Middleware Specialization using Aspect Oriented Programming

Dimple Kaul
Department of Electrical Engineering & Computer Science
Vanderbilt University
Nashville, TN 37235
dkaul@dre.vanderbilt.edu

Aniruddha Gokhale
Department of Electrical Engineering & Computer Science
Vanderbilt University
Nashville, TN 37235
gokhale@dre.vanderbilt.edu

ABSTRACT

Standardized middleware is used to build large distributed real-time and enterprise (DRE) systems. These middleware are highly flexible and support a large number of features since they have to be applicable to a wide range of domains and applications. This generality and flexibility, however, often causes many performance and footprint overheads particularly for product line architectures, which have a well-defined scope smaller than that of the middleware yet must leverage its benefits, such as reusability. To alleviate this tension thus a key objective is to specialize the middleware, which comprises removing the sources of excessive generality while simultaneously optimizing the required features of middleware functionality. To meet this objective this paper describes how we have applied Aspect-Oriented Programming (AOP) in a novel manner to address these challenges. Although AOP is primarily used for separation of concerns, we use it to specialize middleware. Aspects are used to select the specific set of features needed by the product line. Aspect weaving is subsequently used to specialize the middleware. This paper describes the key motivation for our research, identifies the challenges developing middleware-based product lines and shows how to resolve those using aspects. The results applying our AOP-based specialization techniques to event demultiplexing middleware for the case of single threaded implementation showed 3% decrease in latency and 2% increase in throughput, while in the thread pool implementation showed 4% decrease in latency and 3% increase in throughput.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Engineering—*Software Architectures*

General Terms

Performance, Specialization, Middleware

Keywords

Aspect-Oriented Programming, Product lines

1. INTRODUCTION

The implementations of standard, general-purpose, middleware, such as J2EE, CORBA and .NET are very complex. The primary reason for this complexity arises from the need for versatility in supporting a wide range of domains and applications, with different functional and quality of service (QoS) requirements. These qualities make the middleware highly flexible and also optimized for the common cases.

Such versatility and flexibility comes at a premium for product line architectures, which are defined by a narrower scope and by a set of commonalities [9]. The high degree of flexibility and feature richness of standardized middleware imposes performance and footprint overhead for product lines. For example, due to the wide range of features the code bases of these middleware become bloated. A particular product line may only need a few of the features. Therefore there is a need to trim unwanted features in order to support the needs of product lines more effectively. Moreover, opportunities for optimizations in the required features may get masked due to the excessive generality of middleware. A key goal therefore is to provide middleware specializations, which remove unwanted features in middleware while simultaneously provide finer grained optimizations to the required features.

We envision that once we obtain a specialized middleware suited for a product line, system developers can now choose from a narrow set of strategies that can be parameterized in accordance with the needs of a product variant within the product line. We hypothesize that such configurations can lead to improved middleware QoS properties, such as high throughput, low latency, low memory overhead etc in the context of the concerned product line.

Unfortunately, different product lines have their own different set of QoS and functional requirements. Complex middleware designs, such as in J2EE, CORBA or .NET, cannot make any assumptions about any specific domain and hence do not provide any reusable mechanisms to specialize these middleware to suit particular product lines. In the current state of the art, middleware features are managed by selecting them before coding or refactoring of original code. Such an approach may apply to a few applications but ultimately it does not provide any reusable mechanisms that can generically be applied to specialize middleware for product lines. Moreover, ad hoc means to refactoring is always time consuming and error prone.

In order to achieve the vision of specialized middleware, which comprises removing generalization, achieving high degree of configuration and optimization of required features, and validation according to product line-specific needs we need tool-driven mechanisms that will automate the process. These specialization techniques will be helpful only if features are selectable based solely on the various middleware strategies or specifications that will fulfill user requirements. This paper explores the use of aspect oriented programming (AOP) [16] incorporated by the AspectC++ tool (`www.aspectc.org`) to automate the middleware specializations. For our work we chose the ACE C++ middleware (`www.dre.vanderbilt.edu`) as the platform to demonstrate our ideas.

