Automating Middleware Configuration and Specializations via Model-based Aspect-Oriented Software Development

Dimple Kaul
Metavante Corp, USA

Aniruddha Gokhale
Vanderbilt University, USA

Preface

Abstract

Distributed computing infrastructures, such as middleware and virtual machines, are designed to be highly flexible and feature-rich to support a wide range of applications and product lines in multiple domains. Applications with stringent quality of service (QoS) demands (e.g., latency, fault tolerance, and throughput), however, find this feature richness and flexibility to be a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains. To alleviate this tension, 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 in an automated fashion.

This work provides three main contributions to research to make a highly specialized middleware. First, it illustrates about the modeling language we developed to compose and configure systems. Secondly, it demonstrates an approach to middleware specialization using aspects oriented programming. Third, it discusses our approach of automating middleware specialization by integrating our model-based tool and aspect-oriented software development techniques. It also describes our investigations into discovering various secondary and crosscutting concerns in metadata management for large-scale distributed data storage. We describe how we have applied aspect-oriented technique to address these crosscutting concerns in metadata management for a high performance distributed storage framework.

Acknowledgments

Dimple Kaul's Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Aniruddha Gokhale, for providing me with this great opportunity to work in his group. Without his support, guidance and mentorship this work would have not been possible. I would also like to thank Dr. Douglas Schmidt for guiding and giving constructive comments.

During my research work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my thanks to all those who have guided me with my work in the Institute of Software Integrated Systems (ISIS) and Advanced Computing Center for Research and Education (ACCRE) at Vanderbilt University department.

I am also grateful to my defense committee Dr. Aniruddha Gokhale, Dr. Jeff Gray and Dr. Alan Tackett for their time and support for reviewing this work.

On the personal note I own my loving thanks to my husband Deepak and daughter Shireen for their understanding, support, and patience. And finally to my parents, who taught me value of education and who's best wishes and prayers were always with me.

Aniruddha Gokhale's Acknowledgments

I am honored to have had Dimple Kaul as my student who worked with a great deal of dedication that resulted in this wonderful thesis.

I would like to thank the Department of Electrical Engineering and Computer Science at Vanderbilt University for all their support. I would also like to thank my family for the support.

Finally, we thank VDM Verlag Dr. Muller for providing us the opportunity to publish this thesis as a monograph.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed computing infrastructures including general-purpose middleware solutions, such as Java Enterprise Edition (Java EE), CORBA and .NET, and virtual machines (VMs) have been one of the key enabling technologies responsible for the rapid and timely growth of a wide variety of network-based applications and application families (*i.e.*, product lines) found in multiple domains, such as real-time and embedded systems [2], enterprise systems and grid computing [15].

Complex distributed applications hosted on these middleware platforms illustrate the need for various functional and para-functional, such as quality of service (QoS), concerns that must be addressed simultaneously when these properties are provisioned on the middleware platforms. Para-functional concern provisioning comprises the challenge of choosing the right set of configuration and composition parameters of the middleware platforms, and validating that these meet the para-functional requirements of the applications.

Meeting these objectives is a hard problem, which stems primarily from the characteristics of the middleware platforms. Middleware is designed to be general-purpose, highly flexible and very feature-rich *i.e.*, middleware provides a rich set of capabilities along with configurability to support a wide range of application classes in many domains. There are multiple software layers in middleware and these layers provide platform-independent execution semantics and reusable services (*e.g.*, concurrency management, connection management, data marshaling, location transparency), which coordinate how application components are composed and interoperate.

There are many applications with stringent QoS demands (*e.g.*, latency, fault tolerance, and throughput) and they find the feature richness and flexibility to be a source of excessive memory footprint overhead and a lost opportunity to optimize for significant performance gains. There is a need one hand for applications to continue to benefit from the elegant, object-oriented designs and interfaces of middleware for maximum reuse and interoperability. On the other hand, it is necessary for applications to use only the required features of the middleware by automatically specializing it and derive maximum benefits in response to their QoS needs.

Additionally, traditional approaches to middleware provisioning typically use low-level, non-intuitive, and technology-specific mechanisms, which are not reusable across multiple middleware technologies and across applications. For example, we have seen how provisioning requires the manual configuration of XML files [45] that are several thousand lines long. Furthermore, often the QoS validation phase is decoupled from the configuration phase. Moreover, the validation phase uses processes that do not leverage decisions made at the configuration phase, which limits the optimizations and fidelity of the QoS validation phases.

A solution to address these problems is to provide a mechanism that raises the level of abstraction at which system integrators can provision middleware, where we define *middleware provisioning* to include the specialization of middleware so that only the required features of the middleware are selected and configured for the para-functional properties. This phase is followed by validating that these specializations and configurations meet the QoS needs of the applications. Visual aids are one of the best known techniques to intuitively reason about any system [17] at higher levels of abstractions. A desired capability of a visual tool for the middleware provisioning problem is one that can provide a clean separation of concerns [37] between the feature selection and parameter configuration, and QoS validation phases, yet unifies the two phases such that decisions at one phase can automate and optimize steps at subsequent phases.

Research in various advanced programming technologies such as Aspect-Oriented and Feature-Oriented design methodologies [6, 14, 28] have shown promise in terms of managing the complexity of large-scale software design through explicit *separation of concerns* (SoC) [37]. These qualities largely eliminate the overhead of the trial-and-error, iterative process incurred by traditional methodologies. These approaches within a visual tool provide the most benefit by enabling the reasoning of application QoS requirements and desired functionality at an intuitive and higher level of abstraction, and be able to automatically synthesize the low-level middleware specialization aspects so that other Aspect-Oriented Programming (AOP) tools, such as AspectC++ [48], can subsequently specialize middleware code.

1.1 Our approach

In our research we combined the power of two important paradigms called Model Driven Engineering (MDE) [40] and Aspect-Oriented Software Development (AOSD). In this context we describe our Domain-Specific Modeling Language (DSML) called Pattern Oriented Software Architecture Modeling Language (POSAML) [24], which incorporates the notion of middleware building blocks that are viewed as being made up of software patterns [16,43] for building network-centric software systems.

We will discuss the generative capability of this modeling language to generate middleware specific artifacts and AOP specialization files containing important constructs of aspects *i.e.*, pointcuts and advice. Using this model-driven technique to generate these artifacts can result in a highly efficient and optimized middleware system by removing manual steps in configuring and specialization of middleware. We also describe the use of AOP to automate the middleware specialization [22]. In addition to these contributions we will present a case study of using AOSD techniques to address various crosscutting challenges for an application called Logistical Store (L-Store) [51].

1.2 Thesis Organization

The remainder of this work is organized as follows: In Chapter 2 we describe the pattern language modeling details of the POSAML language; In Chapter 3 we describe different middleware specialization techniques mentioned in the literature and point out the reasons why we used AOP to specialize middleware. It also illustrates how we applied AOP to resolve the generality challenges of middleware by focusing on a subset of the middleware used for specialization [22]; In Chapter 4 describes how we specialized middleware by adding modeling capability to POSAML to automate specialization of middleware system. In Chapter 5,we describe a case study of how we used AOSD techniques to address various crosscutting challenges; In Chapter 6 we describe the results and analysis of our experiments comparing the non-specialized and specialized middleware and also the middleware artifacts generated; In Chapter 7 we discuss related research; Finally in Chapter 8 we describe the conclusion, lessons learned and future work.

Chapter 2

Pattern Oriented Software Architecture Modeling Language

In this Chapter we describe challenges in middleware provisioning and how visual domain-specific modeling languages provide the solutions to address these challenges. Middleware provisioning is the activity that comprises the specialization of the middleware platform and its configuration, and validating that these meet the QoS needs of the application under expected workloads.

2.1 Challenges in Middleware Provisioning

The motivation for designing visual tools in middleware provisioning stems from the non-intuitive, non-reusable, and error-prone nature of traditional approaches. There are many challenges involved in designing a visual tool. For visual tools to be effective, they must meet a set of criteria described below and should be able resolve the challenges arising in meeting these criteria.

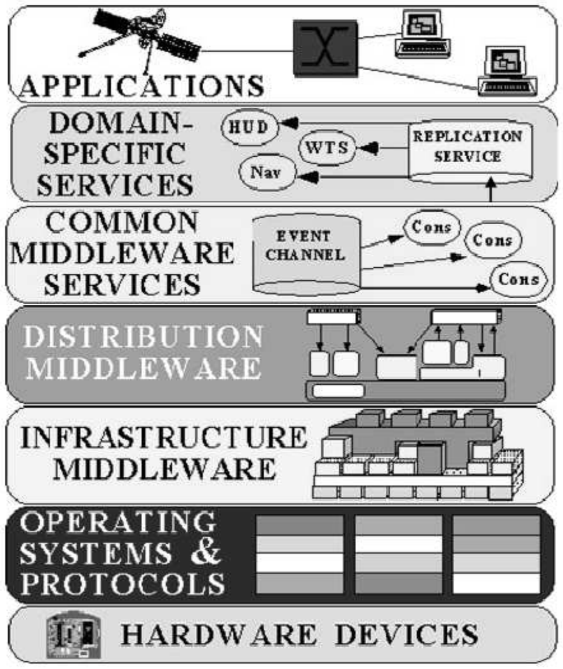


Figure 2.1: Middleware Structure

1. Accounting for variability across a range of middleware technologies:

Figure 2.1 illustrates the structure of contemporary middleware technologies. It depicts multiple layers of middleware each of which addresses specific requirements and provides reusable functional capabilities. For example, the host infrastructure middleware provides a uniform layer of abstraction to mask the heterogeneity arising from different operating systems, hardware and networks; the distributed middleware provides location transparency; common services include directory services, messaging services, and transaction services among others; and domain-specific services include additional reusable capabilities that are specific to a domain (*e.g.*, avionics or telecom).

In a networked environment, distributed applications are typically hosted on multiple heterogeneous middleware platforms. For each host in the deployment environment, the middleware stacks on which an application is hosted may need to be fine-tuned in different ways to meet the different QoS requirements of applications. To support a wide range of application QoS needs, contemporary middleware technologies provide several different reusable capabilities that can be individually configured and composed with each other. This flexibility offered by individual middleware technologies gives rise to variability that a systems integrator faces when provisioning applications on the middleware platforms.

The visual tool used for middleware provisioning must handle this variability in the context of application QoS needs, and provide an intuitive user interface to the system integrator to eliminate provisioning errors. Our approach to resolve these challenges is based on abstracting away the implementation-specific and technology-specific details of contemporary middleware solutions and focus on the patterns of reuse [16] that form the building blocks of the different layers of middleware. Section 2.2 describes how we leverage and formalize these insights to design and implement the POSAML visual tool for middleware provisioning.

2. Separation of concerns:

Each phase of middleware provisioning is filled with numerous accidental complexities. For example, the specialization and configuration phase requires the actors to make the right decisions on the selection of features and their configuration options. When confronted with multiple different middleware technologies, the actor is required to have detailed understanding of the flexibility and configurability of the middleware stack. Different middleware technologies use different data models and messaging standards which require different kinds of optimizations.

The QoS validation phase involves analytical and empirical evaluation of QoS properties. It requires the systems integrators to construct appropriate application testing and middleware benchmarking code in accordance to the configuration decisions. Many different techniques exist for such evaluations and different middleware stacks will need potentially different methods to validate QoS. Each of these phases thus incurs substantial accidental complexity as described, which is further compounded by the fact that they are all driven by the same QoS requirements of the application and decisions at one phase impact the other. There is a need to disentangle the phases at an intuitive level of abstraction that addresses the challenges incurred by the accidental complexities in each phase.

In addition to the previously discussed challenges, their is one more challenge of providing a capability to model and automatically synthesize the aspects for middleware so that there is desired functionality at an intuitive and higher level of abstraction. There is need to do AOSD modeling for automatic generation of specialization files. It is very difficult to control and manage features if the specialization files are hand written. This framework must provide the means to model aspects and its constructs so that specialization files can be generated. This challenge and the solution by adding capability of modeling AOP and synthesis of specialization files to a visual tool are discussed in detail in Chapter 4.

3. Need for a unified framework:

As noted earlier, middleware provisioning needs to be guided by the application QoS needs. This requires that the QoS validation must be performed based on decisions made in the specialization and configuration phase. Thus, the QoS validation phase needs to have complete knowledge of the earlier decisions. The QoS validation phase requires systems developers to develop appropriate application testing and middleware benchmarking code in accordance to the specialization and configuration decisions.

These requirements add a new dimension of variability and dependability to the challenges described in Section 2.1. To address these challenges, visual tools should provide a unified framework that can address the tangled concerns between multiple phases that arise in middleware provisioning. Such a framework must provide the means to capture the specialization and configuration decisions, and make them available in the QoS validation phase.

2.2 Designing Visual Tools for Middleware Provisioning

Model-Driven Environment (MDE) [21,36] has gained prominence in assisting application developers to make the right choices in configuring and composing large systems. Such model-based solutions can help resolve the variability in middleware provisioning and QoS validation. In this Section we describe a Domain-Specific Modeling Language (DSML) called POSAML (*Patterns-oriented Software Architecture Modeling Language*), which enables the modeling of middleware stacks, their configurations and aspects by providing intuitive visual abstractions of middleware building blocks. Moreover, POSAML provides middleware-specific QoS validation and generation of specialization files by virtue of plugging in different modeling interpreters.

Variability in Middleware Composition

When deploying complex applications, systems developers must decide the composition and customization of middleware that hosts the application components. Middleware composition includes assembling individual but compatible building blocks of middleware. The building blocks of middleware are most often patterns-based implementations.

A software pattern [16] codifies recurring solutions to a particular problem occurring in different contexts, which is embodied as a reusable software building block. There are several benefits of design patterns to the software community [44]:

- Design patterns offer reusable solutions to common recurring problems.
- Design patterns make communication between designers more efficient by using common terminology.
- Patterns give a high-level of perspective on the problem and on the process of design and object orientation.

The systems developer chooses a block based on various factors including the context in which the application will be deployed, the concurrency and

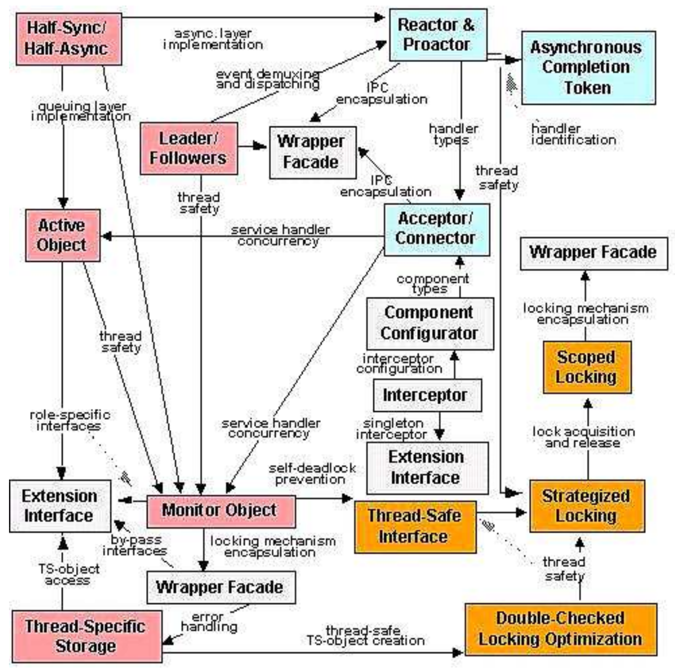


Figure 2.2: Middleware Patterns and Pattern Languages

distribution requirements of the application, the end-to-end latency, timeliness requirements for real-time systems, or throughput for other enterprise systems (*e.g.*, telecommunications call processing). We refer to this incurred design space variability as *middleware compositional variability.*

Figure 2.2 illustrates a family of interacting patterns forming a pattern language [1] for middleware designed to support such applications. In addition to these patterns there are some software design patterns which are almost part of every software system. The middleware can be customized by composing compatible patterns. For example, event de-multiplexing and dispatching via the Reactor or Proactor pattern can be composed with the concurrent event handling provided by the Leader-Follower or Active Object pattern. However, an Asynchronous Completion Token (ACT) pattern works only with asynchronous event de-multiplexing provided by the Proactor. Thus, a combination of Reactor and ACT is invalid.

Building Block Configuration Variability

Middleware developers provide numerous configuration options to customize the behavior of individual building blocks. This flexibility further exacerbates the already incurred variability in design choices that the systems developer is required to make. Any ad hoc decision affects the time-to-market and also can deliver less-than-desired performance. Since the impact is on a per building block basis – as opposed to a composition described in the previous challenge – we refer to this as *configuration dimension variability.*

As a concrete example, the Reactor pattern can be configured in many different ways depending on the event de-multiplexing capabilities provided by the underlying OS and the concurrency requirements of an application. For example, the de-multiplexing capabilities of a Reactor could be based on the `select()` or `poll()` system calls provided by POSIX-compliant or `WaitForMultipleObject()` available on Windows operating system. Moreover, the handling of the event in the Reactor's event handler can be managed by a single thread of control or handed over to a pool of threads depending on the concurrency requirements.

