perform model-to-source (M2S) transformations to produce the optimized source. A drawback of this approach is the additional burden on the middleware developers to construct these models and two-level transformations [19]. Another way is to annotate the framework and pattern source code to identify the specialization points and write source-to-source transformations (S2S) [13]. However it is cumbersome to manually annotate and identify the design patterns and the corresponding implementing sources.

Implementation: In order to avoid these cumbersome techniques, we have developed different generic transformation algorithms for optimizing/transforming each of the commonly used patterns (Bridge, Strategy, Template Method) in contemporary middleware. We have opted to design the transformation algorithms to work with C++ – the most complex middleware implementation OO language being used. In case of other less complicated languages like C#, Java, etc., the algorithms will be much simpler and easier to implement. For example, unwanted indirections (virtual hook methods) in the Strategy pattern can be removed by collapsing class hierarchies, whereas dynamic dispatching (to concrete strategy/feature classes) in the Bridge Pattern can be eliminated by replacing with concrete instances of the strategy/feature implementations. On the other hand, the redundant computations in the middleware call processing path can be optimized by applying layer folding and memoization optimizations.

Algorithm 1 Generic Specialization Advice Generation Algorithm with the Pattern Specialization Plug-Ins

```
F : Framework Feature to be specialized/concretized.
M : Middleware Sources
D : Developer specified advice/specialized code
M_s : Specialized subset of Middleware Sources M
Input - F, M, D
Output - M_s (Initially empty)

begin
F_s := FIND all the framework files that contain the usage of the concrete Framework
  Feature Class f using FORMS
P_s := FIND the pattern implementation files using FORMS
P_d := COLLATE the data necessary for transformation using FORMS and D

{PATTERN SPECIALIZATION PLUG-IN}

REPLACE Base Class occurences with Concrete Class in all framework files F_s
REMOVE the Includes for the Alternative Features from the framework files F_s
REMOVE other Alternative Features from the build configuration using FORMS
return M_s
end
```

The GeMS generic advice generation algorithm 1 generates rules at two levels: (1) the middleware framework level and (2) the constituent design patterns that implement the framework. The framework-specific transformations are performed to accommodate their corresponding constituent patterns-specific transformations. These include specializing the use of the pattern features in the other framework source code, particularly callbacks, feature instantiations and their usages, and the compilation of the framework code. Thus, the algorithm basically performs three major tasks by leveraging and extending the *FORMS* tool - (1) Determines all the framework implementing classes that utilize the feature to be specialized and leverages the corresponding specialization advice provided by the middleware developer, (2) It delegates the pattern specializations to the respective specialization plug-ins as described in algorithm 2, and (3) Specializes the build configuration files for compilation. We have developed similar algorithms for other commonly occurring design patterns within middleware frameworks such as Strategy, Adapter, Template Method, etc. which haven't shown in this paper due to lack of space.

Algorithm 2 Bridge Pattern Specialization Plug-In

```
{Eliminates Indirections - Removes Virtual Method Dispatches}
Input - $P_s$, $M_s$
begin
for each concrete Feature Class Headers $h \in P_s$ do
    ADD Forward Declarations & Public Methods from the Bridge Impl Class
    REMOVE Base Inheritance
    REMOVE all 'virtual' keywords
    CREATE Concrete Feature Class within the main class Constructor
    REMOVE all Alternative Feature references
end for
REPLACE the Bridge Impl Class occurrences with the Concrete Feature Class {also replaces the #includes} in all relevant files
return $M_s$
end
```

Any specialized code/data for the transformations is provided by the middleware developer since they can best determine how to optimize a particular code path within a particular framework. These rules are ultimately fed to the source transformation tools like FOCUS [13] whose Perl scripts execute the transformations on the sources and subsequently FORMS build specialization tools generate the specialized middleware source build configurations.

IV. Evaluating the GeMS Middleware Specialization Process

Since GeMS is a software engineering process, we demonstrate its applicability and evaluate its merits along the following dimensions: (1) We first show how GeMS can be applied to specialize middleware for a representative DRE system case study; (2) We show the savings in effort (and hence improvement in productivity) on the part of a DRE system developer accrued by using GeMS in contrast to manually performing the specializations; and (3) We show the improvement in latencies and static and runtime memory footprints of the specialized middleware version compared to traditional middleware.