This paper is structured as follows: Section 2 describes the central theme of this paper. It describes different middleware specialization techniques mentioned in the literature and pinpoints the reasons why we used Aspect-Oriented Programming (AOP)to specialize middleware. It then describes how we have applied AOP to resolve the generality challenges of middleware by focusing on a subset of the middleware used for specialization; Section 3 describes the results of our experiments comparing the non-specialized and specialized middleware; and finally Section 4 provides concluding remarks and future work.

2. MIDDLEWARE SPECIALIZATION VIA ASPECT ORIENTED PROGRAMMING

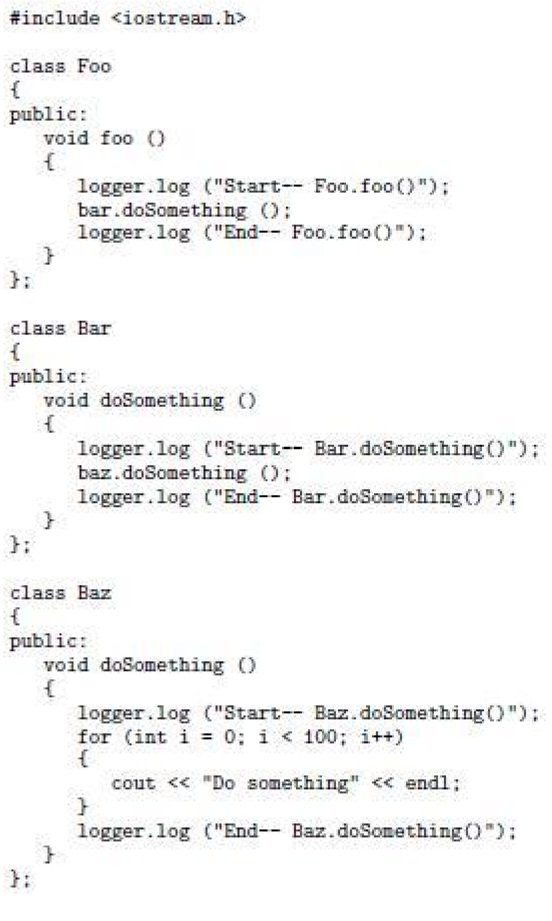
Section 1 motivated the need for specializing middleware to suit them for the requirements of different variants of product lines. Middleware specialization can be achieved by traditional software design and implementation techniques including code refactoring, "ahead of time" design or even using component frameworks [17]. But all these techniques illustrate several drawbacks including large memory requirements stemming from the use of component frameworks, error prone configurations which is usually attempted manually and large performance overheads.

There are various specialization techniques described in literature, which can be leveraged to specialize middleware. For example, Feature-Oriented Programming (FOP) [1] is an appropriate technique to design and implement program families, and which uses incremental and stepwise refinement approaches [1, 11]. FOP aims to cope with the increasing complexity and increasing lack of reusability and customizability of contemporary software systems. Aspect-Oriented Programming (AOP) [16] is another related programming paradigm and has similar goals: It focuses primarily on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability. However, it does not focus explicitly on incremental designs or program families. Aspect-Oriented Programming can change an existing functionality without refactoring of code, addresses concerns with minimum coupling, makes it reusable and implements no hierarchy refinements. These features of AOP can lead to error free and efficient code. It can prevent code clutter, tangling and scattering and makes it easy to add new functionality by creating new aspects. New features or behavior can be added at any stage of development thus relieving the developer of committing to under/over design. So an unknown functionality which cannot be predicted ahead of time is not a problem.

These characteristics of AOP can be leveraged to create an implementation that is easier to design, understand, and maintain resulting in higher productivity, improved quality, and better ability to implement newer features. We therefore leverage these capabilities of AOP as a middleware specialization technique for product lines.