Our research contributions to address the challenges illustrated in previous Sections leverage the fact that standardized middleware are made up of different layers of software performing numerous activities, such as data marshaling, event handling, brokering, concurrency handling and connection management. In an object-oriented design of a middleware framework, these capabilities are realized by building blocks based on proven patterns of software design [16].

Thus a visual representation of these patterns enables systems integrators to view the middleware stacks at a higher level of abstraction that is independent of any specific middleware technology. The QoS validation mechanisms associated with the visual capability subsequently map these abstractions to specific platforms. The architectural patterns present in contemporary middleware systems are discussed extensively in the POSA (*Pattern Oriented Software Architecture: Patterns for Networked and Concurrent Systems*) book [43].

It enables modeling of the patterns described in the POSA book as well as some commonly used software design patterns [16]. In addition to pattern modeling, POSAML provides a clean separation of concerns between the specializations and configurations, generation of specialization files and QoS validation phases within the same unified framework. Besides pattern modeling, POSAML also incorporates feature, benchmark and simulation modeling.

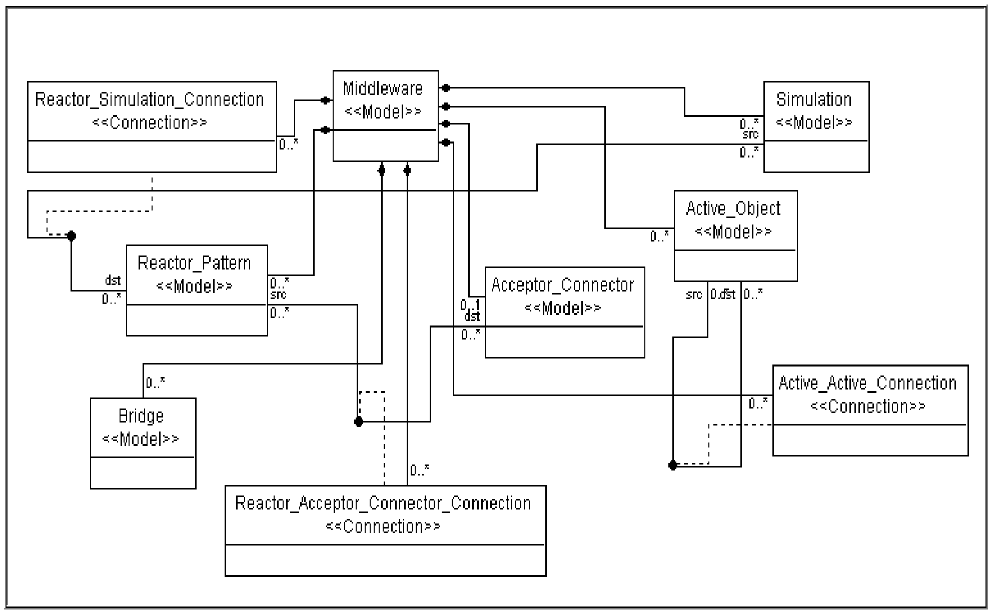


Figure 2.3: Top-level Metamodel of Middleware Structure

The POSAML modeling language has been developed using the Generic Modeling Environment (GME) [32]. Figure 2.3 shows the meta-model for the top-level view of POSAML. GME is a tool that enables domain experts to develop visual modeling languages and generative tools associated with those languages. The modeling languages in GME are represented as metamodels. A meta-model in GME depicts a class diagram using UML-like constructs showcasing the elements of the modeling language and how they are associated with each other. For example, the “model” element defines an element that can comprise other elements. The “Aspect” element describes a specific view provided by the modeling environment. By providing such views, the modeling environment effectively allows visual separation of concerns. The “Connection” element describes the type of association between other modeling elements of the language. The GME environment can be used by application developers to model examples that conform to the syntax and semantics of the modeling language captured in the metamodels.

The meta-model illustrated in Figure 2.3 consists of the visual syntactic and semantic elements that describe individual patterns, and specifies how they can be connected to each other. This Figure also illustrates how the meta-model separates the concerns of modeling the pattern, its configuration and their compositions, and system QoS validation.

Figure 2.3 shows the meta-model for the top-most view of pattern modeling. This meta-model consists of the individual pattern models, and specifies how they can be connected to each other. In addition to this, this meta-model also defines the four aspects:

a. **Pattern Aspect:** The pattern aspect is where a system modeler can compose and model the various patterns in the system. Figure 2.4 shows an example where the designer has modeled the Reactor, Acceptor-Connector, Active Object and Bridge pattern. In addition to this high-level view, the user can click on any one of the patterns and model its internals, as shown in the Figure. Section 2.3 and Section 2.4 describes in depth about pattern modeling of the Acceptor-Connector and Bridge patterns respectively.
   From Figure 2.4, it can be seen that POSAML follows a hierarchical structure. At the top-most level one can model inter-pattern relationships and constraints. At the lower level, a designer can drill down “inside” each pattern to model the participants of the pattern and the intra-pattern relationships between them.
b. **Feature Aspect:** The system designer can set various features of each pattern in the feature aspect of POSAML. For example, the designer

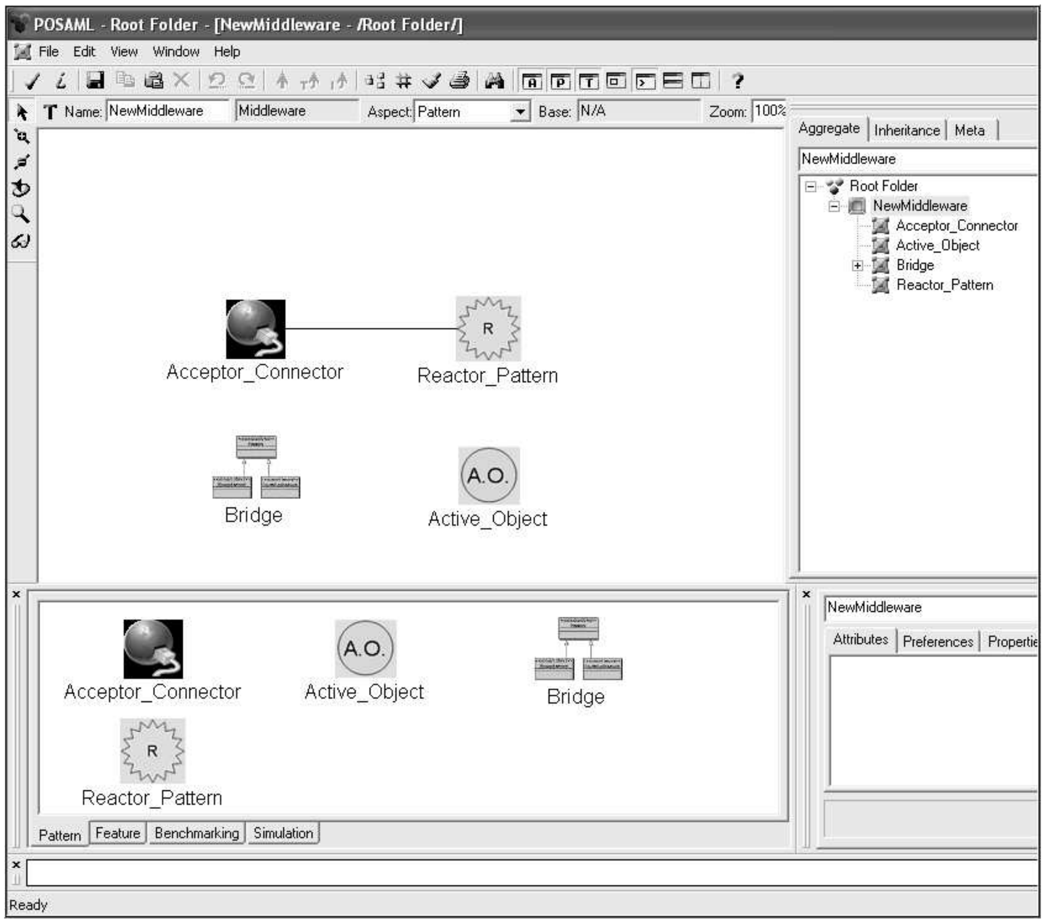


Figure 2.4: Overview of POSAML

can specify the "End-points" feature for the Acceptor-Connector Pattern. The features are written to a configuration files by the Feature interpreter. This configuration file can be used to change the configuration of the middleware system. Section 2.5 discusses the Feature Aspect of POSAML in more detail.

c. **Benchmarking Aspect:** The designer can select which benchmarking parameters to set for the performance analysis of the modeled system. This aspect is out of scope of this work and is described in detailed in [24]

d. **Simulation Aspect:** The designer can model different simulation parameters needed to evaluate trade-offs between configurations. This aspect is out of scope of this work.

The rest of the Section describes the pattern aspect, which allows a systems engineer to model a middleware building block comprising the individual patterns. We focus on a subset of the POSA and simple software design patterns describing the visual modeling elements provided in pattern modeling.

In Section 2.3 and Section 2.4 we illustrate the use of POSAML to model Acceptor-Connector and Bridge patterns and Section 2.5 describes how POSAML provides the provisioning and QoS capabilities by modeling features

2.3 Acceptor-Connector Pattern

The Acceptor-Connector pattern pair decouples connection establishment and service initialization in a distributed system from subsequent processing performed by the two end points of a service once they are connected and initialized [38]. This decoupling is achieved by acceptors, connectors, and service handlers. Connection-oriented protocols provide reliable delivery of data between two or more end points of communication [43]. Establishing connection between end points involves two roles:

- *Passive role*: This type of role initializes end points of communication at a particular address and waits passively for other end points to connect with it.
- *Active role*: This type of role actively initiates a connection to one or more end points that are playing the passive role.

It is important to know the communication role in terms of server and client. Most often the client plays an active role in connecting with a passive server. Thus, in the Acceptor-Connector pattern, the acceptor provides the passive role and the connector plays an active role.

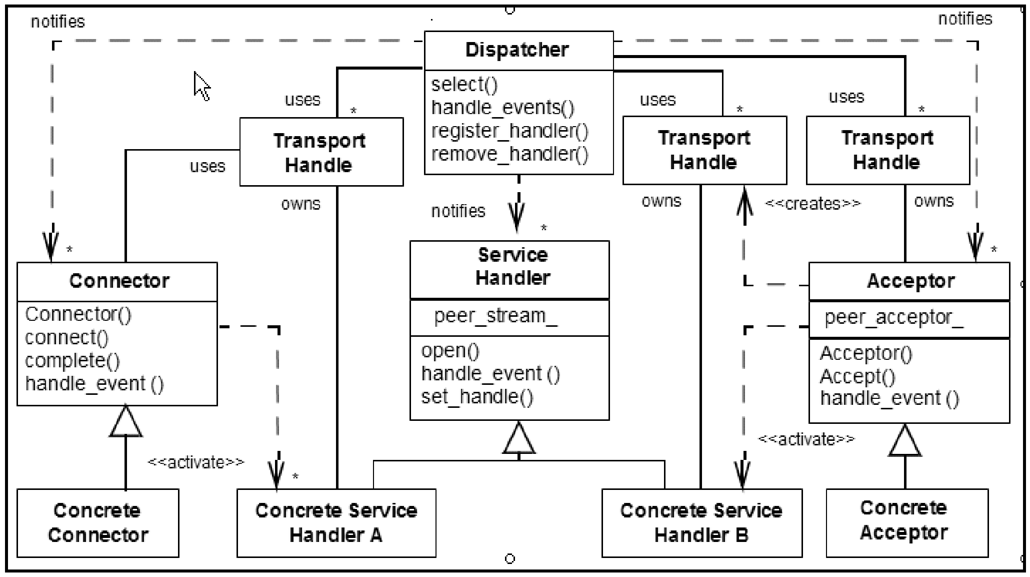


Figure 2.5: UML diagram of Acceptor-Connector Pattern

Figure 2.3 shows the UML structure of the acceptor and connector pattern. Based on this structure we designed the Acceptor-Connector meta-model.

2.3.1 Metamodel of Acceptor-Connector Pattern

In order to understand the detailed relationship between various participants of Acceptor-Connector pattern, we will discuss different phases that are in this pattern. This pattern is actually composed of two patterns called Acceptor and Connector. The Connector pattern contains options to have synchronous or asynchronous type of connection but in POSAML we are supporting only synchronous mode of connection.

Now we will discuss the collaboration between server and client among different participants of Acceptor and Connector patterns at a very low-level of abstraction. Both the Acceptor and Connector pattern meta-model is part of a single paradigm sheet called Acceptor-Connector. In order to understand the Acceptor-Connector meta-model and model properly we have simplified it by dividing the meta-model into two parts. These two parts are shown in Figure 2.9 and 2.7 and are based on the UML diagram of Acceptor-Connector shown in Figure 2.3.

In Figure 2.7 we can see that the root element of this meta-model is Acceptor-Connector proxy model and it is a proxy of Acceptor-Connector model, which is a part of our basic paradigm meta model called middleware shown in Figure 2.3. Acceptor-Connector model proxy is in equivalence relation with local Acceptor-Connector model. Dispatcher atom and Reactor atom proxy are equivalent and these are basically proxies of Reactor atom of Reactor pattern [24]. The Dispatcher atom acts as event handler for both abstract Acceptor atom and Connector atom.

Next we will discuss meta-model of Connector and Acceptor separately:

Collaboration of Connector:

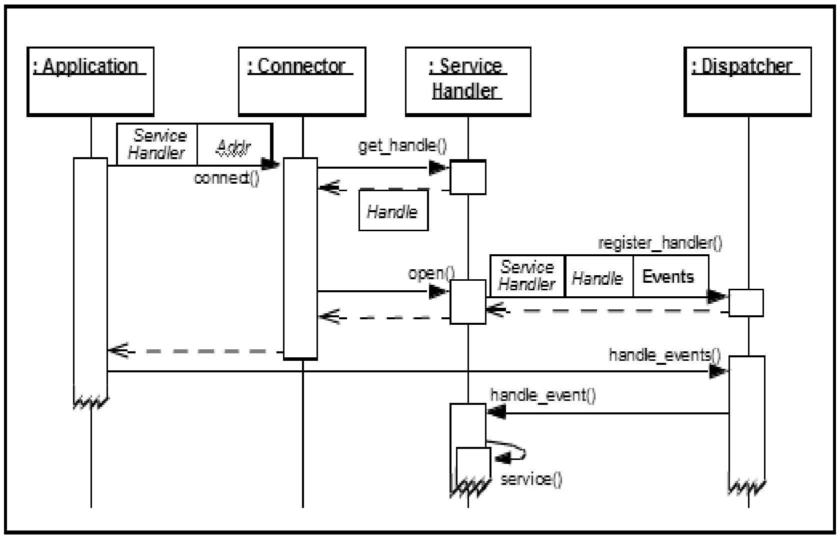


Figure 2.6: Connector Pattern Dynamics

The behavior and flow of connector can be seen in Figure 2.6. The collaboration between the participants in the synchronous Connector scenario can be divided into following stages:

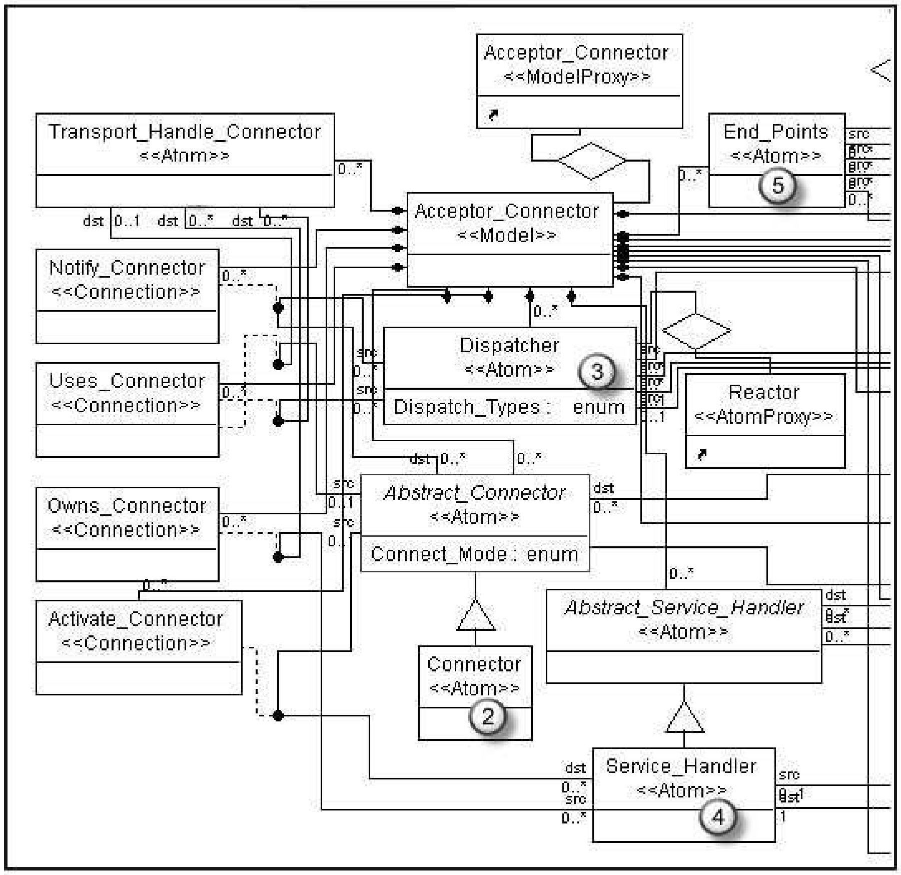


Figure 2.7: Metamodel of Connector Pattern

a. **Connection & Service initialization phase:** In Figure 2.7 a client creates a Dispatcher atom that handles the events associated with that client. Then it creates one concrete Connector atom derived from Abstract_Connector atom. It also activates concrete Service_Handler atom that is derived from Abstract_Service_Handler atom. This will be in charge of requesting the server peer service handler for each type of service needed using Activate_Connector connection. For this, then the Connector atom generates an event and sends it on the Transport_Handle_Connector atom of the server Acceptor of Figure 2.9 in charge of the desired service. In return, the Connector atom gets a connection using End_Point atom that corresponds to the Transport_Handle_Connector atom of the peer Service_Handler atom. This End_Points atom is again equivalent to End_Points atom proxy from the feature meta-model 2.5. This atom has three attributes like host name, listening port, and the protocol. It can then register itself to the Dispatcher atom.

b. **Service processing phase:** When the Dispatcher atom detects an event on the Transport_Handle_Connector atom of the client Service_Handler atom, it notifies the handle event method of this Service_Handler atom using Notify_Connector connection. When the Dispatcher atom detects a ready event on the client Service_Handler atom, it calls its handle event method. The Service_Handler atom then obtains the results from the server. If a client Service_Handler atom wants to close the service, it generates a close event on its server Service_Handler atom Handle. It is then removed from the list of Dispatcher atom by calling the remove handler method.

