Report Number: WUCSE-2003-78

2003-12-04

# Efficient Customizable Middleware

Ravi Pratap Maddimsetty

The rather large feature set of current Distributed Object Computing (DOC) middleware can be a liability for certain applications which have a need for only a certain subset of these features but have to suffer performance degradation and code bloat due to all the present features. To address this concern, a unique approach to building fully customizable middleware was undertaken in FACET, a CORBA event channel written using AspectJ. FACET consists of a small, essential core that represents the basic structure and functionality of an event channel into which additional features are woven using aspects so that the resulting... **Read complete abstract on page 2.**

# Efficient Customizable Middleware

Ravi Pratap Maddimsetty

Complete Abstract:

The rather large feature set of current Distributed Object Computing (DOC) middleware can be a liability for certain applications which have a need for only a certain subset of these features but have to suffer performance degradation and code bloat due to all the present features. To address this concern, a unique approach to building fully customizable middleware was undertaken in FACET, a CORBA event channel written using AspectJ. FACET consists of a small, essential core that represents the basic structure and functionality of an event channel into which additional features are woven using aspects so that the resulting event channel supports all of the features needed by a given embedded application. However, the use of CORBA as the underlying transport mechanism may make FACET unsuitable for use in small-scale embedded systems because of the considerable footprint of many ORBs. In this thesis, we describe how the use of CORBA in the event channel can be made an optional feature in building highly efficient middle-ware. We look at the challenges that arise in abstracting the method invocation layer, document design patterns discovered and present quantitative footprint, throughput performance data and analysis. We also examine the problem of integrating FACET, written in Java, into the Boeing Open Experimental Platform (OEP), written in C++, in order to serve as a replacement for the TAO Real-Time Event Channel (RTEC). We evaluate the available alternatives in building such an implementation for efficiency, describe our use of a native-code compiler for Java, gcj, and present data on the efficacy of this approach. Finally, we take preliminary look into the problem of efficiently testing middleware with a large number of highly granular features. Since the number of possible combinations grow exponentially, building and testing all possible combinations quickly becomes impractical. To address this, we examine the conditions under which features are non-interfering. Non-interfering features will only need to be tested in isolation removing the need to test features in combination thus reducing the intractability of the problem.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-78

# Efficient Customizable Middleware - Master's Thesis, December 2003

Authors: 'Maddimsetty, Ravi Pratap'

December 4, 2003

Abstract: The rather large feature set of current DOC middleware can be a liability for certain applications which have a need for only a certain subset of these features but have to suffer performance degradation and code bloat due to all the present features. To address this concern, a unique approach to building fully customizable middleware was undertaken in FACET, a CORBA event channel written using AspectJ. FACET consists of a small, essential core that represents the basic structure and functionality of an event channel into which additional features are woven using aspects so that the resulting event channel supports all of the features needed by a given embedded application.

However, the use of CORBA as the underlying transport mechanism may make FACET unsuitable for use in small-scale embedded systems because of the considerable footprint of many ORBs. In this thesis, we describe how the use of CORBA in the event channel can be made an

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

# EFFICIENT CUSTOMIZABLE MIDDLEWARE

by

Ravi Pratap Maddimsetty B. Tech.

Prepared under the direction of Dr. Ron K. Cytron

---

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

December, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

---

ABSTRACT

---

EFFICIENT CUSTOMIZABLE MIDDLEWARE

by Ravi Pratap Maddimsetty

---

ADVISOR: Dr. Ron K. Cytron

---

December, 2003
Saint Louis, Missouri

---

The rather large feature set of current Distributed Object Computing (DOC) middleware can be a liability for certain applications which have a need for only a certain subset of these features but have to suffer performance degradation and code bloat due to all the present features. To address this concern, a unique approach to building fully customizable middleware was undertaken in FACET, a CORBA *event channel* written using AspectJ. FACET consists of a small, essential core that represents the basic structure and functionality of an event channel into which additional features are woven using *aspects* so that the resulting event channel supports all of the features needed by a given embedded application.

However, the use of CORBA as the underlying transport mechanism may make FACET unsuitable for use in small-scale embedded systems because of the considerable footprint of many ORBs. In this thesis, we describe how the use of CORBA in

the event channel can be made an optional feature in building highly efficient middleware. We look at the challenges that arise in abstracting the method invocation layer, document design patterns discovered and present quantitative footprint, throughput performance data and analysis.

We also examine the problem of integrating FACET, written in Java, into the Boeing Open Experimental Platform (OEP), written in C++, in order to serve as a replacement for the TAO Real-Time Event Channel (RTEC). We evaluate the available alternatives in building such an implementation for efficiency, describe our use of a native-code compiler for Java, gcj, and present data on the efficacy of this approach.

Finally, we take preliminary look into the problem of efficiently testing middleware with a large number of highly granular features. Since the number of possible combinations grow exponentially, building and testing all possible combinations quickly becomes impractical. To address this, we examine the conditions under which features are *non-interfering*. Non-interfering features will only need to be tested in isolation removing the need to test features in combination thus reducing the intractability of the problem.

To Amma and Nanna, for showing me the way.
And to every person along the way who believed in me.

# Contents

# List of Figures

# Acknowledgments

The completion of a Master's degree is usually a long and arduous journey through many years of learning, hard work and fun. This thesis is the culmination of a similar journey that began in the 6th grade with the desire to delve into the wonderful world of Computer Science, and that wound through the study of Chemical Engineering before finally ending up at the Department of Computer Science and Engineering at Washington University. As with any achievement in one's life, this too would not be possible without the support of numerous unsung people along the way.