2.1 Overview of AOP

Without causing any intrusive changes to entire code, AOP technology helps modularize the implementation, and helps reduces dependencies between modules [14]. AspectC++ is a tool using AOP for the C++ language. For every valid C++ program, AspectC++ can be used. AOP principles supported by tools like AspectC++ address the challenges of crosscutting concerns which pure OO methods fail to do so. Using pointcuts and advices, an aspect weaver brings aspects and components together. An advice defines the code that is defined on these joint points.



```
#include <iostream.h>

class Foo
{
public:
   void foo ()
   {
      logger.log ("Start-- Foo.foo()");
      bar.doSomething ();
      logger.log ("End-- Foo.foo()");
   }
};

class Bar
{
public:
   void doSomething ()
   {
      logger.log ("Start-- Bar.doSomething()");
      baz.doSomething ();
      logger.log ("End-- Bar.doSomething()");
   }
};

class Baz
{
public:
   void doSomething ()
   {
      logger.log ("Start-- Baz.doSomething()");
      for (int i = 0; i < 100; i++)
      {
         cout << "Do something" << endl;
      }
      logger.log ("End-- Baz.doSomething()");
   }
};
```

Figure 1: Normal Logging operation using OOP's method [19]

Consider the classic example of a logging operation in a sample OOP method [19] shown in Figure 1. Here the logger instance of Logger class is called in each and every class that needs to have logging capability. This example demonstrates the tangling of the logging concern across the application logic. We briefly summarize the key ideas in AOP below alluding to an example illustrated in Figure 1:

1. **Advice:** This is the code that is applied to, or that crosscuts the existing code. In our example, this is the logging code that executes when the thread enters or exits a method that we want to untangle.

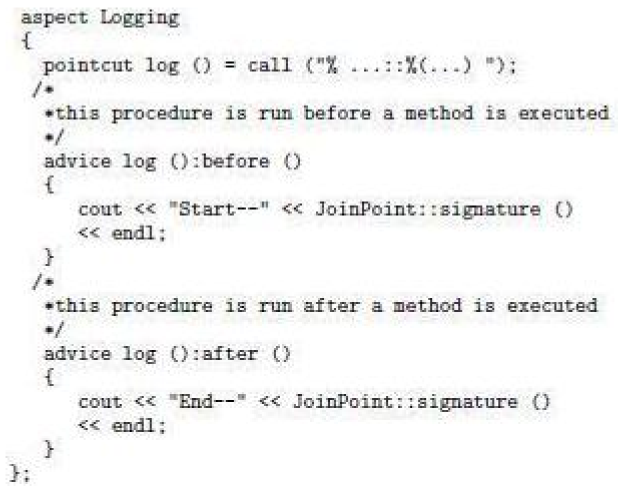
   There are three choices when advice is executed (a) before()- advice code is executed before the original code. It can be used to read/modify parameter values, (b) after()- advice code is executed after a particular control flow or original code is executed. It can be used read/modify return values. and (c) around()- advice body is executed instead of control flow.

2. **Join point:** It denotes a position to give advice in an aspect. Points in the model where aspects can be woven in, e.g., class, methods, structures etc.

3. **Pointcut:** This is the term given to the point of execution in the application at which crosscutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method. Some of the Join points described by pointcut expressions are execution(), call(), cflow(), throws() etc.

4. **Aspect:** The combination of the pointcut and the advice is termed an aspect.

In the other code snippet [19] shown in Figure 2 we observe that by using one simple aspect file developers do not have to change all the classes manually. This makes the code less error prone, flexible and extensible. The above example demonstrates how AOP can be used to specialize ACE middleware because AOP does not change the original code base. Instead different specializations can be captured as aspects in different files and these can then transform the original code base into specialized form.



```
aspect Logging
{
  pointcut log () = call ("% ...::%(...) ");
  /*
   *this procedure is run before a method is executed
   */
  advice log ():before ()
  {
     cout << "Start--" << JoinPoint::signature ()
     << endl;
  }
  /*
   *this procedure is run after a method is executed
   */
  advice log ():after ()
  {
     cout << "End--" << JoinPoint::signature ()
     << endl;
  }
};
```

Figure 2: Rewriting above example in AOP way will be like this [19]