Collaboration of Acceptor:

The behavior and flow of acceptor can be seen in Figure 2.8. Acceptor component provides a means for passive connection establishment. The collaboration between Acceptor and service handler participants are divided into three phases:

a. **Initialization phase:** In Figure 2.9 we can see the meta-model of the acceptor part. The Acceptor atom is derived from Abstract_Acceptor atom. It acts as a server and initializes a connection passively. When an application calls the open method on Acceptor atom.

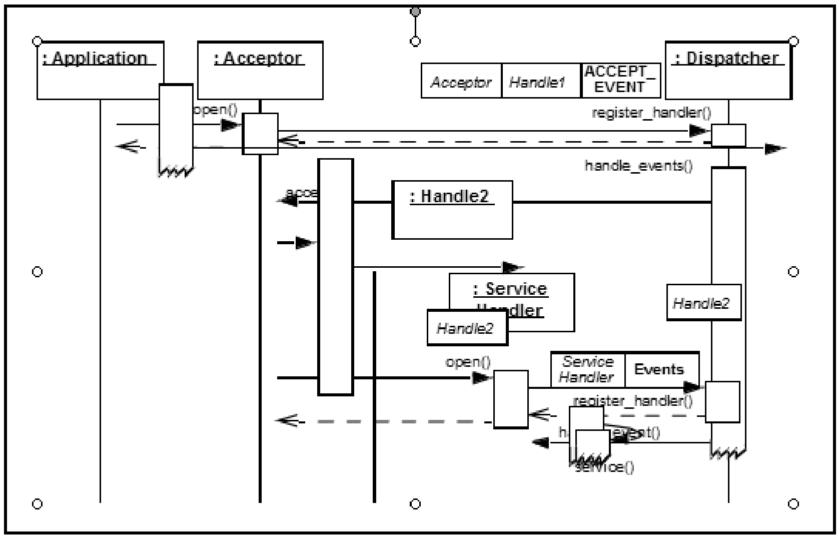


Figure 2.8: Acceptor Pattern Dynamics

This method creates a passive mode `Transport\_Handler\_Acceptor` atom which encapsulates `End\_Points` atom. It binds to end points *i.e.*, IP address, TCP Port and Protocol and then like a server it listens to connection requests from peer Connectors. These `End\_Points` are used by `Transport\_Handlers` using `End\_Point\_Connection` connection proxy obtained from feature view of POSAML.
`Transport\_Handler\_Acceptor` is owned by Acceptor by using `Owns\_Acceptor` connection. After this the open method registers the Acceptor atom object itself to the Dispatcher atom. The Dispatcher atom is equivalent to the Reactor proxy atom. This Reactor proxy atom and `AbstractEvenHandler` proxy atom are defined in Reactor pattern in [24] as Reactor atom and `AbstractEventHandler` atom. After this the Dispatcher atom can notify using `Notify\_Acceptor` connection to Acceptor atom whenever a connection event is coming from a client on the `Transport\_Handle\_Acceptor` atom.

b. **Service handler initialization phase:** The Dispatcher atom detects some event on the `Transport\_Handle\_Acceptor` atom of some Acceptor. It then calls the handle event method of this Acceptor atom for it to handle this particular event. Acceptor atom uses passive mode transport endpoint to create a new `Transport\_Handle` atom using

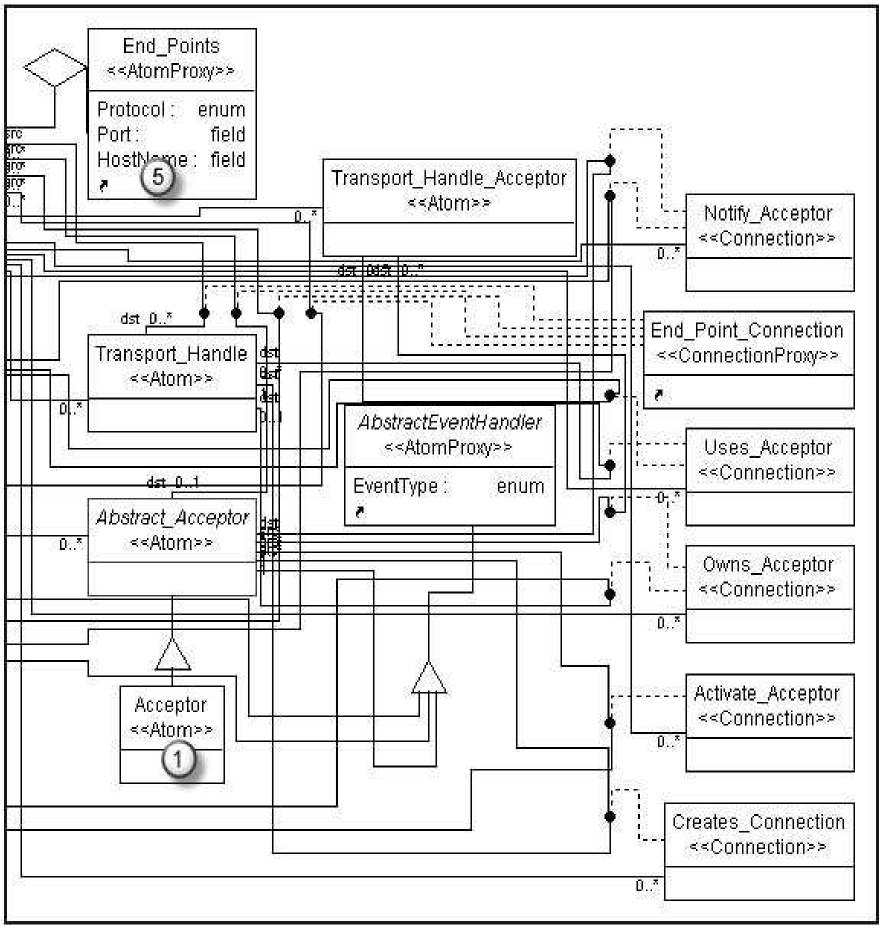


Figure 2.9: Metamodel of Acceptor Pattern

`Create\_Connection` connection. It next activates a new `Service\_Handler` atom using `Activate\_Acceptor` connection that will be in charge of processing the requests of the client. This `Service\_Handler` stores these new `Transport\_Handles` atom information and registers itself to the Dispatcher atom.

c. **Service processing phase:** This stage is about the service usage and closure. After the above two stages, the Dispatcher atom will directly initiate server client communication using `Service\_Handler` atom. In this stage, exchange of data between peers can happen directly using already connected `Transport\_Handles` (using transport endpoints). After the whole process is complete, `Service\_Handler` atom will call the remove handler method of the Dispatcher atom to make it stop listening on its `Transport\_Handle` for the events he had registered before.

2.3.2 Modeling of Acceptor-Connector Pattern

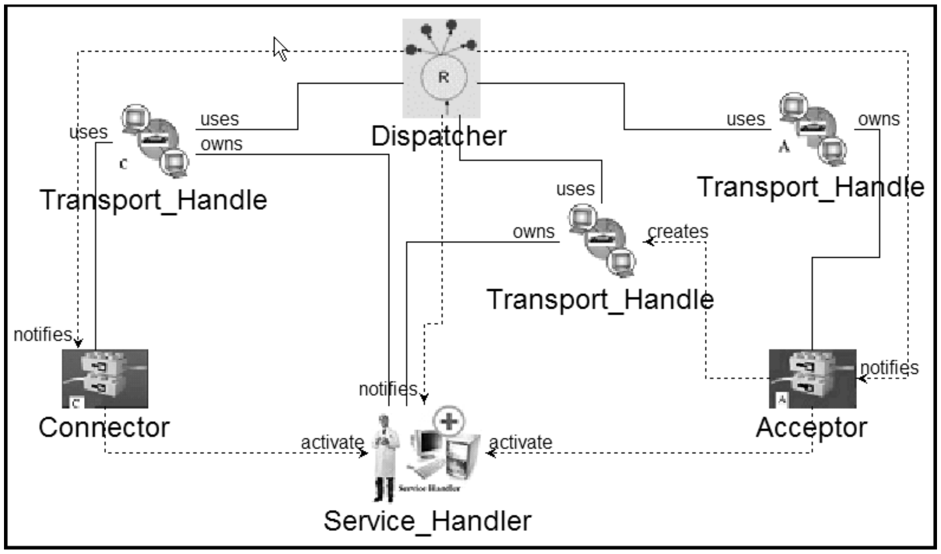


Figure 2.10: Model of Acceptor-Connector Pattern

A system engineer can model the Acceptor-Connector pattern in POSAML for the sample application as shown in Figure 2.10. Various constraints minimize the risk of choosing a wrong combination of elements in the pattern.

Only the correct combinations of connections and features are allowed for a particular pattern. For example, only the “End Point” feature can be added to the Acceptor-Connector pattern. The middleware provisioner models the following participants of the Acceptor-Connector pattern:

a. **Acceptor**: The Acceptor is a factory that implements a passive strategy to establish a connection and initialize the associated Service Handler. It creates a passive mode end point transport handle that has necessary end points needed by the Service Handlers.
b. **Connector**: A Connector is a factory that implements the active strategy to establish a connection and initialize the associated Service Handler. It initiates the connection with a remote Acceptor and has synchronous mode (using the Reactor pattern) and asynchronous mode (using the Proactor pattern) connections.
c. **Dispatcher**: The Dispatcher manages registered Event Handlers. In case of the Acceptor, the Dispatcher de-multiplexes connection indication events received on transport handles. Multiple Acceptors can be registered within the Dispatcher. For Connector, the Dispatcher demultiplexes completion events that arrive in response to connections.
d. **Service Handler**: A Service Handler is an abstract class that is inherited from Event Handler. It implements an application service playing the client role, server role or both roles. It provides a hook method that is called by an Acceptor or Connector to activate the application service when the connection is established.
e. **Transport End points**: These represent a factory that listens for connection requests to arrive, accepts those connection requests, and creates transport handles that encapsulate the newly connected transport end points. By using these end points data can be exchanged by reading or writing to their associated transport handles. A transport handle encapsulates a transport end point.

2.4 Bridge Pattern

The intent of Bridge pattern shown in Figure 2.11 is to decouple an abstraction from its implementation so that the two can vary independently [16].

This pattern is the most commonly used structural design pattern. In terms of the basic intent of the Bridge Pattern, abstraction refers to how dif-

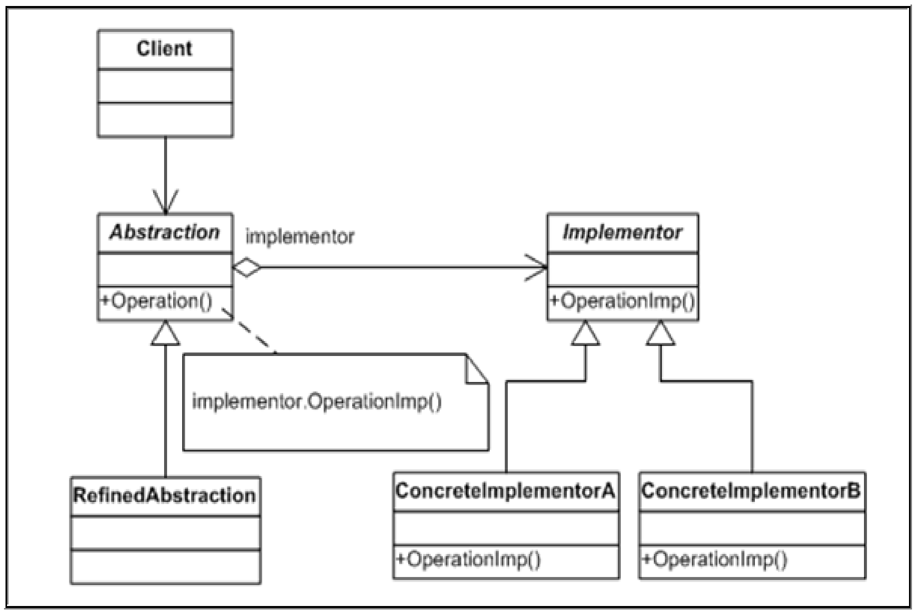


Figure 2.11: GoF UML diagram of Bridge Pattern

ferent things relate to each other, and the implementator is the object that the abstract class and its derivation use to implant themselves with [44]. The purpose of this pattern is to the information hiding principle *"The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules"* [35].

2.4.1 Metamodel of Bridge Pattern

By tailoring bridge pattern 2.11, we designed its meta-model as in Figure 2.12.

Various participants in the bridge meta-model are:

- Abstraction: It defines the abstract interface that is used by the client for interaction with the abstraction. It is also used to maintain implementor reference. The collaboration between the objects and patterns is such that the client requests are forwarded by the Abstraction to the implementor through this reference [16].

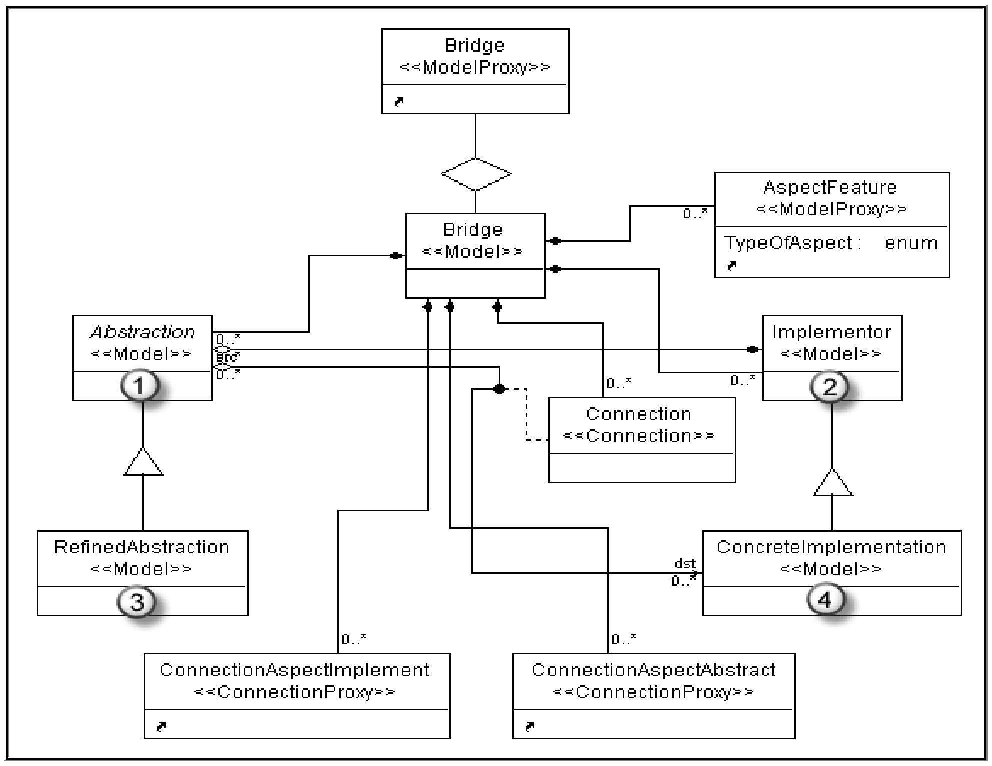


Figure 2.12: Metamodel of Bridge Pattern

- Implementor: It defines the interface for any or all implementation of the Abstraction. There is no requirement that the Abstraction interface and the Implementor interface have a one-to-one correspondence. This is one of the main reasons for the additional flexibility gained from this pattern. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives [16].