I thank my parents and brother for all the support they have ever given me and for teaching me the joys of life - from music to academics. I would also like to thank all my teachers at the Hyderabad Public School, my professors at the Indian Institute of Technology, Madras, and the faculty at Washington University for everything that I have learnt from them and for providing me with the encouragement that is crucial to every student's growth. I particularly thank my advisor, Ron K. Cytron, for taking me under his wing and teaching me so much about research and giving me the opportunity to work on challenging problems. I also thank my colleagues and friends in the DOC Group - Balachandran Natarajan, Morgan Deters, Krishnakumar Balasubramanian, Sharath Cholleti, Frank Hunleth, and many others - for the many enjoyable experiences. In particular, I thank Pavan Mandalkar, Rooparani Pundaleeka and Angelo Corsaro for all the entertaining moments through gruelling sessions of coding and bug-fixing. I also thank DARPA for supporting my research under contract F33615–00–C–1697.

The last two years at Washington University have also been memorable outside of the office. I thank Vignesh Nandakumar for being the patient friend that he is, and for all the stimulating discussions we have had. And finally, I thank Vidya Venkataramani for being my constant companion and for teaching me to rediscover the joys of Indian classical music.

Ravi Pratap Maddimsetty

*Washington University in Saint Louis*
*December 2003*

# Chapter 1

# Introduction

As Distributed Object Computing (DOC) middleware finds application in increasingly diverse areas, successful middleware such as the Common Object Request Broker Architecture (CORBA) [27], Microsoft Component Object Model (COM) [25], and Java [1] Remote Method Invocation (RMI) [39] have grown quickly to include a vast number of features in response to the needs of all users. However, most applications tend to use only a subset of these features yet their footprint and performance can be affected due to the number of features present. Currently, to address these concerns, middleware developers often refactor code to relegate functionality into separate libraries. This process is tedious, time-consuming, and adds complexity for both users and developers, especially for large frameworks such as the ADAPTIVE Communication Environment (ACE) [33]. A compelling need therefore exists for middleware with support for full customization of feature combinations to suit the needs of each target application.

A unique approach to building fully customizable middleware was undertaken in the Framework for Aspect Composition for an EvenT channel (FACET) [20], a CORBA event channel, writting using AspectJ [36]. An *event channel* is a well-established, standard service for decoupling the supplier and consumer of events in a distributed system [29, 37]. FACET consists of a small, essential core that represents the basic structure and functionality of an event channel into which additional features are woven in using aspects. The resulting event channel thus supports all of the features needed by a given embedded application. Chapter 2 provides a more detailed description of FACET's architecture and explains how some of the problems in existing subsetting techniques are solved through the use of a feature framework.

[1] Java is a trademark of Sun Microsystems, Inc.

By tightly controlling exactly what features are included [21, 22], the footprint of the resulting event channel can be half of that required for a full-featured event channel when measurements exclude the size of the supporting Object Request Broker (ORB). However, what is of importance to an embedded application is the combined footprint of the event channel and the ORB. The footprint of a high-quality Event Service implementation such as the TAO Event Service (TAO is the ACE ORB (TAO) [6]) can, in certain configurations, be quite high mostly due to the size of the ORB [20]. Clearly, TAO's footprint is a key concern for small-footprint event channels which need to be deployed in embedded systems with tight constraints and limited resources. While efforts are underway to create reduced-feature, small-footprint ORBs [12], there are compelling applications that do not even need distribution and/or inter-language support. Chapter 3 describes how we made use of CORBA an optional feature in FACET. We describe the challenges in abstracting a systemic concern such as the method invocation (or transport, as we refer to it in this thesis) layer, study design patterns used and present quantitative footprint and performance data detailing the impact of CORBA.

An interesting problem explored in this thesis is the problem of integrating FACET with the Boeing OEP, a C++ avionics software development framework. One of the key components of the OEP is TAO's Real-Time Event Channel (RTEC) [17], an Event Service implementation used to decouple various supplier and consumer components. As part of the integration, we aim to replace the RTEC, written in C++, with FACET, written in Java. In doing do, we want to preserve the external interfaces of the RTEC and ensure that all clients are completely unaware of the change. In Chapter 4 we describe our use of a native-code compiler for Java, gcj [13], and the Cygnus Native Interface (CNI) to allow seamless interaction between Java and C++ code. In addition, we present the data on the efficacy of this approach and provide a comparison with the other standard available for native code interaction - Java Native Interface (JNI).

One of the fundamental design principles of FACET is to split functionality into highly granular features which can be composed as necessary, with the appropriate dependence relationships between them satisfied in any composition. However, this proliferation of features poses a problem in the area of testing because the FACET build framework tests all possible valid combinations of features. Since the number of possible combinations grow exponentially with the number of features, building and testing all such combinations quickly becomes impractical. In Appendix B, we

examine the problem of determining the conditions under which aspects (and consequently, the features) do not *interfere* with each other. Non-interfering features would only need to be tested in isolation instead of in all possible combinations that include the said features thus reducing significantly, the time complexity of generating all possible combinations and testing them.

Finally, Chapter 5 summarizes our work, looks at possible applications of the approach we have developed in this thesis, and talks about some of the things we are planning to do in the future.

# Chapter 2

# Background

This chapter presents some essential details of the architecture of the FACET event channel and provides some necessary background on Aspect-Oriented Programming (AOP) and AspectJ as well as Java and gcj [13]. An overview of the CORBA ORB reference model is presented to put our discussion of the transport layer in context.

## 2.1 DOC Middleware

Developing large software projects has known to be a pretty difficult task [3]. Programming platforms vary widely, outdated and unwieldy programming interfaces abound, and frameworks for addressing communication issues either may not be available or may not be interoperable. It is for these types of problems where middleware has proven to be very useful in practice [34].