2.2 Aspect-Oriented vs. Object-Oriented Progamming

Aspect-Oriented refactoring [18] offers more expressive power than can be achieved by object orientation alone. Our experience conducting this research revealed that aspect-oriented refactoring was often simpler. For example, consider Figure 3, which shows how in pure OOP the classes and the requirements relationship form a mesh. This implies that a requirement is dependent on multiple classes and if there is any change in one requirement it will lead to change in all the classes leading to unnecessary maintenance complexity. Thus in pure OOP in order to change any code using object oriented process only introduces significant complexity in already existing source code. Using AOP by capturing aspects in separate files, however, ensures that the actual source code is hardly touched. In AOP every requirement can be modeled as an aspect. Hence maintaining and changing of requirements is easier and maintainable.

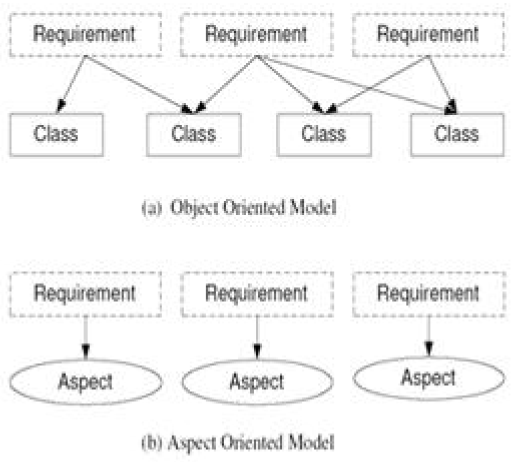


Figure 3: Object Oriented Model and Aspect Oriented Model

2.3 ACE Middleware Specialization using AspectC++

There are very few tools developed specifically for middleware specialization. One of the tools developed is Feature Oriented Customizer (FOCUS) [3]. FOCUS is a domain specific modeling tool that has been developed to automate specialization of middleware. Code is annotated with specialization rules and middleware developer has to select suitable specialization rules. Its transformation engine is a Perl based tool which selects the appropriate specialization files and transforms it into changed source code file. The code is then compiled to executable code. In this tool join-points need to be manually identified and the source code changed manually to insert hooks. Also, correctness of the transformation needs to be validated externally.

We now describe how we have applied AspectC++ for specialization of the ACE middleware. In particular for specialization we focused on the Reactor pattern within ACE. To add or modify different features in Reactor implementations different aspects were defined. These aspects were defined in different files and for different combinations of these aspect files made it possible to achieve different middleware specializations. Because the number of aspects are small and they are totally isolated from actual source code, their management is relatively easy and less error prone. And all this was possible without making any change to code base.

Source code transformation i.e., weaving is done based on aspects at compile time using AspectC++ compiler (ag++). This compiler supports a superset of the C++ language. This language contains constructs to identify join points in the component code and to specify advice in the form of code fragments that should be executed or will execute at these join points. The output of the AspectC++ compiler is plain C++ code, which can be translated with standard C++ compilers to executable code. The compile time for building ACE with AspectC++ woven code is slightly more than the non-aspectized code, however, as shown later this overhead has no impact on the runtime performance. Also, while building the full functional middleware with selected specializations, the resulting executable passed all the build verification tests in ACE indicating validity of aspectized code.

Case Study: Select Reactor and Thread Pool Reactor Specialization

Middleware is often developed as set of frameworks that can support multiple types of functionalities. This overly excessive generality of functionalities can be configured using different options, such as different concurrency models (Thread-per-connection, Thread pool, or Thread-per-request).

In this case study we will describe preliminary work that illustrates the use of AspectC++ for the specialization of ACE middleware, in particular we are targeting a class of product lines that are network centric and must deal with event-driven style of programming. An OO based event-driven interface in ACE is the Reactor. The Reactor framework in ACE implements the Reactor pattern, which decouples the demultiplexing and dispatching of events from the handling of the events. It was developed to support different types of alternative concurrency models. For this paper we focus on two types of concurrency models i.e., single threaded and thread-pool reactor.