A. Illustrating GeMS on a DRE Case Study

We now show how GeMS is applied to specialize middleware for a representative DRE system case study using the specializations cataloged in the knowledge base `SP-KBASE` shown in Table I.

1) Avionics: The Boeing Boldstroke Basic Single Processor (BasicSP) Product-Line:

Scenario Description: BasicSP (Basic Single Processor) is a scenario from the Boeing Bold Stroke avionics mission computing product-line [20], which is a component-based, publish/subscribe platform built atop the TAO Real-time CORBA Object Request Broker (ORB) [21]. Figure 2 illustrates the *BasicSP* application scenario, which is an assembly of avionics mission computing components reused in different Bold Stroke product variants.

BasicSP involves four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays at a rate that is configurable.

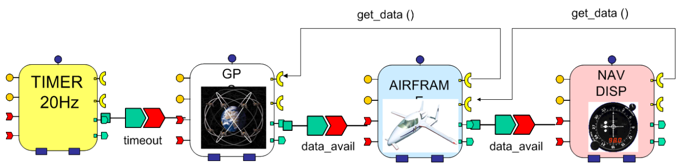

Fig. 2. **The Basic Single Processor (*BasicSP*) Application Scenario**

The time to process inputs to the system and present output to cockpit displays should thus be less than the rate, which as shown in the figure is a single 20 Hz frame.

Problems: The real-time concerns are orthogonal to the traditional horizontal middleware decomposition. In the BasicSP scenario the real-time requirements of predictable latency of 20Hz is desired by each of the individual components so that the aircraft pilots receive their location in real-time. At the same time, these application invariants are not known in advance so they cannot be automatically used to deduce the specializations that can be potentially performed. Moreover, the system requirements may change if the system is deployed in a different physical domain or a different aircraft. For example, a different variant of this scenario for different customer requirements, however, may use different framework components or may send different events to consumers or may service operations via different request dispatchers or may run on nodes with different byte orders, but with the same compiler/middleware implementation, in which case data need not be aligned. These changing requirements render point specialization solutions useless and therefore the need for a systematic, extensible and reusable specialization approach becomes even more apparent.

2) Applying GeMS to Specialize Middleware for BasicSP: We show how the GeMS model interpreters traverse the BasicSP model to realize the specializations.

Applying GeMS Step 1 *(Deducing the Context): Structural Invariants* - The *BasicSP* case study uses a "push-event, pull-data" communication model, which forms the basis of the structural composition of the system. On receiving an event, the Airframe and Nav_Display components repeatedly use the same get_data() operation to fetch new GPS and Display updates, respectively. In a connection between `GPS` and `Airframe` components, therefore, the get_data() operation is sent and serviced by the same request dispatcher.

Configuration Invariants - In *BasicSP*, the connection properties such as the pulse rate of 20 Hz, and corresponding data delivery deadlines form the application QoS configuration model. In this case study, the processing rate is fixed at a maximum latency rate of 20 Hz, the transport protocol used is VME backplane, and the request demultiplexing mechanism within the middleware is reactive.

Deployment Invariants - The target nodes on which the *BasicSP* components are deployed (not shown in the Figure) have the same byte order (endianess) since the processors used in this case study are homogeneous.

Applying GeMS Step 2 *(Inferring the Specializations): Structural Invariants* - The *BasicSP* push-event, pull-data communication model imposes the need for features that support event communication as well as request-response semantics from the underlying middleware. Since there are no concurrent requests, no concurrency support is needed of the middleware, and hence we can deduce only a single request dispatcher is involved which translates to the 'S4' specialization in Table I.