DOC middleware is a specific category of middleware that addresses the many accidental and inherent complexities [4] of network and distributed programming. *Accidental* complexity refers to the programming issues with using tools, languages, interfaces, and frameworks that are difficult to use and prone to errors. Network programming has historically been difficult due to the lack of availability of anything besides low level socket interfaces. On the other hand, *inherent* complexities arise out of inherent difficulties with developing any program in the domain regardless of language, tools, or libraries. For networking, these include issues such as fault tolerance, security, concurrency, and program distribution.

ACE [33] and TAO [6] are two of many examples of DOC middleware frameworks that address the difficulties of distributed network programming. Both of these

frameworks have matured over many years of use for both research and industry applications [15] [9]. Issues identified during their development and evolution, though, led to the development of FACET.

FACET is an implementation of a CORBA [27] *event channel* that uses AOP to achieve a high level of customizability. Its functionality is based on features found in the Object Management Group (OMG) Event Service [29], the OMG Notification Service [26, 14], and the TAO Real-time Event Service [30, 16].

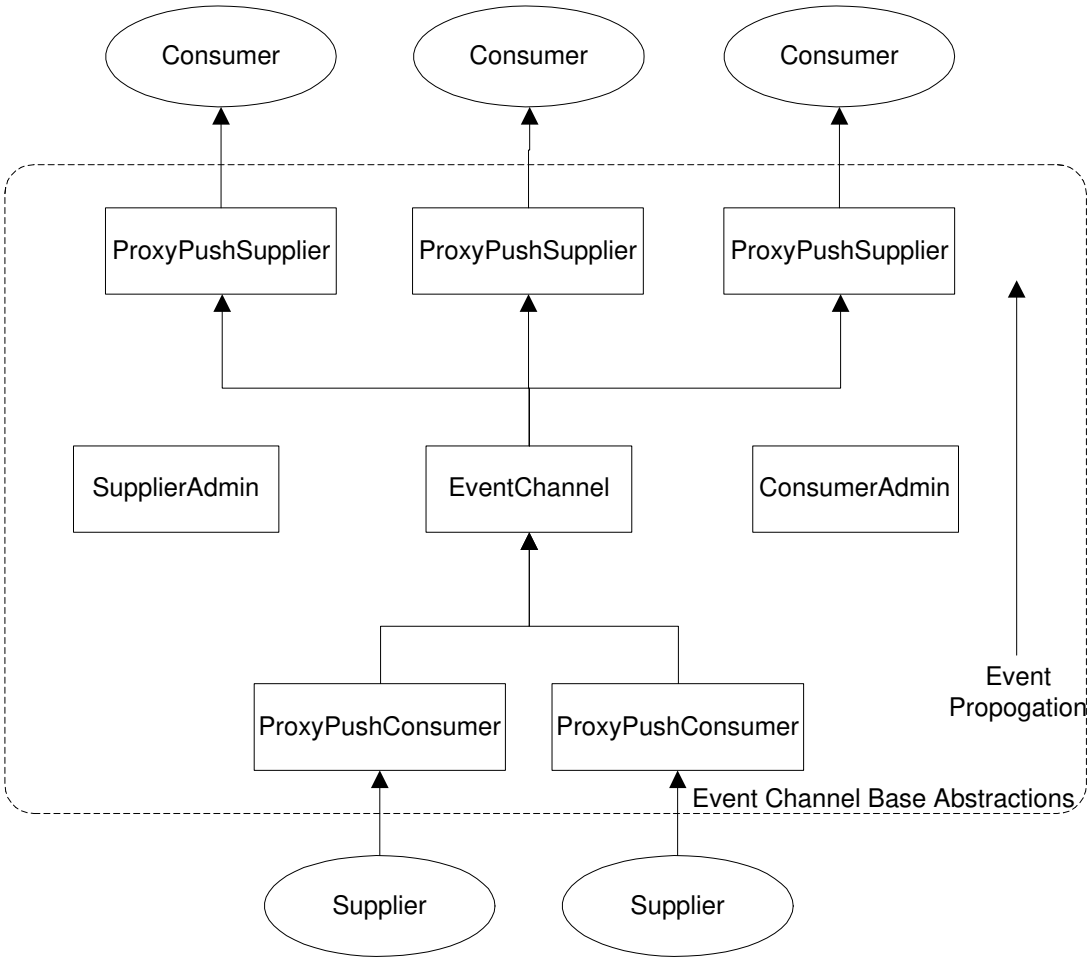


Figure 2.1: Main participants in an event channel.

An event channel is a common middleware framework that decouples event suppliers and consumers. The event channel acts as a mediator through which all events are transported. Figure 2.1 shows the main participants in an event-channel framework. At its simplest,

1. Suppliers push events to the event channel
2. The event channel applies any filtering, correlation or other specified features to the events
3. The event channel pushes appropriate events to consumers.

Event channel implementations differ in the types of events that they handle and in the processing and forwarding that occurs within the channel in addition to the size of their footprint and the throughput performance (in events/sec) that they are capable of.

## 2.2 FACET Architecture

Figure 2.2 depicts the five major components that are fundemental to the FACET middleware. Each of these components interacts in some way with each of the other components, and without such interaction, some major functionality would be lost.