The OO design philosophy in ACE enables support for all these alternate mechanisms transparently, which is achieved by an elegant class hierarchy of base and subclasses, and template parameterization. For example, for all types of concurrency models of reactor implementations, ACE uses the `ACE_Reactor_Impl` as the abstract base class which delegates the actual work to its subclasses e.g., `ACE_Select_Reactor` (for single threaded reactor implementation)or `ACE_TP_Reactor` (for thread pool reactor implementation) via virtual method calls.

The choice of the reactor implementation is chosen via ACE-specific configuration mechanisms. It is assumed that once a particular type of reactor is selected, it never changes during the lifetime of a system. Based on this choice of the reactor implementation, we use AspectC++ advice whose goal is to eliminate the virtual method call between the abstract base class and the implementation of the reactor chosen. Thus, the advice effectively replaces the abstract base class `ACE_Reactor_Impl` method call by child class `ACE_Select_Reactor` or `ACE_TP_Reactor` method directly.

In the following we illustrate some of the specializations we implemented using method transformations in the Select and Thread Pool Reactor classes:

```
/**
  * This aspect is for Single Threaded specialization
  */
aspect Single_Thread_Implementation
{
  /**
    * It redirects purge_pending_notifications
    * method of ACE_Reactor_Impl to same method
    * of ACE_Select_Reactor subclass.
    */

  advice call ("% ACE_Reactor_Impl
    ::purge_pending_notifications(...)"):around ()
  {
    ((ACE_Select_Reactor_Impl *) tjp->target ())->
      ACE_Select_Reactor_Impl
      ::purge_pending_notifications
       (*tjp->arg < 0 >(),*tjp->arg < 1 >());
  }
}

/**
  * This aspect is for Thread Pool specialization
  */
aspect Thread_Pool_Implementation
{
  /**
    * It redirects handle_events method of
    * ACE_Reactor_Impl to same method of
    * ACE_Select_Reactor subclass.
    */

  advice call ("% ACE_Reactor_Impl
         ::handle_events(int)"):around ()
  {
    ((ACE_TP_Reactor *) tjp->target ())->
      ACE_TP_Reactor
      ::handle_events (*tjp->arg < 0 >());
  }
}
```

In the above code snippet of single thread implementation we are redirecting method call of `ACE_Reactor_Impl::purge_pending_notification` to `ACE_Select_Reactor::pu\-rge_pending_notification` directly. It should be noted that this method is called almost 16 times in a single server/-client scenario. So the removal of the indirection provides performance gains that are amortized over a large number of requests. This is expected in event driven services that have to deal with a large number of client requests.

3. RESULTS & OBSERVATIONS

This section describes results of our experiment comparing the performance of the original ACE reactor pattern with the specialized version. We collected empirical data that compared the specialized version of ACE with the original version along different dimensions including end to end latency and throughput. We used the ACE middleware's performance test suite to conduct these performance tests and study the impact of AOP on latency and round trip throughput changes.

Our experiments illustrate that the refactored middleware framework showed a significant improvement running the ACE performance tests. To demonstrate the benefits of implementing AOP for ACE middleware framework we discuss two specializations of ACE concurrency models in the reactor and illustrate the improvements in performance i.e., latency and throughput.

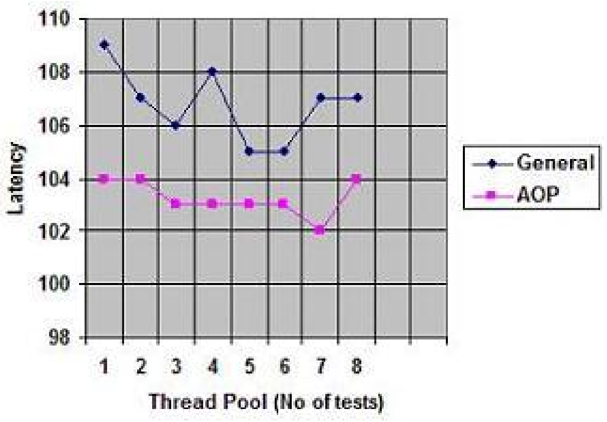


Figure 6: Thread Pool reactor (Latency Vs No. of test run)