- Refined Abstraction: It extends the interface defined for Abstraction class.

- ConcreteImplementor: It implements the interface defined by the Implementor class. In other words, it defines a concrete implementation of the Abstraction.

2.4.2 Modeling of Bridge Pattern

The system modeler can model the Bridge pattern in POSAML. The two main essential elements that can be modeled are Abstraction, which is responsible for initiating any operation, and Implementation, which is an object responsible to carry out an operation.

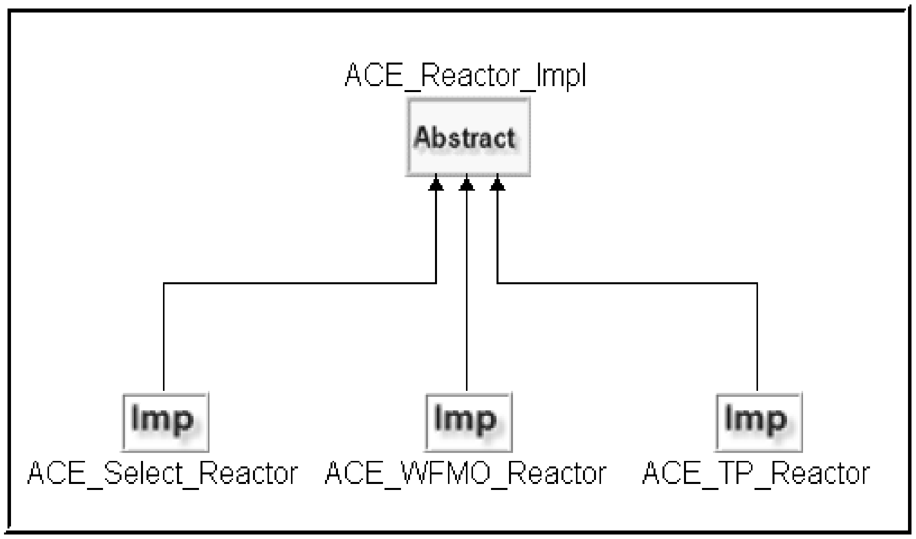


Figure 2.13: Model of Bridge Pattern

Figure 2.13 shows a sample application of modeling Bridge pattern. In this model we can see that the abstract base class `ACE_Reactor_Impl` and concrete implementation of subclasses like `ACE_Select_Reactor`, `ACE_WFMO_Reactor` and `ACE_TP_Reactor`. In Chapter 4 we describe how to model specialization aspects for this particular example and how automatic generation of the specialization file is possible using modeling interpreters [32].

2.5 Feature View in POSAML

A Feature model [11] is defined as an abstraction of a family of systems in a particular domain capturing commonalities and variabilities among the members of the family. In POSAML, a feature modeling aspect provides domain-specific artifacts to model a system, in contrast to using low-level platform-specific artifacts. The feature modeling capabilities in POSAML provide structural representations of different possible middleware pattern properties. In our case, the feature modeling comprises several para-functional requirements such as the choice of network transport, listening end points, concurrency requirements, and periodicity of requests, all represented as higher-level artifacts.

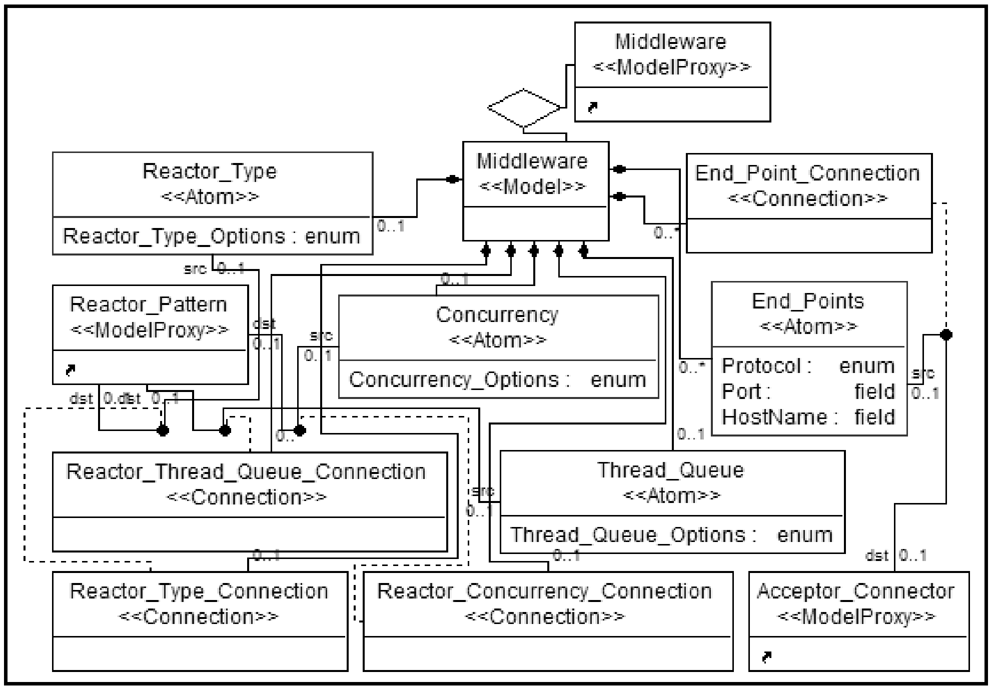
This level of modeling enables system provisioners to select various strategies, resource settings, and factories within the middleware that can be parameterized according to user needs by driving the selection process using the visual feature modeling framework. For example, the designer can specify the "End points" feature for the Acceptor-Connector Pattern to describe the ports and communication mechanisms used by the client and server to communicate with each other.

The middleware configuration is accomplished through POSAML's feature modeling [5, 11] capability, which assists a systems engineer in configuring a variety of different middleware features (*e.g.*, choosing the pattern and its configuration parameters). The traditional middleware architectures suffer from insufficient model level reusability. This model aspect of POSAML brings a way to set behavioral of middleware.

Developing a feature model out of a meta-model involves defining valid entity and relationships in the schema. All these features are pattern specific. For example Concurrency, Thread Queue, and Reactor Type are Reactor specific, Active Object map size is Active Object pattern specific and End points are Acceptor and Connector pattern specific.

2.5.1 Metamodel of Feature View

This part of the Section discusses about the GME meta model of feature aspect of POSAML shown in Figure 2.14. Some of the features designed in the meta model are as following:



Middleware <<ModelProxy>>
Reactor_Type <<Atom>>
Reactor_Type_Options : enum
Middleware <<Model>>
End_Point_Connection <<Connection>>
Reactor_Pattern <<ModelProxy>>
Concurrency <<Atom>>
Concurrency_Options : enum
End_Points <<Atom>>
Protocol : enum
Port : field
HostName : field
Reactor_Thread_Queue_Connection <<Connection>>
Thread_Queue <<Atom>>
Thread_Queue_Options : enum
Reactor_Type_Connection <<Connection>>
Reactor_Concurrency_Connection <<Connection>>
Acceptor_Connector <<ModelProxy>>
src
dst
0..1
0..*

Figure 2.14: POSAML Metamodel: Feature View

a. **Concurrency:** This feature is important for different middleware to manage concurrency and allow long running operations to execute simultaneously without impeding the progress of other operations. It specifies the concurrency strategy an ORB uses. Concurrency has `Concurrency\_Option` as attribute and this attribute has two strategies as menu items. Server concurrency strategies found in contemporary middleware solutions, such as the TAO CORBA middleware [42], support different types of concurrency strategies
(1) *Reactive* : This is a default concurrency type. It registers the connections transport end points with Reactor. When events occur, it starts dispatching these events to the reactive connections in order. In this mechanism, an ORB handles each request reactively *i.e.*, the ORB

runs in one thread and service multiple requests/connections simultaneously using the `ACE_Reactor`, which uses select or a similar event demultiplexing mechanism supported by the platform. Its connections are single-threaded and that is it is scalable but is not a very good concurrency mechanism.
(2) *Thread Per Connection* : The ORB handles new connections by spawning a new thread whose job is to service requests coming from the connection.

b. **Reactor Type:** This feature is used to specify the kind of reactor used by the system. For example, depending on the concurrency strategy chosen, the reactor could be single-threaded or multi-threaded.
It is an advanced resource factory option and if this type of factory is loaded, any default directives of original resource factory will have no effect. Different strategies can be plugged within the reactor for event demultiplexing. This depends on whether the reactor is used to demultiplex network events or GUI events. `Reactor\_Type` has `Reactor\_Type\_Option` as attribute and this attribute has five strategies as different reactor types:
(1) Thread pool (tp) : It uses the `ACE_TP_Reactor`, a select based thread pool reactor which is the default.
(2) Multi threaded (select_mt) : It uses the multi-thread select based reactor.
(3) Single threaded (select_st) : It uses single-thread select based reactor.
(4) Wait for multiple objects (wfmo) : It uses WFMO reactor and can be used only for windows.
(5) Message Wait for multiple objects (msg_wfmo) : It uses Msg WFMO reactor. It is used only for Windows.

c. **Thread Queue:** In the case of concurrent request handling by a reactor, different strategies can be selected for handling queued events (*e.g.*, FIFO or LIFO). Thread Queue has an attribute called `Thread\_Queue\_Options`. This attribute has two options *i.e.*, first-in-first-out (FIFO) and last-in-first-out (LIFO). This feature applies only to the `ACE_TP_Reactor`, *i.e.*, when the reactor type is thread pool. This specifies the order, LIFO, the default, or FIFO, in which waiting threads are selected to run by the `ACE_Select_Reactor_Token`. Thread Queue is also a part of an advance resource factory.

d. **End points:** This feature applies to the acceptor-connector patterns, which instructs the system of the listening end points for the server role. The range of available end points in POSAML include listening ports (*e.g.*, TCP port number), host IP addresses or canonical names, and the protocol used (*e.g.*, TCP, UDP, Shared Memory or other custom transports).

2.5.2 Modeling of Feature View

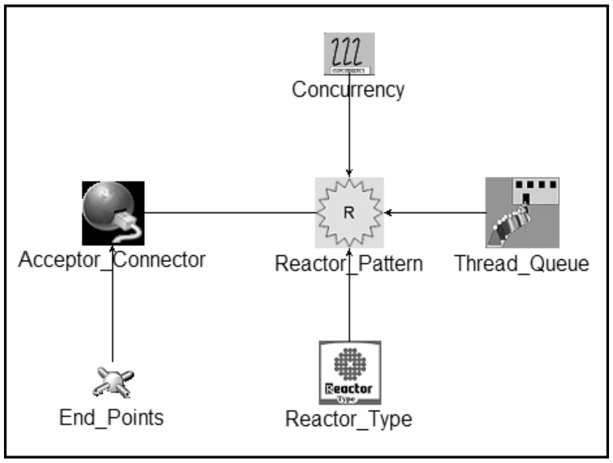


Figure 2.15: POSAML Model: Feature View

A middleware developer uses the feature aspect of POSAML as a visual tool to select different pattern-specific features of middleware. In Figure 2.15 we can see the how modeler can model various features in this modeling language. A modeler can model zero or more features using this tool. Once the feature modeling part is completed, then the next step is to transform pattern-specific features into a configuration file using model interpreters. If features are not selected from the model, default values of these features will be picked.

In order to minimize the risk of choosing wrong connections and options, various constraints. Some of these constraints are checked using OCL constraint language *i.e.*, checking for non-null references or proxies and some of them are checked at the time of execution of interpreters, *i.e.*, when we interpreter a model that time we check if a feature is connected to correct pattern or not. The selected features set with different options are exported into files using configuration interpreter and are used to configure middleware system . We will discuss the configuration model interpreter in Section 6.1.

Chapter 3

Middleware Specialization

3.1 Various Specialization Techniques

Chapter 1 motivated the need for specializing middleware to suit 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 [18]. 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. Owing to all these drawbacks of above specializations, they are not best suited for the product variants of product lines which may have a specific set of performance requirements.

There are various specialization techniques described in the literature, which can be leveraged to specialize middleware. For example, Feature-Oriented Programming (FOP) [5, 7] is an appropriate technique to design and implement program families, and which uses incremental and stepwise refinement approaches [7, 49]. FOP aims to cope with the increasing complexity and increasing lack of reusability and customizability of contemporary software systems. Aspect-Oriented Programming (AOP) [26] 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.

	AOP	**FOP**
1	Lack of Stepwise Refinement	Stepwise Refinement
2	Homogeneous concerns	Heterogeneous concerns
3	Non Hierarchy-Conform refinement	Hierarchy-Conform refinement
4	Cross cutting modularity	Lack crosscutting modularity
5	Excessive method extension	Higher level of abstraction
6	Power of quantification	Same as OO framework
7	Hard to Implement	Simple to Implement
8	Used in Industry	Concept in academics
9	Java, C, C++, Perl	Only Java

Table 3.1: Difference between AOP and FOP

Table 3.1 lists various differences between Aspect-Oriented and Feature-Oriented Programming techniques.

3.2 Overview of Aspect-Oriented Programming

Without causing any intrusive changes to the entire code base, AOP technology helps modularize the implementation, and helps reduce dependencies between modules [48]. The currently most used tools are AspectJ [3], AspectC++ [49], AspectWorkz, JBoss AOP and Spring AOP. Almost for every programming language there is an aspect-oriented programming tool. AOP principles supported by such tools address the challenges of crosscutting concerns which pure OO methods do not. According to [25] all these tools are built on similar principles, which are Advice, Aspect, Joinpoint and Pointcut. Using pointcuts and advice, an aspect weaver brings aspects and components together. An advice defines the code that is defined on these joinpoints.

1. **Advice:** This is the code that is applied to, or that crosscuts the existing code. 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. Different points in the code where aspects can be woven 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.

When we are using Aspect-Oriented Programming, we can write aspect code in aspect files and in most of the cases we do not have to modify primary concern or main business logic classes. This makes the code flexible, extensible and less error prone. AOP is the best way 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.

3.3 Aspect-Oriented vs. Object-Oriented Programming

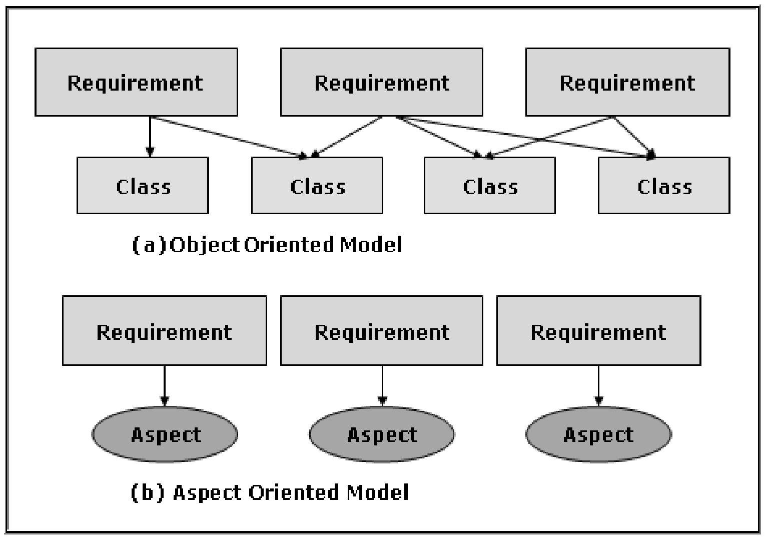


Figure 3.1: Comparing Object-Oriented and Aspect-Oriented Model

Aspect-Oriented refactoring [10] 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 3.1, 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 3.2 illustrates the different aspect-oriented development phases that can be applied to already existing software systems. The first phase identifies a list of various secondary concerns such as transaction control, security, and logging as described in Chapter 5. These secondary concerns

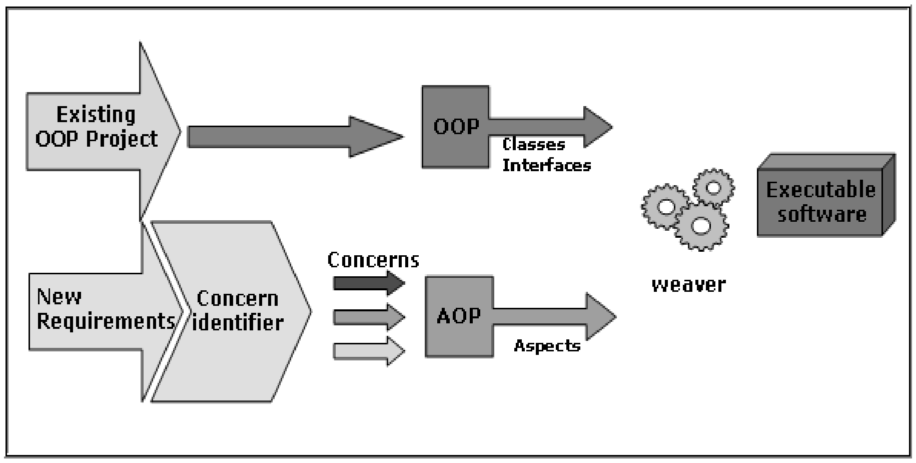


Figure 3.2: Phases of AOSD for an existing project

can be different specialization which are discussed in the next Section. In the next phase, these secondary concerns are implemented separately using aspect-oriented techniques. Finally, an aspect weaver weaves these aspects with the object-oriented classes of the already existing project.