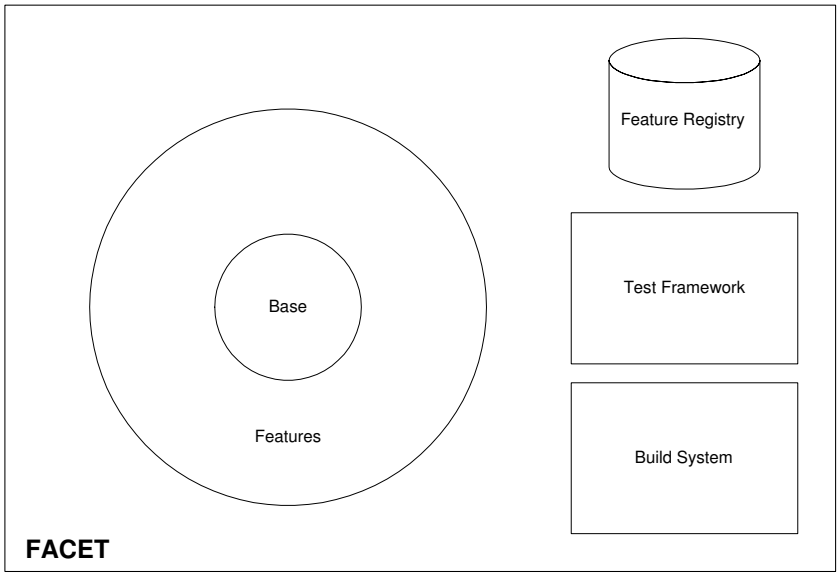


Figure 2.2: The main components in FACET.

The implementation of the event channel is first separated into a base and a set of user-selectable *features*. The base represents an essential level of functionality. Each feature adds a structural and/or functional enhancement to the base or to

other features, and AOP language constructs integrate or *weave* feature code into the appropriate places in the base as well as the features. The construction of FACET follows a bottom-up approach in which features are implemented as needed. As new requirements are presented, they are decomposed into one or more features. In the case of FACET, the features of several existing event services were selected one-by-one for incorporation.

The base consists of a simple implementation of interfaces similar to those found in the CORBA Event Service with a few minor differences. To support functionality not found in the base implementation, FACET provides a set of features that can be enabled and combined, subject to some dependence constraints. These features include:

- Interfaces and implementation to support *pulling* events through the event channel.
- Various event-payload types such as CORBA Anys, CORBA octet sequences and strings.
- Event structures such as headers that are made visible to the event channel and used by other features. These include event type-labels for dispatch and filtering, a time to live (TTL) field to support federated event channels, and timestamp fields for profiling.
- Dispatch strategies that trade off channel performance and memory usage.
- Event-correlation support that allows consumers to specify logical operations (AND, OR etc.) on sequences of events that should be received by a channel before notification.
- Plugging the use of CORBA in and out

In addition to the base and features, Figure 2.2 illustrates three other major components in FACET. The Feature Registry maintains all of the relationships and metadata concerning every feature. It has the responsibility for validating event-channel configurations and providing dependence relation information to the other components. The Build System is then responsible for selecting and compiling the appropriate source files that correspond to the desired feature configuration. The Test Framework has the responsibility of verifying that each feature and its compositions

perform actually as intended. It is used to gain a high level of confidence that changes to the base or to other features do not have unintended consequences in any configuration.

## 2.3 Separation of Concerns and AOP

AOP and the ability to encapsulate cross-cutting concerns into units is central to the approach of separating features in FACET. In this section, we present some background on the subject separation of concerns and AOP.

### 2.3.1 Separation of Concerns

Separation of concerns [8] is the general term given to the process of identifying and encapsulating related ideas and concepts together. Separation of concerns for Object-Oriented Programming (OOP) involves identifying the structure of classes and interfaces that define an application. However, separating concerns based on structural elements is only one of many dimensions where separation can occur. The inability of OOP to separate other concerns such as synchronization and memory management has led to significant research in identifying new approaches such as AOP. These approaches are collectively termed Advanced Separation of Concerns (ASoC) due to their ability to enable more flexible separations [20].

Before describing the languages and paradigms used to encapsulate nonstructural concerns, it is useful to describe other types (or dimensions) of concerns. These can be broadly categorized as *systemic* and *functional* concerns [31].

- Systemic concerns include synchronization, realtime, scheduling, transaction semantics, caching and prefetching strategies and memory management concerns.
- Functional concerns comprise application logic and features. These differ from systemic concerns in their scope and intention. For example, a application logic such as a new business rule may effect several computations and decisions in separate classes, but a systemic concern such as synchronization affects many classes systemwide.

Both of these types of concerns crosscut many classes, and by encapsulating them into separately compilable units, one can selectively enable or disable their behavior.

### 2.3.2 AOP and AspectJ

AOP [23] is a software development paradigm that enables one to separate concerns that crosscut sets of classes (or other abstractions from some other dominant decomposition) and encapsulate those concerns in self-contained modules called *aspects*. The AspectJ [36] programming language adds AOP constructs to Java [2] and uses the following terminology. Within an aspect, the locations at which *advice* should be applied are defined using *pointcuts*. Each pointcut is made up of one or more *joinpoints*, which are well-defined locations in the execution of a program. The code applied at a pointcut is called *advice*. In addition to applying advice, languages supporting AOP often allow new methods or other language features to be *introduced* into existing classes. Of all the advanced separation of concerns languages suitable for developing middleware, AspectJ is currently the most mature and was thus selected as the language of choice for the implementation.

Subsetting experience tells us that reducing the coupling between classes in a library can reduce the footprint of applications that use selected parts of that library. AOP provides a novel mechanism to reduce footprint size even further by enabling crosscutting concerns between modules to be encapsulated into user-selectable aspects. The advantage of using AOP is that the hooks and callbacks required for subsetting (using standard, object-oriented techniques) are no longer required. This delays the need to conceive where points of variation are needed in the code and also reduces the need to refactor large amounts of existing code to insert these hooks after the fact.

Desirable combinations of these aspects are then selected by middleware users so to include the minimum functionality needed to support a given application. By performing a fine-grain decomposition of the functionality, a middleware framework could add very little bloat to an application, and thereby free the embedded developer from concerns about excessive overhead. Unfortunately, fine-grain decompositions significantly increases the complexity of testing the software extensively. We frame some of the issue in Appendix B.