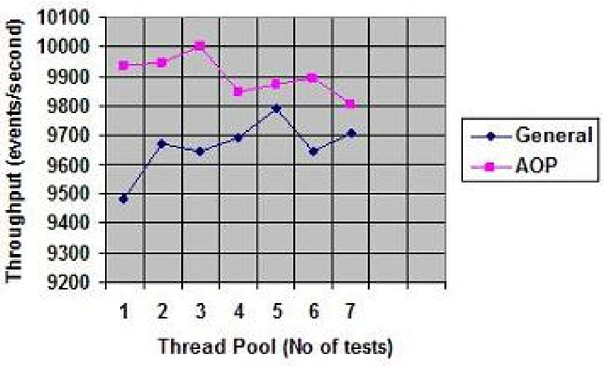


Figure 7: Thread Pool reactor (Throughput Vs No. of test run)

Case I: Single Threaded Reactor

In this case we use AOP to remove the virtual table indirection by bypassing the *virtualness* of abstract base class methods of reactor and calling the child class methods directly assuming that in this case application is using only single threaded reactor.

Figures 4 and 5 show improved end-to-end latency and increase in throughput applying AOP for the specialization of the reactor.

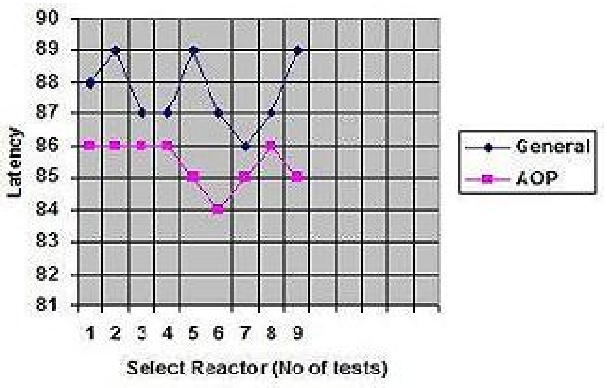


Figure 4: Single threaded reactor (Latency Vs No. of test run)

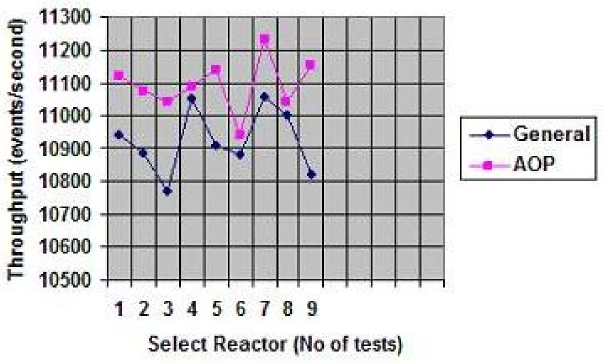


Figure 5: Single threaded reactor (Throughput Vs No. of test run)

Case II: Thread-pool Reactor

In this case we use AOP to remove the virtual table indirection by bypassing the *virtualness* of abstract base class methods of reactor and calling the child class methods directly assuming that in this case application is using only thread-pool threaded reactor.

Figures 6 and 7 show improved end-to-end latency and increase in throughput when specialization of reactor is done using aspects.

Observations and inferences

Table 1 lists out the percentage decrease of latency and increase in throughput in select and thread pool reactor implementation:

For our feasibility study reported in this paper, the specializations of removing *virtualness* was performed manually i.e., aspects were written by hand for a selected set of methods of the reactor class in ACE. The experiments we performed and the results we obtained are encouraging, which demonstrates the promise of applying Aspect Oriented Programming to specialize middleware. Another critical observation we made in our study was that the techniques for specializing a single threaded reactor were entirely different from those that were needed for a thread pool reactor. We infer that the process of manually applying specializations for very large middleware frameworks therefore cannot scale and hence requires a generative approach that will enable the automated synthesis of the specialization files. Our current work focuses on this dimension of the research where we are applying model-driven generative programming [6] to automate the process of middleware specialization.