Configuration Invariants - In *BasicSP*, the constant pulse rate of 20 Hz indicates the periodic nature of events and the rate at which data will be pulled. It also indicates the deadline for communication and computation for the periodic task. Periodicity maps to the 'S1' specialization. Since the period of the end-to-end task is fixed, such hard real-time requirements call for features that support fixed priority scheduling translating to the 'S2' specialization. In RTCORBA, the feature that supports this requirement is the `SERVER_DECLARED` model. Since no other priorities and concurrent requests are involved, it needs a simple reactive event demultiplexing and single threaded event processing model within the underlying middleware. Hence, it calls for a single threaded Select Reactor-based [22] request handling. For RTCORBA, this property indicates there is no need for the thread pool mechanisms. Moreover, since only one transport mechanism is used, there is no need for sophisticated software solutions that support pluggable transport protocols, such as the extensible transport mechanism in RTCORBA. Both these invariants translate to the the 'S5' specialization.

Deployment Invariants - In *BasicSP*, since there is no need for byte order checking and codeset negotiations (by virtue of using a homogeneous set of processors), there is no need for marshaling data according to the byte order and data encoding rules including those involving alignment of data along word boundaries. Similarly, there is no need for mapping priorities between sending and receiving components. All these translate to the 'S3' specialization.

Applying GeMS Step 3 *(Identifying Joinpoints):* The identification of specialization joinpoints for the middleware through optimizing the design patterns is automatically performed by the *generic inspection engine* as described in Section III-B3. The necessary annotations get automatically inserted in the pattern implementation sources which are recognized by the FOCUS source code manipulation tool. However, for the other non-structural specializations, the annotations need to be manually defined by the middleware developer since those require explicit specification of the specialized advice that may exhibit different behavior from the original code at which it is applied.

Applying GeMS Steps 4 and 5 *(Advice Generation and Execution):* For lack of space we do not show the complete generated specialization advices. Instead, Listing 1 shows a snippet for the rules that get generated for the bridge pattern corresponding to the steps specified in the Algorithm 2. The FOCUS tool subsequently specializes the middleware code.

Listing 1 Generated Transformation Rules for Bridge Specialization

```
<module name="ACE/ace">
  <file name="Select_Reactor_Base.h">
    <add>
      <hook>REACTOR_SPL_INCLUDE_FORWARD_DECL_ADD_HOOK</hook>
      <data>class ACE_Sig_Handler; </data>
    </add>
    <remove>virtual</remove>
    <remove>: public ACE_Reactor_Impl</remove>
    <remove>#include "ace/Reactor_Impl.h"</remove>
    <substitute>
      <search>ACE_Reactor_Impl</search>
      <replace>ACE_Select_Reactor_Impl</replace>
    </substitute>
  </file>
  <file name="Reactor.cpp">
    <add>
      <hook>REACTOR_SPL_CONSTRUCTOR_COMMENT_HOOK_END</hook>
      <data> ACE_NEW (impl, ACE_Select_Reactor); </data>
    </add>
  </file>
</module>
```

B. Improvements in Developer Productivity through Auto-Generation

We leverage FOCUS [13] to execute the generated specialization advice on the middleware source code. The FOCUS source transformation rules for specializing the design patterns and middleware frameworks are represented in XML. Manually writing these rules by the middleware developer on a per instance basis is not only cumbersome and excessively tedious but also complex to maintain as the middleware source code evolves. Auto-generating them using the GeMS algorithms as described in Section III-B4 alleviates the burden on the developers as well as makes them easy to extend and maintain. Table III shows how many lines are auto-generated on a per-pattern basis and how these translate to cumulative savings for the entire middleware framework that is implemented using that pattern.

TABLE III
Middleware Developer Effort Savings

Design Pattern (Middleware Framework)	#lines Generated	#lines Handwritten	% Savings
Bridge (Reactor)	115/443	17	96.16 %
Strategy (Flushing)	29/201	4	98.01 %
Strategy (Wait On)	29/141	4	97.16 %
Template Method (Pluggable Protocol)	172/974	25	97.43 %

However, developers will still need to provide the specialized code if they wish to specialize a particular middleware call path in their own way. This specialized code is applied like an *aspect advice* at the code joinpoints specified through annotations. As shown, the auto generation almost completely eliminates the burden of manually writing the transformations and figuring out the specialization joinpoints with savings in excess of $\tilde{9}7\%$. For the sake of terseness, we have only shown a few of the frameworks that were optimized.