## 2.4 Overview of the CORBA Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS

platform, communication protocols and interconnects, and hardware [18]. Figure 2.3 illustrates the key components in the CORBA reference model [28] that collaborate to provide this degree of portability, interoperability, and transparency. [1]

CORBA ORBs [27] allow clients to invoke operations on distributed objects without concern for the following issues:

- Object location: CORBA objects either can be collocated with the client or distributed on a remote server, without affecting their implementation or use.
- Programming language: The languages supported by CORBA include C, C++, Java, COBOL, and Smalltalk, among others.
- OS platform: CORBA runs on many OS platforms, including Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.
- Communication protocols and interconnects: The communication protocols and interconnects that CORBA run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.
- Hardware: CORBA shields applications from side effects stemming from differences in hardware, such as storage layout and data type sizes/ranges.

Figure 2.3 illustrates the components in the CORBA 2.x reference model, all of which collaborate to provide the portability, interoperability and transparency outlined above.

Each component in the CORBA reference model is outlined below:

- *Client:* A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. A client has no knowledge of the implementation of the object but does know its logical structure according to its interface. It also doesn't know of the object's location - objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, Figure 2.3 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

[1] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [27].

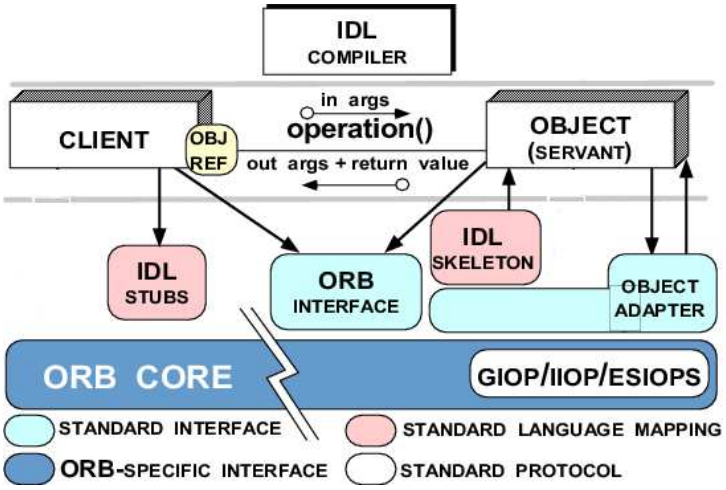


Figure 2.3: Components in the CORBA 2.x Reference Model

- *Object:* In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

- *Servant:* This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

- *ORB Core:* When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol.

- *ORB Interface:* An ORB is an abstraction that can be implemented various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an interface to an ORB. This ORB interface provides standard operations to initialize and shut down the ORB, convert object references to strings and back, and so on.

- *IDL Stubs and Skeletons:* IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [11] and marshal application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [11] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

- *IDL Compiler:* An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations.

- *Object Adapter:* An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Even though different types of Object Adapters may be used by an ORB, the only Object Adapter defined in the CORBA 2.x specification is the Portable Object Adapter (POA).

## 2.5 Java and gcj

The Java platform consists of a Java Virtual Machine (JVM) and a set of standard class libraries. Applications written in Java can be compiled into `.class` files containing bytecodes, a machine-independent, relatively compact binary format. Bytecodes can then be executed by a JVM on any platform. Such execution could take the form of interpretation or compilation with direct execution or a combination of each.

Java has traditionally been either interpreted or compiled using a Just-In-Time (JIT) compiler such as Sun's HotSpot Virtual Machine. The appearance of gcj (part of the GNU family of compilers and essentially a front-end to the popular

gcc), a portable, optimizing, ahead-of-time compiler for Java, allows one to compile applications to the native code for a particular platform. gcj generates executables and libraries that can be run directly by the operating system with no intervention from a virtual machine.

The advantages of such an approach can be numerous. In situations where performance is critical, compiling to native code can speed code up by an order of magnitude. In addition, it is possible to apply numerous optimizations in the generation of code for the target platform. Since one of the complaints against Java is that applications tend to be at least 3 to 10 times slower than corresponding C++ applications, this allows Java code to be run at speeds comparable to C and C++ code for a target architecture.

Since Java applications typically make use of other third-party libraries as well, it is also essential that all the supporting libraries be compiled to native code. We compile these libraries to shared library objects which can then be linked into the main application. Essentially, the approach is similar to that for C++ applications and libraries.

FACET supports compilation to native code using gcj as part of the build process. When enabled, the build process proceeds in the following manner:

1. Supporting libraries such as jUnit are compiled to standalone, shared libraries *e.g.*, libjunit.so.

2. The code from various features selected in the configuration, along with the base, is compiled into a separate shared library, libfacet.so.

3. Executables for unit tests are linked against the above libraries.

The JVM is capable of loading classes on demand as they are referenced in an executing application. One of the downsides to compiling Java to native code, however, is the lack of an ability to load classes dynamically. The reason for this comes from the way shared libraries work on most platforms — the first reference to a symbol in the library causes the entire library to be loaded into memory. Although libgcj, the gcj runtime, can dynamically load and interpret class files, resulting in mixed compiled/interpreted applications, it cannot do so with Java classes compiled to shared libraries.

Another key feature of Java is the garbage-collected heap. Garbage collection in the JVM is done by a separate thread which reclaims objects which are no longer

referenced. Since Java compiled to native code is not run by a JVM, libgcj features a conservative garbage collector — the Boehm garbage collector — to support this important language feature.

gcj also features the CNI standard for tight interoperability between C++ and Java. The C++ and Java compilers at issue (gcc and gcj) use the same calling conventions, object layout, and name mangling, thus promoting interoperability through the libgcj runtime.