Reactor	Select	ThreadPool
Latency	-3%	-4%
Throughput	2%	3%

Table 1: Average Percentage Change

4. CONCLUDING REMARKS

This paper motivates the need for middleware specializations and describes a study using Aspect-Oriented Programming (AOP) to automate the specializations. As a result of this study we infer that the use of AOP can play an important role in automating middleware specializations. For proof of concept we used AspectC++ to specialize the reactor implementation in the ACE middleware by removing indirection by bypassing "virtualness" of base class and directly calling child class methods. The preliminary results are encouraging and demonstrate that application of our specializations improves end-to-end throughput and latency over general purpose middleware.

We are continuing to work on implementing aspects for other QoS challenges of middleware using AspectC++ and specializing other patterns. To overcome the limitations of manually implementing the aspects, we are exploring the use of model-driven techniques, which will enable a system developer to express specialization requirements at higher levels of abstractions and automatically synthesizing the aspects.

5. REFERENCES

[1] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer, 'Scaling Step-Wise Refinement' ICSE May 2003, Portland, Oregon.

[2] Sven Apel, Thomas Leich, Marko Rosenm uller, and Gunter Saake. 'Combining Feature Oriented and Aspect Oriented Programming to Support Software Evolution', RAM-SE'05, July 2005, Glasgow, Scotland.

[3] Arvind Krishna, 'Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems', MODELS 2005, October 2, 2005, Jamaica.

[4] Don Batory, 'Feature-Oriented Programming and the AHEAD Tool Suite', ICSE 2004, Edinburg, Scotland.

[5] Don Batory, 'Feature Oriented Programming for Product-Lines Tutorial', ICSE 2003, Portland, Oregon.

[6] Czarnecki K., Eisenecker U. W. 'Overview of Generative Software Development'. Unconventional Programming Paradigms (UPP) 2004, Mont Saint-Michel, France.

[7] Roberto E. Lopez-Herrejon. 'Understanding Feature Modularity in Feature Oriented Programming and its Implications to Aspect Oriented Programming'. ECOOP2005 PhDOOS Workshop and Doctoral Symposium, Glasgow, Scotland.

[8] Aniruddha Gokhale, Douglas C Schmidt, Balachandran N, Jeff Gray, Nanbor Wang. 'Model-Driven Middleware'. Middleware for Communications, (Qusay Mahmoud, ed.), John Wiley and Sons, 2004.

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

[10] D.Batory, Roberto Lopez-Herrejon, Jean-Phillipe Martin. 'Generating Product-Lines of Product-Families'. IEEE International Conference on Automated Software Engineering (ASE'02), Edinburgh, UK.

[11] Olaf Spinczyk, Daniel Lohmann. 'Aspect-Oriented Programming with C++ and AspectC++'. AOSD 05, Chicago, USA.

[12] Charles Zhang, Dapeng Gao and Hans-Arno Jacobsen. 'Generic Middleware Substrate through Modelware'. ACM/IFIP/USENIX 6th International Middleware Conference (Middleware 2005), Grenoble, France

[13] R. E. Lopez-Herrejon and D. Batory. 'Improving Incremental Development in AspectJ by Bounding Quantification'. In Software Engineering Properties and Languages for Aspect Technologies(SPLAT), 2005. Chicago, USA.

[14] Olaf Spinczyk, Daniel Lohmann. 'AspectC++ Quick Reference'.

[15] D.Batory. 'A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite'.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina. 'Aspect-Oriented Programming'. In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[17] Wasif Gilani, Nabeel Hasan Naqvi, Olaf Spinczyk. 'On Adaptable Middleware Product Lines'. ACM International Conference, Proceedings of the 3rd workshop on Adaptive and reflective middleware 2004. Toronto, Ontario, Canada.

[18] Adrian Colyer, Andrew Clement. 'Large-scale AOSD for Middleware'. AOSD 2004. Lancaster UK.

[19] Aspect-Oriented Programming article on http://sel.ics.es.osaka-u.ac.jp/research/aspect/index.html.en