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. This specialization technique will be helpful only if features are selectable based solely on the various middleware strategies or specifications that will fulfill user requirements.

3.4 Approach to Specialize Middleware via AOP

In this Section we explore the use of Aspect-Oriented Programming (AOP) incorporated by the AspectC++ [49] tool to automate the middleware specializations. For this work we chose the ACE C++ middleware [39] as the platform to demonstrate our ideas.

Because the size of the aspect code is less and this code is totally isolated from actual source code, their management is relatively easy, less error prone and easy to plug and play. All this was possible without making any change to the actual code base. Source code transformation, *i.e.*, weaving is done based on aspects at compile time using the AspectC++ compiler (ag++ of version 1.0pre2). 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.

Middleware is often developed as a set of frameworks that can support and is portable for all the platforms and supports large number 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).

3.5 Reactor Specialization using AOP

In this Section, we describe our 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. 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.

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 as show in Figure 3.3. ACE middleware framework supports several implementations of Reactor pattern. (1)`ACE_Select_Reactor`– single-thread event demultiplexer , (2)`ACE_TP_Reactor`– multi-threaded event

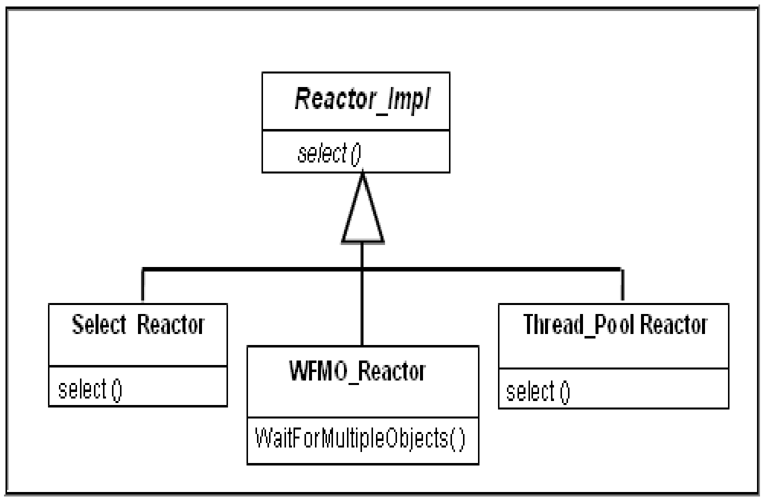


Figure 3.3: Reactor Specialization using AOP

demultiplexer and (3)`ACE_WFMO_Reactor`– windows event demultiplexer. The OOP 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.

As proof of concept we will focus on two types of concurrency models *i.e.*, single threaded and thread-pool reactor. 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 and Bridge pattern.

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.

A product variant may need only one implementation at run-time. 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. This way the application specific reactor implementation method is called directly. This specialization removes the generality penalty by removing the extra indirection caused by virtualness.

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

```
/**
  * Aspect for Single Threaded specialization
  */
aspect Single_Thread_Aspect
{
  /**
    * 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 >());
  }
}
```

Figure 3.4: Specialization file for Single threaded reactor

In the code snippet shown in Figure 3.4, it shows the single thread implementation of ACE reactor. Here we are redirecting method `purge_pending_notification` of abstract class to the same method name of concrete implementation directly. It should be noted that this method is called almost 16 times in a single server/client scenario.

Similar transformations are achieved in the code snippet shown in Figure 3.5. It shows that the threadpool implementation for ACE reactor for other method. 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.

```
/**
  * Aspect for Thread Pool specialization
  */
aspect TP_Thread_Aspect
{
  /**
    * 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 >());
  }
}
```

Figure 3.5: Specialization file for Thread Pool reactor

Chapter 4

Automating Generation of Specialization Aspects

Visual modeling and Aspect-Oriented Programming Software Development are two different software engineering paradigms which have been developed independently. In this Chapter we argue for their integration to address the challenges of middleware specialization. We added a capability in POSAML to model aspects and used the modeling interpreter to automate generation of specialization file for middleware. Automating the generation of using modeling is an important feature because the solution and the description of a feature will be localized and hence will be easy to control and manage.

This capability of POSAML to specify or model different AOP constructs like aspects, pointcuts, and advice is used to generate specialization files. These specialization files, in turn are used to optimize middleware system. Modeling of AOP design is similar as that of normal OO design and the only difference is the way the variables are constrained. For every secondary concern an aspect can be modeled and specialization files can be generated.

4.1 Metamodel of Aspect for POSAML

Figure 4.1 shows the metamodel of the Aspects for POSAML. In this metamodel `AspectFeature` model is the specialization aspect feature we want to model. For every feature or specialization we can have an `AspectFeature`. It has an attribute for specifying the type of aspect. There are some fixed types of aspects, like Logging, VerifyAccess and RemoveVirtualness. While

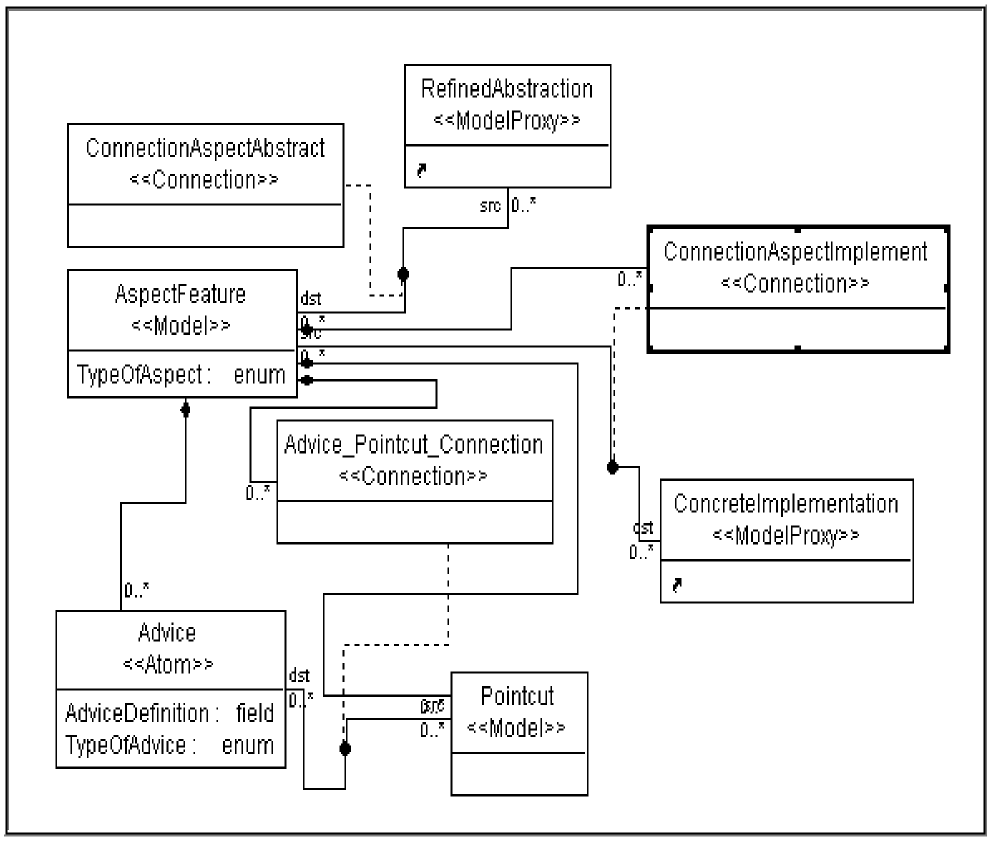


Figure 4.1: Metamodel of Aspect for POSAML

modeling aspects in our modeling tool we can select one of these types of aspects or generate our own aspect feature. These types of aspects would provide more flexibility for our future work. Every `AspectFeature` has an `Advice` atom and a `Pointcut` model. A `Pointcut` model can be connected to an `Advice` atom. An advice can have only one pointcut and an aspect can have multiple advice. Multiple joinpoints can be linked together using different logical operators to form a pointcut. The detailed description of modeling a pointcut is discussed in Section 4.2.

4.2 Modeling of Aspect for POSAML

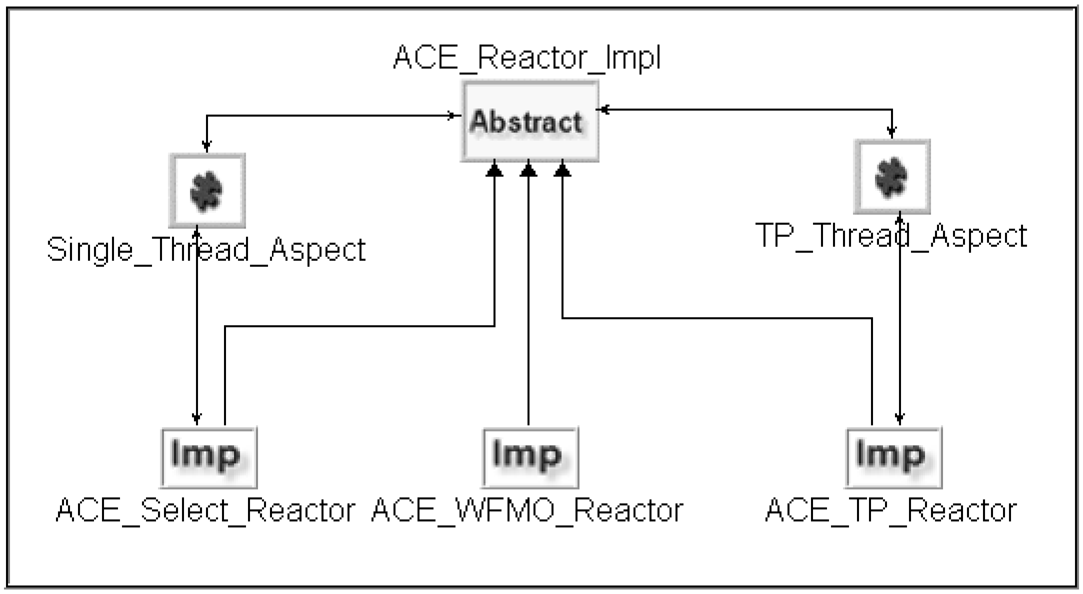


Figure 4.2: Modeling of Aspect in POSAML

Figure 4.2 shows an example of how we can model the bridge pattern with the different specialization or aspect features. In this example we have modeled abstract base class `ACE_Reactor_Impl` with its concrete implementation of subclasses like `ACE_Select_Reactor`, `ACE_WFMO_Reactor` and `ACE_TP_Reactor`. It also shows two modeled specialization aspects: these aspects are named as `Single_Thread_Aspect` and `TP_Thread_Aspect`.

`Single_Thread_Aspect` specialization is for single threaded implementation of Reactor and `TP_Thread_Aspect` specialization is for threadpool implementation of Reactor as discussed in Section 3.5.

Modeling of constructs

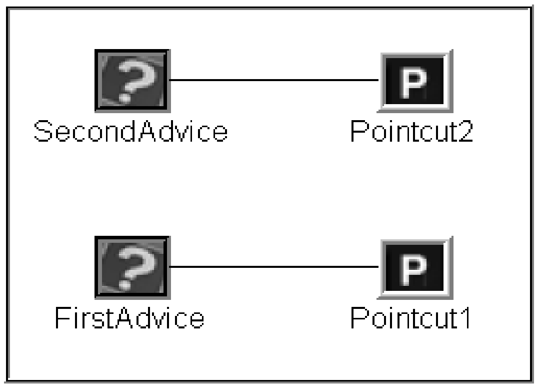


Figure 4.3: Modeling of Aspect constructs in POSAML

Delving deeper into the model reveals that the modeled aspect contains different constructs. Some of these constructs are advice and pointcut. In order to model aspects, we have to model these constructs also. These constructs vary according to the functionality of the feature. An aspect can have more than one advice and each advice has a pointcut connected to it as shown in Figure 4.2. Advice has only one attribute to define advice code and this attribute is variable for all other features. There is a constraint that a particular advice can have only one pointcut at a time. In case we want to have more than one pointcut for an advice, we can design that while modeling pointcut.

Metamodel and Model of Pointcut

Figure 4.4 shows the meta-model of pointcut. The pointcuts can be designed by having different set of joinpoints having different logical operator relationships (AND, OR) between them. This modeling tool provides the capability to design very complex pointcuts. In Figure 4.5, we have given a sample model of pointcut. In this example there are eight separate joinpoints (a, b, c, d, e, f, g, h) with different joinpoint equations. For ever joinpoint we can

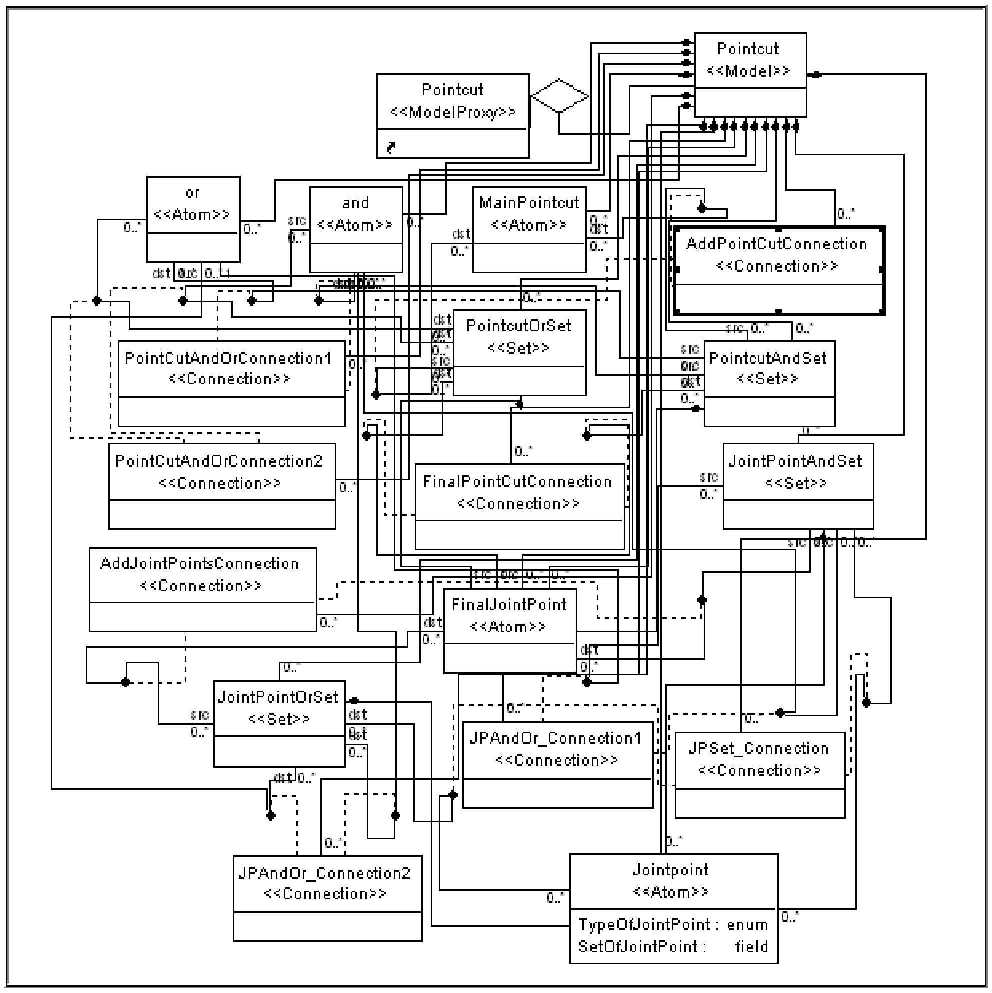


Figure 4.4: Metamodel of Aspect construct Pointcut

set two attributes (1) Type of joinpoint *i.e.*, call, execute (2) Equation of the joinpoint.