# Chapter 3

# Transport Layer Abstraction

In previous work, AOP has been applied to build software using the compositional approach by building a core of basic functionality and then codifying all additional features into separate aspects [21, 22, 20]. Since the transport layer (as we refer to it in this thesis, another name for which would be *method invocation layer*) used in the event delivery mechanism (in this case, CORBA) is a cross-cutting concern for the set of classes implementing the functionality of the event channel, it would be possible to abstract this into a separate aspect such that the inclusion or exclusion of the same produces an event channel with the desired functionality.

In this chapter, we describe our AOP approach for CORBA abstraction in FACET. We focus only on the particulars of our implementation that are relevant to the transport layer and its abstraction and omit details concerning the Feature Registry and the Testing Framework. In addition to CORBA, we also take a brief look at how the techniques developed can be applied in the case Java RMI is used as the transport layer.

## 3.1 Motivation

One of the special challenges associated with embedded systems is supporting their great diversity. Even within the very closely related set of avionics systems associated with the Boeing Bold Stroke product line software initiative, systems may have anywhere from one to ten processors, may run on Versa Module Europe (VME) and/or fiber channel based interconnects, may have one or more languages, and may run on a range of different operating systems. When these characteristics are taken together, the simplest deployed systems are single processor applications written completely

in C++ without any interprocess communication, and the most complex ones are multiple VME backplanes connected by fiber channel, each with multiple processors, also written entirely in C++. Some of these systems are small enough that the footprint imposed by the ORB and the performance penalty as a consequence of the increased communication and dispatching overhead is not acceptable. Clearly, the need to support such diversity is important for customizable middleware.

FACET was originally designed to use CORBA so that events can be sent to and received from remote consumers and suppliers. The advantages of CORBA include the ability to distribute the consumers and suppliers as well as to fashion their implementation for any language that maps to CORBA (*e.g.,* Java and C++). The language independence is obtained by specifying interface definitions via CORBA's IDL.

However, in certain usage scenarios where distribution and multi-language support is unnecessary, the use of CORBA becomes unnecessary. In this situation, the underlying transport mechanism can be a simple method call, doing away with the need to make use of an ORB. Indeed, one form of this optimization is routinely used in the Bold Stroke event service, in the form of the Subscription and Filtering configuration [16]

Following the compositional approach, we sought to provide a standard interface for the event channel while making the use of CORBA optional as well. In other words, merely by selecting an EnableCorba or DisableCorba feature (which are mutually exclusive since it would make no sense to enable both) at build-time, it should be possible to obtain an event channel with the desired configuration.

In what follows, we describe the challenges in abstracting the use of CORBA in the event channel and how these were addressed by the use of AOP [24].

## 3.2 Implementation

Since FACET was originally designed to use CORBA, its interfaces were specified in IDL and the implementation code was written in terms of CORBA Stub and Skeleton classes [27]. The challenge lay in separating the concerns related to CORBA from the actual event channel implementation code for implementing the various features offered by an event service (present in various other classes). In the following, we examine some of the challenges in and describe the techniques we adopted.

### 3.2.1 Challenges in Abstracting CORBA

**Changing Inheritance Hierarchy** Consider the ProxyPushConsumer interface [29] of the event channel, when configured to push structured event data. In the case CORBA is in use, the IDL describing this interface is shown in Figure 3.1

```
interface PushConsumer {
    void push (in Event data);
    void disconnect_push_consumer ();
};

interface ProxyPushConsumer : PushConsumer {
    void connect_push_supplier (in PushSupplier supplier);
};
```

Figure 3.1: ProxyPushConsumer IDL interface

The IDL compiler when given the above would generate the necessary Stub and Skeleton classes [27]. To implement the ProxyPushConsumer interface, the event channel's implementation would include a ProxyPushConsumerImpl class with a definition as described in Figure 3.2

```
public class ProxyPushConsumerImpl
       extends ProxyPushConsumerPOA {

       // Appropriate implementation

}
```

Figure 3.2: ProxyPushConsumerImpl in the CORBA case

However, in the case CORBA is not needed, the interfaces can directly be specified in Java as shown in Figure 3.3. The corresponding implementation of the ProxyPushConsumer interface is shown in Figure 3.4. This idea recurs for all interfaces exposed by the event channel and is a concern which is independent of the manner of implementation.

**Narrowing References** The other issue that we need to address is the method by which object references are narrowed to the corresponding interface references.

With CORBA enabled, it is necessary to invoke helper methods to obtain a reference from a Servant [27] object. For example, to obtain the ProxyPushConsumer

```
public interface PushConsumer {
    public void push (Event data);
    public void disconnect_push_consumer ();
}

public interface ProxyPushConsumer
       extends PushConsumer {

    public void connect_push_supplier (PushSupplier supplier);
}
```

Figure 3.3: ProxyPushConsumer in the non-CORBA case

```
public class ProxyPushConsumerImpl
       implements ProxyPushConsumer {

       // Appropriate implementation

}
```

Figure 3.4: ProxyPushConsumerImpl in the non-CORBA case

interface reference from the servant ProxyPushConsumerImpl object, one needs to make use of the `narrow` method on the ProxyPushConsumerHelper class. However, in the non-CORBA case, to obtain the interface type reference, it is sufficient to directly cast the object reference to the interface type since there are no Servant objects and the implementation class actually implements the PushConsumer interface directly. The two methods are shown in Figure 3.5

```
if (...) {
   // CORBA case
   ProxyPushConsumer ppc =
     ProxyPushConsumerHelper.narrow (poa.servant_to_reference (impl));
} else {
   // non-CORBA case
   ProxyPushConsumer ppc = (ProxyPushConsumer) impl;
}
```

Figure 3.5: Narrowing references

This problem is similar to that of strategizing method implementations based on the context. The difference in this case is that the context is decided by which aspect (EnableCorba or DisableCorba) is applied to the event channel.