C. Empirical Evaluations

We evaluated the outcome of applying the GeMS specialization process by measuring the following criteria: (1) the static footprints of the middleware binaries, (2) dynamic footprints of the BasicSP applications, (3) the average latencies of requests, and finally (4) the overall throughput of the application components. We have applied the GeMS specialization process to the widely used TAO Real-time ORB implementation for DRE systems software. Table IV-C reveals that the resultant savings are substantial for DRE applications meant to be deployed on resource constrained embedded devices. The dynamic footprints are a lot higher (5x) than the static footprints of the middleware binaries since the specialized middleware binaries were generated for each BasicSP application components.

TABLE IV
Middleware Performance Improvement Metrics

Metrics	**Before Specialization**	**After Specialization**	**% Savings**
Footprint (Static)	3,226 KB	2,082 KB	35.4 %
Footprint (Dynamic)	13,588 KB	10,657KB	21.57 %
Average Latency	3367 μs	2160 μs	35.84%
Throughput	0.26 reqs/s	0.41 reqs/s	36.59%

V. Conclusions

General-purpose middleware has been incrementally optimized over the period of time to efficiently handle the expected application functionality as well as provide the flexibility and adaptability to handle changing requirements and changing runtime conditions. However, the primary goal behind middleware design being generality and portability, it lacks finer customization and tunability to specific application requirements. To resolve this generality and specificity tension, middleware is usually specialized (customized and adapted) on a case-by-case basis. However this process becomes tedious and non-repeatable as the application requirements change as well as underlying platforms evolve. It is important that any modification to the middleware sources be retrofitted with minimal to no changes to the middleware portability, standard APIs interfaces, application software implementations, while preserving interoperability wherever possible. Otherwise such specialization approaches obviate the benefits accrued from using standards-based middleware. Additionally the accidental complexity from manually applying such approaches to mature middleware implementations renders the specializations tedious and error prone to implement.

In this paper, we presented an automatic, systematic and reusable process for specializing general-purpose middleware that enables the vertical decomposition of middleware along the domain concerns by deducing the invariant properties, inferring the specializations and generating the transformations required to specialize middleware sources. Our approach is realized within the GeMS tool. We also provided detailed evaluation of the process by quantifying the developer productivity improvements and reduction in latency, response time and memory footprint of the resulting specialized middleware.

References

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

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

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

[4] 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.

[5] 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*. Vancouver, British Columbia, Canada: ACM Press, 2007, pp. 25–35.

[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*. New York, NY, USA: ACM, 2008, pp. 108–119.

[7] G. Gottlob, M. Schrefl, and B. Röck, "Extending object-oriented systems with roles," *ACM Trans. Inf. Syst.*, vol. 14, no. 3, pp. 268–296, 1996.

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

[9] 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*, May 1999, pp. 107–119.

[10] A. Dabholkar and A. Gokhale, "Feature-Oriented Reverse Engineering-based Middleware Specialization for Product-Lines," *Journal of Software (JSW) - Special Issue on Recent Advances in Middleware and Network Applications*, vol. 6, 2011 (to appear).

[11] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[12] 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*. Chicago, Illinois: ACM Press, 2005, pp. 63–74.

[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*, Leuven, Belgium, Apr. 2006, pp. 205–218.

[14] R. Marlet, S. Thibault, and C. Consel, "Efficient Implementations of Software Architectures via Partial Evaluation," *Automated Software Engineering: An International Journal*, vol. 6, no. 4, pp. 411–440, October 1999. [Online]. Available: citeseer.csail.mit.edu/marlet99efficient.html

[15] 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*, Jun. 1997, pp. 220–242.

[16] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems," in *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 190–199.

[17] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. San Francisco, CA: Morgan Kaufmann Publishers (Elsevier), 2005.

[18] A. Gokhale, D. Kaul, A. Kogekar, J. Gray, and S. Gokhale, "POSAML: A Visual Modeling Language for Managing Variability in Middleware Provisioning," *Elsevier Journal of Visual Languages and Computing (JVLC) 2007*, vol. 18, no. 4, pp. 359–377, 2007.

[19] openArchitectureWare, "openArchitectureWare," www.openarchitectureware.org, 2007.

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

[21] 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*, vol. 3, no. 2, Feb. 2002.

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