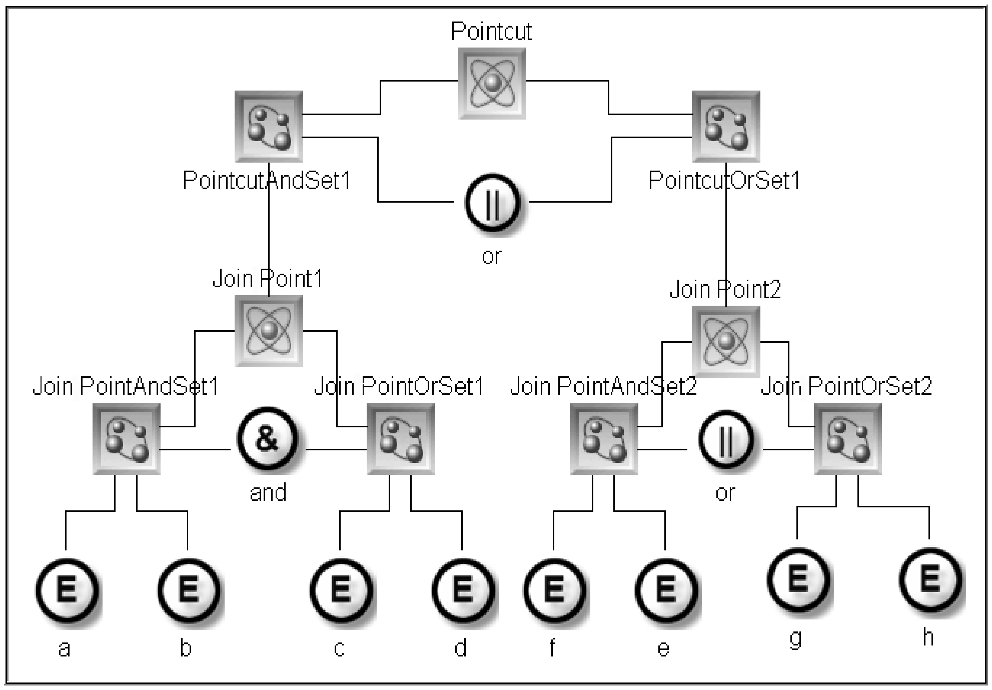


Figure 4.5: Example of Pointcut Model

There are four sets in the model. Two of the sets are for the joinpoint *i.e.*, JoinPointSetAND and JoinPointSetOR and two are for pointcut sets *i.e.*, PointcutSetAND and PointcutSetOR. These are there to manage the logical operation relationship between different joinpoint and pointcut sets.

The joinpoint equations combine according to needed logical operator relationship between them by linking these individual joinpoints to JoinPointSetAND or JoinPointSetOR sets. These sets have logical operation between themselves, giving us main joinpoints. These main joinpoints according to logical operation between them are linked to PointcutSetAND or PointcutSetOR. These sets again have logical operation between them and supplies the main pointcut expression.

The relationship between different joinpoints and sets and pointcuts are resolved by an associated modeling interpreter. It parses through the model and gets the relationship with the different joinpoints and pointcuts to form a final pointcut. The automatic generation of aspect or specialization files is done by these modeling interpreters. It parses through the model and gathers all the information to build an aspect file. It internally forms a final pointcut, gets information about the advice and aspect code etc and then generates aspect files.

The generated files have AspectC++ code as shown in Figure 3.4 and Figure 3.5. These files are saved by the AspectFeature model name in the directory pointed. As illustrated in Chapter 3 we proved how we could use these specialization files to optimize a middleware system. By combining both approaches of modeling and AOP we can get better control on the design of an application to be changed. To integrate POSAML and the new capability of modeling AOP we needed a technique to model aspect-oriented constructs in our modeling tool. Using AOP compilers these specializations or aspects are woven into the target application code at the joinpoints.

Chapter 5

Case Study

This Chapter presents a case study illustrating how using AOSD [23] is useful to resolve the tangled concerns for a distributed storage management system for a High performance computing (HPC) application called L-Store [51]. Such applications require huge data storage in the order of tera to peta bytes over a span ranging from a few seconds to weeks or even years for their correct operation. This huge data is stored at geographically distributed sites. It leverages use of enabling technologies such as Logistical Networking (LN) [8] and the Internet Backplane Protocol (IBP) [4]. LN provides new capabilities to schedule data movement and storage on a global scale while IBP provides a middleware for managing remote storage and data objects of varying sizes. They mask the distribution of the storage and instead provide a single file system abstraction to applications.

To elucidate these design challenges better, we first outline our L-Store metadata management system architecture. We then illustrate how different crosscutting concerns make the design of such systems complex.

5.1 Logistical Storage

Logistical storage (L-Store) is a Java based distributed file system providing a virtualization of a single file system to the applications that use it. It is used for storing arbitrary sized data objects. L-Store was created primarily to assist campus researchers who have accumulated large datasets like High Energy Physics (HEP) pile up sample and image process, remote data mining applications, for areas such as weather simulation. It stores metadata information of stored files in a database server for relatively smaller sized metadata but has the ability to leverage the Chord Distributed Hash Table (DHT) architecture for scalability.

L-Store has the ability to transfer huge amounts of data for storing and access between remote labs and between different data centers. It provides real-time data transfer across geographically isolated data stores. L-Store is a conceptually designed complete virtual file system. It uses the Internet Backplane Protocol (IBP) as the underlying abstraction of distributed storage, distributed hash tables (DHT) as a scalable mechanism for managing distributed metadata and software agent technology for implementing a distributed architecture.

In Internet Backplane Protocol (IBP) [4] Exnodes are the pointers to allocations. IBP is a service that allows users to store data in the network. IBP allows allocations up to 4 GB in size. When system developer requests an allocation, a depot (which is an IBP server) returns a capability (or key). It is safer to use these capabilities than ftp or http for file distribution since the allocation key provides a secure access to the files. Unlike ftp and http, the key does not reveal details about the underlying file system. The IBP protocol transfers data between IBP depots by treating the entire data as a big chunk and transferring individual smaller slices. IBP provides fault tolerance and recovery features in a transparent fashion. This protocol is used by L-Store for the storage of files distributed across different storage sites.

In order to manage distributed data and to provide a single file system abstraction, L-Store is required to maintain metadata information for the distributed data. With increasing number of files that store these large distributed data sets, the corresponding amount of metadata also increases. With an explosion in the size of the metadata itself, the problem of metadata management must be resolved for applications like L-Store.

When the amount of metadata is relatively small, L-Store manages it on a single metadata server and was using Postgresql database server to store metadata. This kind of metadata management cannot scale to large sized systems. In order to make data more scalable we are leverage the Chord distributed hash table architecture [50]. During our research we worked on the two versions of L-Store, one was database based metadata server and other was using DHT architecture.

After understanding the design and implementation of these two versions of L-Store it revealed scope for some secondary design concerns for L-Store, such as transaction management, logging, and exception handling which was tangled across the code. Some new concerns like connection pooling and security that needed to be added were found to be crosscutting with respect to the primary design concern of L-Store. The primary sources of these crosscutting concerns stemmed from the need to assure transactional and persistence control, connection pooling, authentication and authorization, and exception handling and logging, which are deemed orthogonal to the primary goals of L-Store.

To improve the maintainability, extensibility, and portability of code, we resolved these sources of code tangling using aspect-oriented programming. To remove the crosscutting nature of these concerns better, we first outline our L-Store metadata management system architecture. We then illustrate how different crosscutting concerns make the design of such systems complex.

Our experience with the design and implementation of the L-Store metadata storage management system revealed a number of sources of crosscutting concerns that affect the maintainability, flexibility, extensibility and in some cases even performance. Below we describe the crosscutting concerns and how they manifest themselves in the L-Store architecture and then we describe how we resolved these design challenges.

5.2 Challenges: Crosscutting Concerns in Logistical Storage

Maintaining persistence in transactions

Maintaining correct transaction control and persistence is vital for database or any system consistency. A transaction is a logical unit of work that may include any number of database updates. During normal behavior, the issue of transaction consistency arises only in a few cases, such as before any transactions have been executed, between the completion of a successful transaction and before the next transaction begins, when the application terminates normally, or the database is closed. However, in the case of failures, without proper rollback mechanisms, transaction processing can result in inconsistent data.

L-Store internally maintains database tables for access control management and other functionalities it provides. There are some tables to store the IBP exnodes, exnode mappings, user to exnode mappings, protected rights of exnodes, among others. L-Store database transactions are executed during application operations, such as an upload of a file, which requires L-Store to update the corresponding metadata information stored across different database tables. For example, database entries that may need to be updated based on an application action include updates to the exnode, `exnode_mappings` and some access control related tables like `protected_objects` and `protected_rights`.

Thus, during this transaction if any exception is raised or an error occurs, and the transaction is aborted, there is a need to rollback the partially executed transaction. If not handled, a user may see inconsistencies such as a file being listed as available but cannot be accessed. This can prove to be a bottleneck for the application if it is not responsible to handle these failures. Handling these database transaction failures is a crosscutting concern since each different operation supported by L-Store will require handling these cases in order to maintain consistency of metadata. Thus, it is necessary to make transactions persistent so that rollbacks or other failure handling can be seamlessly implemented.

Conventional connection pooling methods

As alluded to earlier, L-Store stores some meta data for the metadata management. Connection management is an important parameter that dictates resulting performance. For metadata management, connection management involves a number of steps. First, the connection to the server is established over the network. Next, the user trying to connect is authenticated with the server. Finally, a connection is established and operations are performed. Once all activities are performed, the connection is closed resulting in the connection and server resources being freed.

Owing to all these steps, connection management can be a bottleneck for applications using L-Store, whose main objective is to provide real time access to large quantities of distributed storage virtualized as a single file system. Thus, it is important to optimize connection management in L-Store.

Connection Pooling is a process of obtaining and managing connections faster in an application. Conventional database connection pooling maintains a pool of connections in which a connection is allocated to an application when it requests a new connection and this connection is returned to the pool once the application closes the connection. This type of connection pooling is available only for databases.

For L-Store we moved from database version of metadata management to DHT version, so keeping connection pooling the same for both versions was a challenge. In addition to this there are several conventional database connection pooling drivers like JDBC 2.0 which provide a rich set of features to the applications. They provide a standard way of creating and disposing database connections. They reduce time to obtain new database connections but may cause extra memory and resource constraints.

Moreover, the feature richness can become excessive for many applications since they must use all the functionality provided by these drivers even when they do not need them. Even if there is an option to configure some of these drivers, it is very difficult to configure them and then to test them.

In many application scenarios that use L-Store there is a need to bypass some features so that performance can be improved. In the current set of database drivers, this is not feasible and in most cases these standardized drivers may have to be replaced with proprietary drivers, which is not an acceptable alternative since the cost of developing and maintaining the code base increases.

There may be times during the lifecycle of L-Store that the connection pooling feature may have to be toggled on and off. With conventional pooling it may require changing most of the modules that use pooling [31]. These database connection pooling drivers provide a good database connection pooling solution for the application, but the application becomes tightly coupled to the database driver for resource pooling. The tangling between the resource pooling and database connectivity concern is thus a big challenge needing resolution.

Authentication and authorization feature

Security is important in any software system. It is particularly an important challenge for distributed systems and by nature it tends to crosscut other design issues in any application. It consists of many components like authentication, authorization, auditing, and cryptography. In L-Store there is significant sharing and storing of data across geographical distributed locations. In order to provide secure access and proper protection to the data and resources there should be a security aspect for L-Store. In order to provide security in L-Store based application we focused on the two main components – authentication and authorization.

Authentication is a process that verifies that a user's credentials are valid at the time of login or in subsequent sessions. Authorization determines if the authenticated users have permission to access some system resource. For example user 'A' cannot download a file which has been uploaded by user 'B' unless user 'A' has been permitted to do so.

Using conventional methods of providing security including different API's like OpenSSL, x.509 and JAAS leads to changing multiple modules in the code base of the application. The access control of L-Store was designed based on the entity relation of the various database tables. To add security to the architecture would have forced a change to a large number of modules in our code base. After analyzing our design we found out that the authentication part was straightforward and was not really an orthogonal concern.

L-Store's core functionality was designed in such a way that it was better to use an object-oriented approach to implement this feature. However, after designing authorization we found that it was going to affect all the important modules of L-Store and it was really an orthogonal concern. There were many file related functionalities like upload, download, list, make directory, remove directory and stat among others, which needed verifying of access control.

These challenges stem from the conventional object-oriented design of applications, which are tailored to meet the primary concerns. It is difficult to accommodate secondary concerns such as authorization seamlessly in the same object-oriented design framework and leads to a scattering of decisions [31] *i.e.*, the decision for operations to be checked against permissions is scattered throughout the system, and therefore any modifications to it can cause invasive changes.

Lack of consistency in exception handling

Exception handling and logging are an integral part of almost every application. Making applications *exception safe* is the responsibility of the application developer. Logging may be necessary for accounting or debugging. Often times, however, application developers ignore these secondary concerns and concentrate on the core design challenges of the application. The secondary concerns, such as exception handling and logging, become an afterthought in the design of complex systems [33].

We observed that the design of L-Store suffered from the same weaknesses. Various logging techniques and toolkits can be used for logging. For any logging toolkit, such as log4j [20], developers are still required to write log statements wherever logging is needed. Similar arguments hold for exception handling. Logging and exceptions are interrelated to each other. Logging of exceptions is an important part of the system. Whenever an exception is thrown, applications need to log it so that system failures and problems can be recorded and monitored. This type of logging is also called tracing or monitoring.

Logging and exception handling are fundamentally secondary concerns that crosscut the application code base. Due to code tangling, any changes to the logging or exception handling policy will affect large portions of the code base requiring most often manual changes.

In Section 5.3 we have shown another example of RequestHandler and there system developer can see that exception handling has been done. This type of inconsistency can result in many problems.

5.3 Solution Approach: Use of Aspect-Oriented Techniques

We used Aspect-Oriented programming to provide an elegant solution to address the outlined crosscutting concerns in L-Store. AOP is an advanced programming technique used to separate crosscutting concerns in a modularized fashion. For example, since authorization is to be uniformly implemented in all the units of application, it is better to use aspect oriented techniques so that any changes to authorization are done at one place. In the future if this application may expand or change access control functionality it will be very easy if we make it a separate concern.

In traditional object-oriented programming languages if we add this type of concern on top of existing system core functionality we have to convert these secondary concerns into a class and then use them in primary concerns. These classes would not be reusable and they cannot be inherited and refined properly. They will ultimately be scattered across the program and will be very difficult to manage and work with.

Since access control is a feature which tends to change with the evolution of an application, it is always a good idea to use aspects to design it. We can easily modify and understand security very clearly. AOP provides many powerful techniques to enhance code but sometimes it creates problems because it does not directly affect source code. Reading through code and understanding it becomes difficult but then even comprehension of object-oriented programming is also difficult often times. Also, we have to make sure that the code added or the changes made by AOP to the application should be orthogonal in nature. But, sometimes aspects can be deeply crosscutting, and this happens when the application state, structure and the logic influence the aspect code in such a way that the aspect is only applicable in one specific application context [52].

Applying AOP Technologies to L-Store Design

Section 5.2 described various secondary and crosscutting concerns that make designing complex systems, such as L-Store challenging. In this Section we illustrate how we resolved these challenges using AOP techniques. While addressing these secondary concerns we took care of the changeability and extensibility issues of the code. L-Store is a Java based application; hence, we used the AspectJ [3] to resolve the challenges. In the remainder of this Section we first briefly describe AspectJ and then show how we used AspectJ to resolve the challenges outlined earlier.

AspectJ

AspectJ [26] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of crosscutting concerns that affect several units of a system. This separation of concerns allows better modularity, avoiding tangled code and code spread over several units thereby improving system maintainability. AOP [27] does for crosscutting concerns what OOP has done for object encapsulation and inheritance by providing language extensions and mechanisms that explicitly capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. We have applied AspectJ to resolve the crosscutting concerns in L-Store. We used the AspectJ Development Tool (AJDT) on the Eclipse IDE for our R&D.

Transaction Control and Persistence

L-Store is a Java based distributed file storage application. This application needs to store file information, *i.e.*, Metadata information of the stored files into database server. This metadata server is used by a large number of users and is designed to support millions of transactions. As we described in Section 5.2, because of the lack of persistence there could be loss of updates, inconsistency of data and dirty reads. It is essential for a database transaction to be persistent and all database dependent applications to guarantee the ACID properties [12], *i.e.*, atomicity of operations, data consistency, isolation when performing operations, and data durability even if the system fails.