**IDL Instrumentation** Since FACET's features change the public interfaces of the Event Channel, a problem that we face is that of instrumenting the IDL to match FACET's configuration. In previous work [20], this was done using primitive search-and-replace scripts but such an approach decreases the maintainability of the code. A challenge therefore is that of instrumenting the IDL through an automated process which ensures that changes need to be made in only one place in the event channel.

In what follows, we document two design patterns that we discovered in the process of solving the problems mentioned above : the Placeholder Class and Placeholder Method patterns. We also describe our approach to generating IDL using reflection.

### 3.2.2 Placeholder Class Pattern

In certain applications, a class providing some functionality does so in a manner independent of its parent class hierarchy. However, based on the configuration, it might need to implement an interface or extend a class while continuing to expose the same public methods. This pattern provides an efficient method to achieve such dynamic polymorphism using aspects.

#### Context

An environment where inheritance hierarchy for a class changes based on the configuration of the application, while functionality provided by the class remains the same.

#### Problem

A class might sometimes provide the same functionality in two different contexts which differ only in the type that is expected by the users of that class. For instance, in one context, the class might be expected to be of a class type $T$ while in some other context, it might be expected to be of some interface type $T'$. The thing to note is that the two contexts are mutually exclusive and arise out of differences in the way an application is configured.

Clearly, we want to allow both users of the class to be able to access the same implementation using the appropriate type reference without having to resort to using

the Adapter pattern since it would require us to maintain a separate adapter for each type $T$. Since the number of such adapter classes grows with the number of contexts, it quickly becomes impractical to maintain all such adapter classes.

## Solution

Use a single, empty class (called the Placeholder) as the base class (or interface) for the class providing the functionality and use aspects to modify the inheritance hierarchy of the Placeholder class by using the 'declare parents' construct in AspectJ to dynamically change the class that it extends as well as the interfaces that it implements.

## Structure

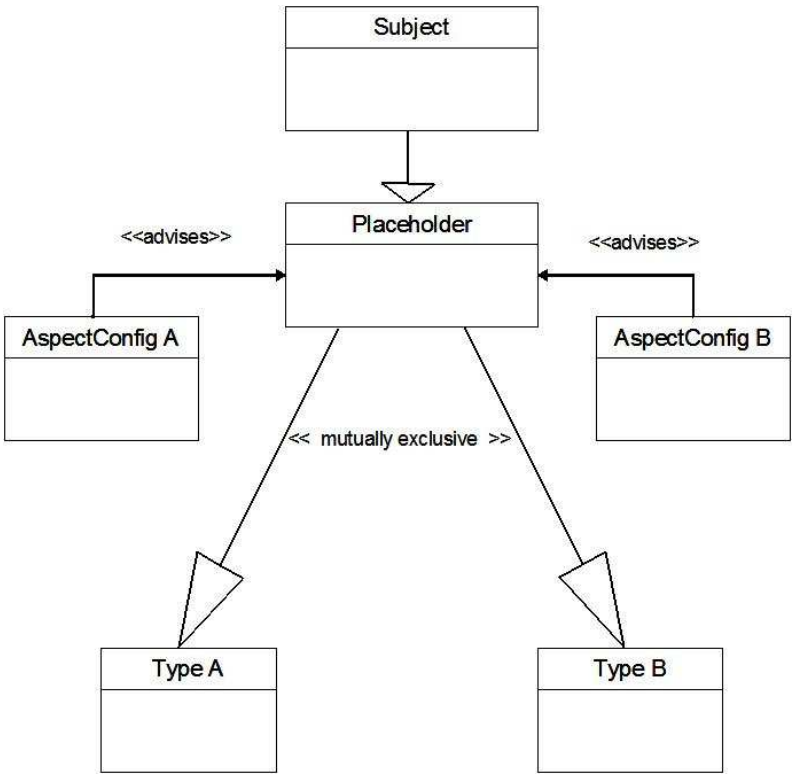


Figure 3.6: Placeholder Class pattern

## Participants

**Subject:** Class or interface whose inheritance hierarchy must change.

**Placeholder:** Empty base class or interface.

**Type A:** Type that Subject must subclass in configuration A.

**Type B:** Type that Subject must subclass in configuration B.

**AspectConfig A:** Aspect for configuration A; advises Placeholder appropriately to sublcass Type A.

**AspectConfig B:** Aspect for configuration B; advises Placeholder appropriately to subclass Type B.

## Implementation

Consider the following when using the Placeholder Class pattern:

1. Changing the interfaces or the base class may require you to implement methods that were not originally envisaged in the implementation of the Subject. Be sure to supply the necessary advice to ensure you maintain interface contracts.

2. Since Java does not allow multiple inheritance, it is possible to apply this pattern to allow the Subject to morph into multiple concrete types.

## Consequences

The Placeholder Class pattern has the following consequences:

1. The Subject cannot directly extend any auxiliary class that it may need in its implementation (since it already extends the Placeholder) because Java does not allow multiple inheritance.

2. The pattern allows aspect oriented polymorphism in the sense that it is possible for the Subject to dynamically morph into different types based on the aspect that advises it.

### Known Uses

The Placeholder Class pattern is used in FACET in the context of the CORBA abstraction. For example, the ProxyPushConsumerImpl class needs to extend the ProxyPushConsumerPOA class in one case and implement the ProxyPushConsumer interface in another. A Placeholder Class called ProxyPushConsumerBase is used as the base class for ProxyPushConsumerImpl and is appropriately advised by aspects EnableCorba and DisableCorba

### Related Patterns

This pattern is similar to the Adapter pattern but does not require one to write a number of separate classes to wrap around the Subject.

## 3.2.3 Placeholder Method Pattern

It is often necessary to change the behaviour of a certain operation (or method) based on the configuration of the application at compile time. This pattern provides an efficient method of implementing hooks and strategy methods using aspects.

### Context

Compile time configuration environment which determines the behaviour of certain operations performed by (or in) an application.