To address these challenges, there was a need to make some modifications to some part of the original L-Store core code to implement transaction control. Originally in every operation provided by L-Store, there was a call to a database connect and release. The coupling with the primary concern was such that in order to provide transaction control we had to modify some of the methods which ended up establishing and releasing connection to the database. In the code snippet below we show parts of the original L-Store code before the secondary concerns were modularized.

```
1:  public void HandleRequest() throws Exception {
2:    String parent = br.readLine();
3:    String newDir = br.readLine();
4:
5:    try {
6:      Connection dbConn = null;
7:      dbConn = DbUtil.getDBConnection();
8:      DbLstore.insert_directory(dbConn, parent, newDir);
9:      bw.write(LStoreRequests.ALL_OK);
10:     bw.write(LStoreRequests.EOR);
11:     bw.flush();
12:
13:   } catch (SQLException sqle) {
14:       throw new Exception ("Error creating directory: " + sqle);
15:
16:   } finally {
17:       DbUtil.releaseDBConnection(dbConn); }
18: }
```

In the above code snippet we see that for every method there is a separate `getDBConnection()` method call (line 7). This method call is used to create a database connection and here it is used in `insert_directory` method (line 8) and then `releaseDBConnection` (line 17) is called. The modularization of transaction control as an aspect is required for lines 7 through line 17 since otherwise any intermediate failures will result in inconsistencies. To avoid this database inconsistency for every call to the database we introduced an aspect called transaction control is shown in the code snippet below.

```
1: /**
2:   * This aspect is for transaction control
3:   * of database connection
4:   */
5:  public aspect TransactionControl {
6:
7:  /**
8:    * On call of methods that match this pointcut
9:   */
10:   pointcut transactionMethod (Connection conn)
11:     :call(public static * *.*.*.DbLstore.*(..))
12:      && args(conn, ..);
13:
14:
15: /**
16:   * Placeholder for transaction policies
17:   */
18:   Object around(Connection conn):transactionMethod(conn){
19:
20:     Object res = null;
21:     try{
22:       conn = DbUtil.getDBConnection();
23:       res = proceed(conn);
24:       commitTransaction(conn);
25:       DbUtil.releaseDBConnection(conn);
26:     } catch(SQLException qle){
27:       rollbackTransaction(conn);
28:       System.out.println ("Rolled back transaction");
29:
30:     }
31:     return res;
32:   }
33: }
```

In the above code snippet line 5 shows the `TransactionControl` aspect created to handle transaction control of the database. Line 10 is the pointcut named `transactionMethod`. It picks out the set of join points *i.e.*, the well defined points in the program flow where the database connection is required. It will pick all the methods of `DbLstore` library having arguments as database connection. DbLstore is a database library used by L-Store application for database related connections. Whenever these methods of DbLStore library are called, we need a database connection.

Whenever any functionality or method needs storage connection, it is called from the main business logic. Using a proper pointcut definition it is detected by the aspect. And the advice in the aspect provides the necessary connection. For example, this aspect code (line 22) will establish a database connection. This database connection is passed on to the methods of DbLstore using 'proceed(conn)' (line 23) where 'conn' is the database connection. This database connection is used in the method being called and then if everything is fine it will commit the transaction (line 24) and then release the database connection (line 25).

If any kind of failure or any exception is raised it is caught in the same advice and the database transaction is rolled back (line 27). This common algorithm is modularized into an aspect and woven into the code base automatically by the AspectJ weaver. In `TransactionControl` aspect, `rollbackTransactions()` and `commitTransaction()` are the methods that invokes the `java.sql.Connection.commit()` and `java.sql.Connection.rollback()` methods.

Another approach to transaction control can be through implementing three advice instructions *i.e.*, to start, successfully terminate, and abort transactions. The first one is a 'before' advice that starts a transaction just before the execution of any transactional method. It will be similar to initialization of database. The second one uses the "after returning" advice when a method returns with success and in this case we can commit the transaction. The third one uses the "after throwing" advice, which is called when some exception is raised due to some failure. In this step we can rollback the transaction. This type of implementation is given in detail in [46].

Connection Pooling

In order to make connection pooling more optimized and portable we used AspectJ to add a connection pooling feature. As discussed in Section 5.2 there are various constraints in bypassing traditional database connection pooling drivers when not required, and these issues can be overcome by using aspectized connection pooling [31].

In this approach, if the metadata management is done using a pre-existing driver-supported connection pooling mechanism, it will act as the secondary pooling strategy because AspectJ will override the default connection pooling strategy. This type of connection pooling is easy to use. Connection pooling functionality generated by AspectJ is customized according to the needs of an application and can be independent of the storage mechanism used. Using AspectJ we can provide connection pooling for only those modules where the benefits of improved speed outweighs the cost of extra space [31]. This implementation of connection pooling is based on [31]. The advantage of this scheme is that only selected clients will be impacted by the new strategy, which can be driven by modifying a pointcut to select any number of packages and classes in an application. At any time, if the specialized strategy is not needed, an advice can nullify the effect.

Two types of pointcuts are designed in this case:

Connection creation: This pointcut (see code snippet below) is used to capture all the join points where an L-Store primary concern needs a connection from the pool instead of creating a new one.

```
pointcut connectionCreation()
  : call(public static Connection org.lstore.util.DbUtil.getDBConnection());
```

Connection destruction: This pointcut (see code snippet below) is used to capture all the join points where the connection is returned to the pool of connections instead of destroying it.

```
pointcut connectionRelease
  (Connection connection)
  : call(public void org.lstore.util.DbUtil.releaseDBConnection(Connection))&& target(connection);
```

The above two pointcuts will be used in the following manner. First, we create an advice for the connection pooling logic for any connection to use it from a pool instead of creating a new one. This advice is called `connectionCreation` and is shown below.

```
Connection around()
  : connectionCreation() {

  Connection connection = null;
  try{
    connection = connPool.getPoolConnection();
    if (connection == null) {
      connection = proceed();
      connPool.registerPoolConnection(connection);

    }
  }catch(SQLException e){
     //Handle exception
  }
  return connection;
}
```

The next advice is to put the connection back to the pool after using it, as seen in the pointcut `connectionRelease` below. In this connection release aspect we use the “around” advice indicating the condition when the aspect must be applied.

We could have also used “before” advice as well. But in the “around” advice it is easy to add exception handling. If we want to have exception handling using “before” advice we have to add an additional “after” advice. The original body of the method is the same as the body of the advice with special handling for “proceed” in the “around” advice.

Whenever core methods or the transaction control aspect try to release any connection this advice will try to put the connection into the connection pool and on failure, it will use “proceed” to release connection.

```
void around(Connection connection)
  : connectionRelease(connection) {

  if (!connPool.putPoolConnection(connection))
    proceed(connection);
}
```

Authentication and Authorization:

Section 5.2 describes how security is a one of the major crosscutting concern which can be addressed by aspect-oriented techniques very well. For authentication we did not use any AOSD techniques so we do not discuss this issue, however, aspects were required for authorization.

We implemented a very basic preliminary access control. To authorize users and to keep track of what are the users' rights we decided to use the Policy Machine model. A Policy Machine model (PM) [13] is a standardized access control mechanism and requires changes only in its configuration in the enforcement of arbitrary and organization specific attributes-based access control policies. Some of the enforceable policies are combinations of different access control policy instances like Role-Based Access Control (RBAC) [29], Multi-Level Security (MLS) [34] and Identity-Based Access Control (IBAC).

To address the crosscutting challenges with authorization, as a proof of concept, we started with the basic idea of Identification Based Access Control (IBAC) policy in L-Store. In the future we plan to implement the entire PM. IBAC is a very straightforward access control mechanism where the owner of the resource can set access control. Most of the file systems like Unix and NTFS use this type of access control. For authorization, most of the access control was done using proper entity relation between database tables. The database table relationship is designed in such way that it follows IBAC.

In Figure 5.1 we can see that a user can have read or write permissions for files. If the user is the owner of the directory by default it can upload, stat, list or download file. If a user is not the owner of the file it cannot perform all these operations. A user can grant permissions to any other user to access file for either read or write.

For this secondary concern we had to add some code directly into core code. The following are the code snippets showing some part of code to check permissions of the user logged in. This aspect was called every time a user performs some system call like make directory, add user, change permission, grant permissions of object.

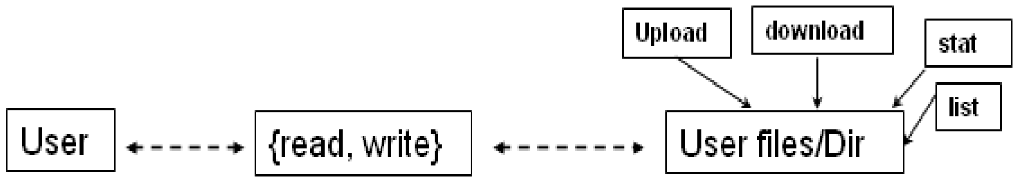


Figure 5.1: Identification Based Access Control configuration

```
1:  Object around(BaseTransaction tran)
2:    :execution(public * org.lstore.core.
3:     BaseTransaction.perform(..)) && this(tran){
4:
5:     try {
6:       if(verifyAccess(tran))
7:         return proceed(tran);
8:
9:     } catch( Exception ex) {
10:      ex.printStackTrace();
11:    }
12:    return LStoreRequests.createBasicReply(false);
13: }
14:
```

In this code snippet we see an advice which is called on execution of the "perform" method of any transaction. For all transactions we verify access rights (line 6) and if the user is authorized, the transaction proceeds with the actual functionality but if the authorization fails, a reply (line 12) to the client is sent indicating unauthorized access. As a side effect of addressing these challenges exception handling is also addressed as in the `verifyAccess` method (line 6) or in proceed (line 7) which is the call for original functionality.

In Figure 5.2 we show how aspects intercept different types of transactions for verifying access of user for different read and write functionalities. In this

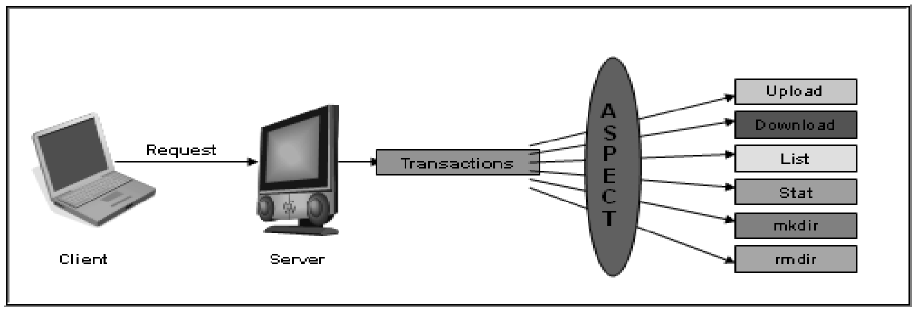


Figure 5.2: Access Control using AOP

Figure we see that there are various modules like upload make directory, list, stat, and remove directory etc. When a user tries to execute any of the functionalities, the user needs to be authorized. Instead of checking permission in each and every module individually, we used AOP to verify access at base transaction level.

Logging and Exception Handling

Logging and Exception Handling are the most common example uses of aspects. They are an inherent crosscutting concern and tend to spread across entire application code. Exception handling was already provided in L-Store but everywhere these exceptions were implemented differently and inconsistently since they were implemented by different people at different stages of development.

In order to generalize all the exceptions we used softened exception handling of AspectJ. For exceptions which are not handled those exceptions are caught by using 'after throwing' advice. We also created some aspects to trace, profile and debug application code.

5.4 Scalable Metadata Management Implementation

In order to make the metadata server more scalable and secure and not to be a single point of failure we move to Chord Distributed Hash table implementation for distributing the metadata on multiple servers.

Chord supports a single operation: given a key, it maps the key onto a node. In our case the “key” is a hash of the directory and file name. A client only needs to know a Chord node on the ring to perform a key lookup. For this type of distributed file storage system we designed and implemented secondary concerns like security, transaction control, and logging. There were many changes in the main business logic code but there were very few changes to the aspects already implemented for the database version of L-Store. More details of the L-Store distributed file system metadata server architecture is given in [51].

Chapter 6

Results and Observations

6.1 Configuration and Specialization Files Generation

A unified framework should provide the mechanisms for the decisions made at configuration time to be available at QoS validation time, and enable the synthesis of validation artifacts. In a MDE framework, such capabilities are realized via generative programming capabilities. Within the GME DSML development environment, in particular, these capabilities are realized by GME model interpreters, which traverse the graphical hierarchy of a model. The POSAML meta-model is a middleware-independent modeling language. By leveraging the GME environment's capabilities, different middleware-specific interpreters can be plugged in. The following describes model interpreters that we have developed as a part of our visual tool:

Interpreter to generate Configuration Files

This interpreter is used to generate two artifacts which are required to configure middleware. One of the generated artifacts is a service configuration file which is used to set QoS related configuration policies by middleware for different applications. It has different options that control the behavior of strategies and resources used by middleware framework. This file allows an application to configure service objects statically and dynamically configuring middleware. For different configuration we have different options. These Options [41] in service configuration file can represent either the components provided by middleware framework or customized components developed by users. If this configuration file is not available, ACE/TAO middleware framework selects all default configurations.

Following are the different factories that can be configured:

1. **Advance Resource Factories:** This factory controls the creation of configurable resources used by ACE/TAO's ORB core. The resource factory is responsible for constructing and providing access to various resources used by the ORB irrespective of whether they perform client or server roles.

2. **Server Strategy factory:** This factory creates various strategies of special utility to the ORB that is useful for controlling the behavior of servers. This factory is responsible for creating strategies useful for server objects like the concurrency strategy and the request demultiplexing strategies used by the POA.

3. **Client Strategy factory:** This factory creates various strategies of special utility to the ORB, useful for controlling the behavior of clients. This factory is responsible for creating strategies useful for clients such as request multiplexing strategies, wait strategies, connect strategies.

The service configuration file will contain strategies as shown below:

```
static Advanced_Resource_Factory "-ORBReactorType tp -ORBReactorThreadQueue LIFO"
static Server_Strategy_Factory  "-ORBConcurrency reactive"
```

Additonally another configuration file generated is a script file. This script file has inputs to run any application like Naming service or a benchmarking evaluation tool with proper end points like listening ports, protocol and host name. These endpoints are also used by Acceptor-Connector pattern for different transport handles.

```
benchmark_test -ORBEndpoint iiop://127.0.0.1:9000
```

The middleware provisioner is shielded from these details since the interpreters automate the task of generating the platform-specific details.

Interpreter to generate Specialization Files

This interpreter is used to generate specialization files which are required to specialize middleware. The code snippet shown in Figure 3.4 and Figure 3.5 is the specialized aspect code. These specialization files are used for the two different specializations discussed in Section 6.2 and Section 6.2. Different types of specializations can be captured as aspects in different files and can be used transform original code into a specialized form according to our requirements.

6.2 Latency and Throughput Results

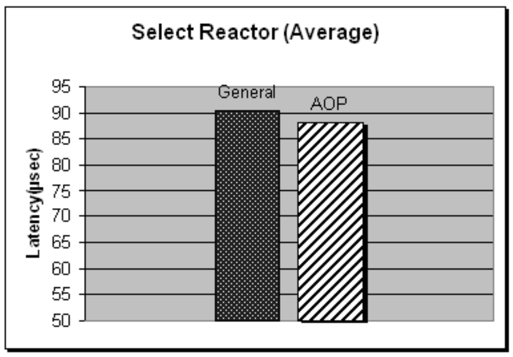
This Section describes results of our experiment comparing the performance of the original ACE reactor pattern with the specialized version. We collected empirical data that compared the specialized version of ACE with the original version along different dimensions including latency and throughput. 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.

For our experiment we used Red Hat Linux 9 with kernel version 2.4.20-8 operating system with CPU speed 2.7GHz, memory of 1GB RAM and 1MB of cache size. For testing our specialized and non-specialized middleware system we used `ACE_ROOT/TAO/performance-tests/Latency/*` testing programs and for all test cases we used Real-Time scheduling class.

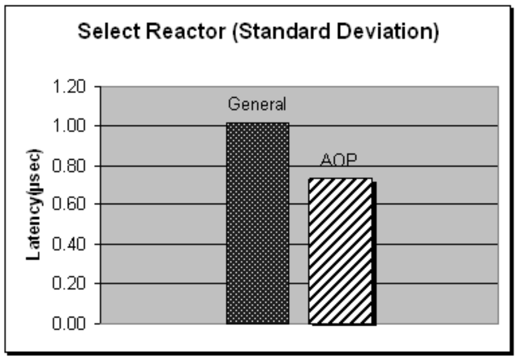
Our experiments illustrate that the refactored middleware framework showed a significant improvement running the ACE performance tests. To demonstrate the benefits of implementing AOP for ACE middleware framework we discuss two specializations of ACE concurrency models in the reactor and illustrate the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

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 the application is using only single threaded reactor. After applying specialization of AOP to the single threaded implementation of reactor, improvements in latency and throughput were observed.



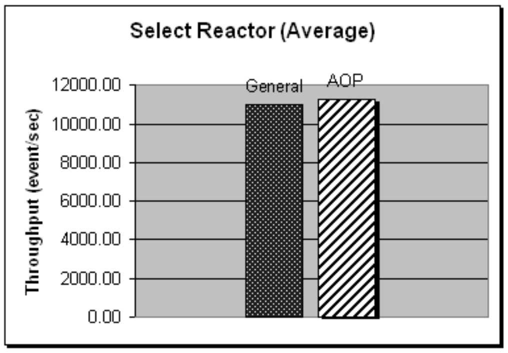
(a) Average



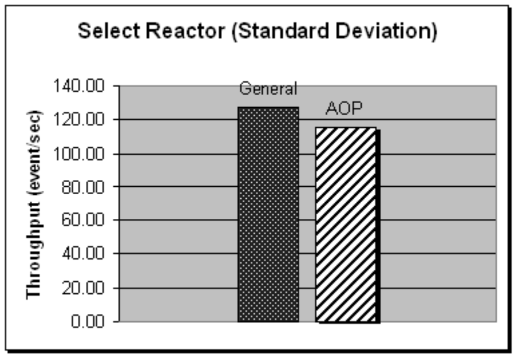
(b) Standard Deviation

Figure 6.1: Select Reactor Latency

Figure 6.1 and shows improved average and standard deviation end-to-end latency. Figures 6.2 shows the increase in the average and standard deviation in throughput after specialization.



(a) Average



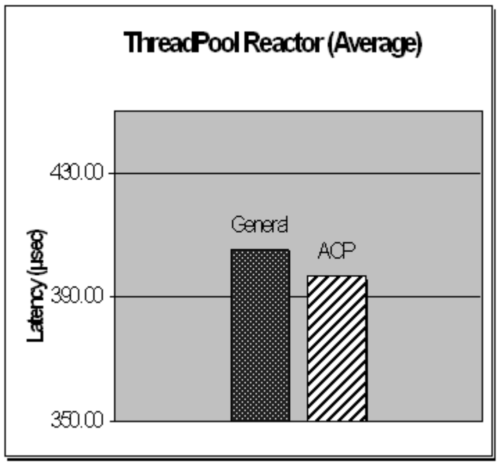
(b) Standard Deviation

Figure 6.2: Select Reactor Throughput

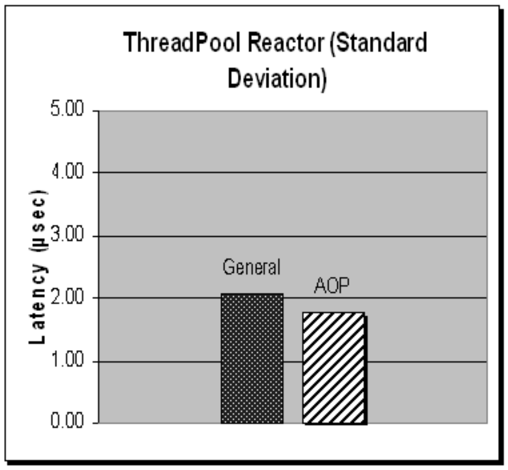
Thread-Pool Reactor

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.

After applying specialization of AOP to the thread-pool implementation of reactor, improvements to latency and throughput were observed. Figure 6.3 shows improved average and standard deviation end-to-end latency.

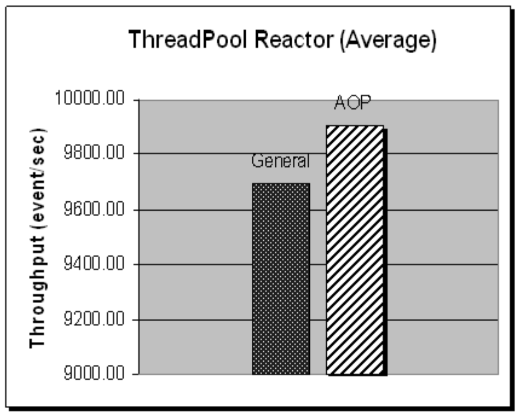


(a) Average

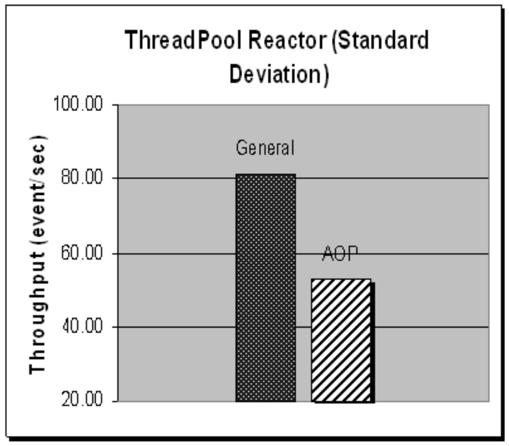


(b) Standard Deviation

Figure 6.3: Threadpool Reactor Latency



(a) Average



(b) Standard Deviation

Figure 6.4: Threadpool Reactor Throughput

Figure 6.4 shows increase in the average and standard deviation in throughput after specialization.

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

Table 6.1: Average Percentage Change

Table 6.1 lists out the percentage decrease of latency and increase in throughput in select and thread pool reactor implementation.

Chapter 7

Related work

One of the tools developed specifically for middleware specialization is Feature-Oriented Customizer (FOCUS) [30]. It is a domain-specific modeling tool that has been developed to automate specialization of middleware. In this specialization tool 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. Then using general middleware compiler, executable code is generated. In this tool join-points need to be identified and the source code changed manually to insert in these join-points (hooks). Correctness of the transformation has to be validated externally. It is expected that the FOCUS approach will be used by middleware developers and not system developers.

Skeletons and Templates are alternatives to achieve separation of concern between the core functionality and secondary concerns. One main difference between skeletons [47] and reusable AOP modules is related to how secondary concerns and core functionality are composed together to yield an application. In the former approach, the core functionality must be decomposed into code fragments to fill the hooks provided by the skeleton/template. In the AOP approaches, this composition is based on joinpoints, which results in less invasive changes to the core functionality.

In paper [19] authors discuss the generation of aspect oriented code (AspectJ [3]) skeletons from a UML model. Their approach offers a mapping between the structure of the model and the structure of the resulting program. The skeletons, however, cannot be executed, as the actual behavior is not modeled. Our work differs from this because we generate fully executable specialization aspect code from the POSAML model. One can make code alteration at modeling level *i.e.*, expressive power of aspects can be defined at modeling level.

There has been some research related to aspect-oriented design model [9, 53] which discuss creation of UML metamodels to integrate aspect-oriented ideas and concepts into design phase of software engineering.

Chapter 8

Summary and Conclusion

Distributed systems implemented with standardized middleware present several challenges with respect to the accidental complexities associated with provisioning (*i.e.*, configuration and QoS validation) and specialization of middleware. In current practice, these challenges are solved through low-level, non intuitive and non reusable means. The manual nature of these techniques is error prone and tedious, and prohibits a system provisioner from rapidly exploring various design alternatives. To address these challenges, our research work presents POSAML, which is a visual modeling language that addresses the provisioning, and approach to express specialization requirement problem at a higher-level of abstraction. It also presents our work related to use of aspect-oriented programming technique to specialize middleware.

We have found that POSAML allows various provisioning scenarios to be explored in a rapid manner that is middleware-independent. The concerns that are separated among the various aspects in POSAML provide an ability to evolve the configuration in a manner that isolates the effect to a single design change. When a choice is made for a pattern, POSAML removes all of the inconsistent choices among other patterns. This allows the provisioner to work with a narrowed search space and ignore all incompatible configurations. Furthermore, model interpreters associated with POSAML assist in generating the artifacts needed to perform QoS validation.

Model-Driven Engineering and Aspect-Oriented Programming approaches are considered to be very useful paradigms. Their approaches are considered to be two complementary solutions which have almost similar goals. There are some areas where they can work together and in some areas they compete.

Our initial results of specializations indicated that the performance of the system improved with increase in number of specialization. If there are very few opportunities for specialization, the use of POSAML and AOP is probably not desirable, however with more opportunities for specialization across multiple layers of the middleware, the automation capabilities are desirable.

Lessons Learned

During our research work we applied POSAML to model several case studies implemented in the ACE/TAO middleware. Although our experience in using POSAML to configure and provision these case studies has been positive, there are still a few limitations that remain. For example, our generative techniques are applied only for the TAO middleware *i.e.*, configuration and QoS validation are specific to this middleware only. The limitations for adding capabilities of modeling aspect to POSAML are that the designer needs to be AOP aware since for applying aspect constructs like advice, pointcuts and joinpoints, the model designer needs to know AOP.

While working on the case study we discussed in Chapter 5 we noticed that there are many concerns like logging and exception handling which are perfect examples of concerns that can be cleanly separated out from the primary concerns, and plugged into the fabric of the application code base. There are however other secondary concerns that cannot be cleanly separated out from the core logic because of the tight integration with the core functionality. For example for security and transaction control we had to modify the system code to some extent.

Some of the limitations of aspect-oriented programming we learned during this work is that it can sometimes increase the complexity in the design of the basic architecture since factoring out some secondary concerns is hard due to the need for minor but invasive changes in existing code base.

The problem is even more prominent when the modularization of secondary concerns and additional development of primary concerns goes on in parallel. In our case we had to deal with a situation where application developers were restructuring the code base as we were modularizing the secondary concerns, which impacted our effort since it affected the conditions when the aspects were to be woven in.

For some developers who do not know about the structure of the code base, it becomes very difficult to fix software defects by just reviewing or inspecting code. One more limitation which is a very well known problem is that a developer cannot add code or functionality at any arbitrary location. There has to be a well defined joinpoint for every change. This limitation sometimes is fixed by making minor modification to the primary concern.

Irrespective of all these limitations AOSD helps in the overall reduction of code tangling and increases the separation of concerns. It makes development time faster and reduces code size. It is always easy to fine grain your secondary concerns when it is decoupled from main business logic. It is easy to plug in and out aspects, this feature helps in making customized applications.

Future Work

The research work illustrated in this work is a first step towards customization, configuration, and QoS validation of middleware systems using modeling tools and automating generation of specialization using model based aspect-oriented software development. This modeling tool is at preliminary stage and does not cover all the desired features. We plan to pursue development and improvement of this tool, in many ways.

There is still need to work on this tool to enhance its features. Currently very few patterns can be modeled using POSAML. We plan to add more patterns that are required to build to form middleware system. Only very limited number of features can be modeled, we need to add more number of features so that middleware system is fully configurable.

An automated code generation for other languages like AspectJ is planned. It should very easy to add different model interpreter or code generator rules to do specified work. But, current interpreter is very strongly coupled with one aspect language *i.e.*, AspectC++, and we plan to make it more generalized. We intend to improve and extend this aspect modeling. For the generation of aspect code, all rules are not covered. Only main important features are designed in detail. We need to develop all the AOP rules for this tool.

Bibliography

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, NY, 1977.

[2] Shahzad Aslam-Mir. Experiences with Real-time embedded CORBA in Telecom. In *OMG's First Workshop on Real-time and Embedded Distributed Object Computing*, Falls Church, VA., July 2000. Object Management Group.

[3] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from `http://eclipse.org/aspectj`, 2006.

[4] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, Martin Swany, Rich Wolski, and Graham Fagg. The internet backplane protocol: A study in resource sharing. *Future Generation Computing Systems*, 19(4):551–561, May 2003.

[5] Don Batory. Multi-Level Models in Model Driven Development, Product-Lines, and Metaprogramming. *IBM Systems Journal*, 45(3), 2006.

[6] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[7] Don Batory, Jacob Neal Sarvela, and Axel Rauschmeyer. Scaling Step-Wise Refinement. In *International Conference on Software Engineering*, pages 187–197, Portland, OR, May 2003.

[8] Micah Beck, Ying Ding, Terry Moore, and James S. Plank. Transnet architecture and logistical networking for distributed storage, September

2004. Available from: `http://loci.cs.utk.edu/publications/2004_Transnet_Architecture.php`.

[9] Christina Chavez and Carlos Lucena. A metamodel for aspect-oriented modeling. In Omar Aldawud, Grady Booch, Siobhán Clarke, Tzilla Elrad, Bill Harrison, Mohamed Kandi, and Alfred Strohmeier, editors, *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, March 2002. Available from: `http://lglwww.epfl.ch/workshops/aosd-uml/Allsubs/aspUML.pdf`.

[10] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 56–65. ACM Press, March 2004.

[11] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[12] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[13] David F. Ferraiolo, Serban Gavrila, Vincent Hu, and D. Richard Kuhn. Composing and combining policies under the policy machine. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 11–20, New York, NY, USA, 2005. ACM Press.

[14] Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke. *Aspect-Oriented Software Development*. Addison-Wesley, Reading, Massachusetts, 2004.

[15] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Harper Collins, 1999.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[17] Holger Giese, Ingolf H. Kruger, and Kendra M. L. Cooper. Workshop on Visual Modeling for Software Intensive Systems. *Procedings of 2005*

IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), page 4, 2005.

[18] Wasif Gilani, Nabeel Hasan Naqvi, and Olaf Spinczyk. On adaptable middleware product lines. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 207–213, New York, NY, USA, 2004. ACM Press.

[19] Iris Groher and Stefan Schulze. Generating aspect code from UML models. In Omar Aldawud, Mohamed Kandé, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhán Clarke, Aida Zakaria Santeon, Peri Tarr, and Faisal Akkawi, editors, *The 4th AOSD Modeling With UML Workshop*, San Francisco, CA, Oct 2003.

[20] Jakarta Log4J Homepage. Web Page. Available from: `http://jakarta.apache.org/log4j/`.

[21] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[22] Dimple Kaul and Aniruddha Gokhale. Middleware Specialization using Aspect Oriented Programming. In *Proceedings of the 44th Annual Southeast Conference*, Melbourne, FL, April 2006. ACM.

[23] Dimple Kaul, Aniruddha Gokhale, Alan Tackett, Larry Dawson, and Kelly McCauley. " applying aspect oriented programming to distributed storage metadata management ". In *Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD'07) at the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, Vancouver, Canada, March 2007. AOSD.

[24] Dimple Kaul, Arundhati Kogekar, Aniruddha Gokhale, Jeff Gray, and Swapna Gokhale. Managing Variability in Middleware Provisioning Using Visual Modeling Languages. In *Proceedings of the Hawaii International Conference on System Sciences HICSS-40 (2007), Visual Interactions in Software Artifacts Minitrack, Software Technology Track*, Big Island, Hawaii, Jan 2007.

[25] Mik Kersten. Aop@work: Aop tools comparison. *part 1. Technical report, University of British Columbia*, 2005. Available from: `www-106.ibm.com/developerworks/java/library/j-aopwork1`.

[26] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[28] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[29] Grzegorz Kolaczek. Specification and verification of constraints in role based access control for enterprise security system. In *International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 190–195, 2003.

[30] Arvind S. Krishna, Aniruddha Gokhale, Douglas C. Schmidt, Venkatesh Prasad Ranganath, John Hatcliff, and Douglas C. Schmidt. Model-driven Middleware Specialization Techniques for Software Product-line Architectures in Distributed Real-time and Embedded Systems. In *Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines*, Half Moon Bay, Jamaica, October 2005.

[31] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*, chapter 13. Manning Publications Co., Greenwich, CT, USA, 2003.

[32] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[33] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings*

of the 22nd International Conference on Software Engineering, pages 418–427. ACM Press, 2000.

[34] M. D. McIlroy and J. A. Reeds. Multilevel security with fewer fetters. In *Proc. Spring 1988 EUUG Conf.*, pages 117–122, London, April 1988. European Unix Users Group. also in *Proc. UNIX Security Workshop,* Usenix Assoc., Portland, August 1988, 24-31.

[35] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall, Englewood Cliffs, NJ, 1997.

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

[37] 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*, pages 107–119, May 1999.

[38] D. C. Schmidt. Acceptor-connector: An object creational pattern for connecting and initializing communication services. In *Pattern Languages of Program Design*, 1995.

[39] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6th *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[40] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[41] Douglas C. Schmidt. Options for tao components. In *The TAO Documentation*, Vanderbilt University. Available from: `http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/Options.html`.

[42] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[43] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

[44] Alan Shalloway and James R. Trott. *Design Patterns Explained.* Software Patterns Series. Addison-Wesley, 2002.

[45] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In Patrick Donohoe, editor, *Software Product Lines: Experience and Research Directions*, volume 576 of *The Springer International Series in Engineering and Computer Science*, New York, NY, USA, Aug 2000. Springer-Verlag.

[46] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj, 2002.

[47] Joao L. Sobral, Miguel P. Monteiro, and Carlos A. Cunha. Aspect-oriented support for modular parallel computing. In Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 37–41, Bonn, Germany, Mar 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01.

[48] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, February 2002.

[49] Olaf Spinczyk and Daniel Lohmann. Aspect-Oriented Programming with C++ and AspectC++. In *Tutorial at Aspect Oriented Software Development (AOSD)*, 2005.

[50] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[51] Alan Tackett, Bobby Brown, Laurence Dawson, Santiago de Ledesma, Dimple Kaul, Kelly McCaulley, and Surya Pathak. Qos issues with the

l-store distributed file system, Oct 2006. Available from: `www.cis.uab.edu/gpce-qos/papers/Alan.pdf`.

[52] B. Vanhaute, B. De Win, and B. De Decker. Building frameworks in aspectj, 2001.

[53] Christina von Flach G. Chavez and Carlos J. P. de Lucena. Design-level support for aspect-oriented software development. In Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, October 2001. Available from: `http://www.cs.ubc.ca/{\tilde}kdvolder/Workshops/OOPSLA2001/submissions/27-chavez.pdf`.