### Problem

A common necessity is the ability to provide alternate implementations of a method in a class based on configuration of a given application at compile time. Ordinarily, this is achieved through the use of the Template Method pattern, which involves subclassing a base class and then providing the appropriate implementations for all the template methods of the base class.

The problem with the above approach is that when many different variations of the methods are required, it leads to a proliferation of different classes all of which differ only slightly. Managing the instantiations and use of these classes based on the context becomes quite tedious.

## Solution

Provide a template method with a default behaviour and then use aspects specific to the configuration which advise these methods to transparently provide the correct implementation.

The class remains the same — only the implementations of its methods change according to advice from the aspects that are applied.

## Structure

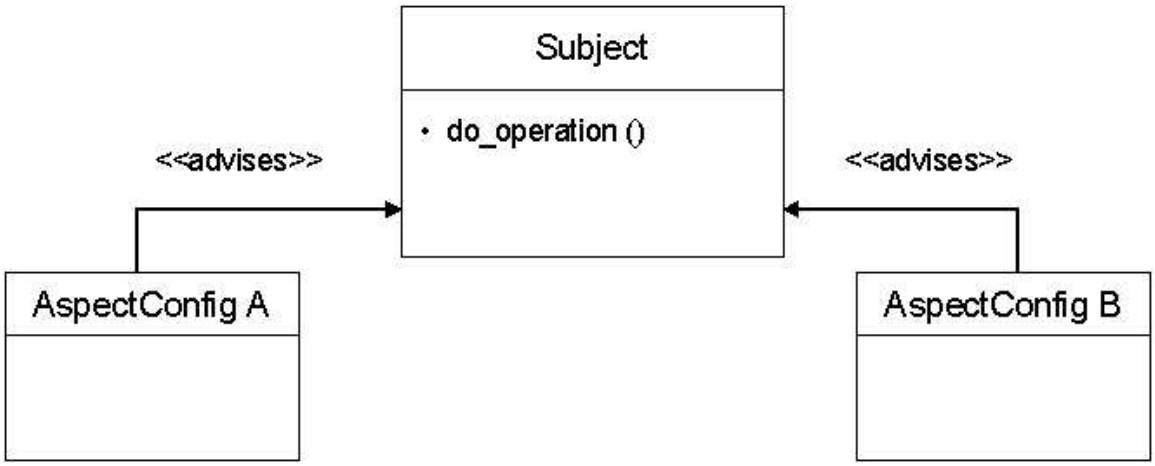


Figure 3.7: Placeholder Method pattern

## Participants

**Subject:** Class that contains the method whose implementation needs to be modified.

**AspectConfig A** : Aspect for configuration A; advises `do_operation` appropriately.

**AspectConfig B** : Aspect for configuration B; advises `do_operation` appropriately.

## Implementation

Consider the following when using the Placeholder Method pattern:

1. The default implementation of a method should ideally be empty to ensure that in order to provide a correct implementation, the appropriate aspect has to be applied

2. Advice should preferably be *around* advice so that alternate implementations can easily be provided. Use of *before* and *after* advice can potentially reduce the readability of the code.

### Consequences

The Placeholder Method has the following consequences:

1. There is a certain amount of overhead associated with advice since AspectJ makes use of hook methods for dispatching.

2. It is assumed that the different variations of the base class providing different functionality are not necessary. Once an aspect is applied, the original implementation of the method no longer exists.

### Known Uses

The Placeholder Method pattern is used in FACET in a number of ways. It is used to abstract operations specific to the ORB — *e.g.*, obtaining an ORB reference, activating the POA Manager, etc — as well as for other operations such as narrowing of object references to interface references. Since these operations have different meanings in the CORBA and non-CORBA contexts, the EnableCorba and DisableCorba aspects provide the appropriate implementations when applied.

### Related Patterns

The Placeholder Method pattern is a close cousin of the Template Method pattern.

## 3.2.4 IDL Generation

When different features are enabled in FACET, the IDL of the event channel needs to change accordingly to reflect the presence of those new features, in the case CORBA is enabled (there is no need to generate IDL when CORBA is disabled). In previous work, the changes to the IDL of the event channel were conducted by the use of scripts

which makes the changes using primitive text processing and search-and-replace style techniques [21]. However, for our purposes, using such a script would entail having to use different aspects for the cases when CORBA is enabled and when it is not. From a software engineering standpoint, this would be very inefficient and greatly reduce the maintainability of the code.

To address this, we chose to investigate a new technique which involves the generation of the IDL for a given configuration by *reflection* on the classes comprising the event channel's interface. With this, it is only necessary to specify the aspect introductions in the Java code - the corresponding changes to the IDL happen automatically since the mapping from CORBA to Java is well-known.

The generator is run as part of the three-stage build process:

- Aspects that perform introductions are applied to the classes which form the public interface of the event channel
- The IDL Generator reflects on these classes and generates the IDL. The IDL compiler is then run to generate the stub and skeleton classes. In the case CORBA is disabled, this step is automatically skipped.
- All classes comprising the event channel along with the relevant aspects for the particular feature set are compiled and the relevant jUnit [10] tests are run [21].

The IDL Generator we have developed is generic in its implementation and can be used to generate the IDL interfaces corresponding to any set of Java classes. Conceivably, this technique can be extended to any language which has a strong runtime type system and allows reflection.

## 3.3 Experimental Results

In this section, we present results that we obtained in estimating the effect that CORBA had on the footprint and throughput performance of FACET [24]. To collect such data, a set of popular configurations was identified based on feedback from several developers of the TAO users community who are using event channels in their application development. In addition, to gauge the effect of individual features on the overall size and performance of the FACET event channel, each feature was studied by measuring its effect across all configurations that included or omitted the given feature.

One method to measure the footprint of a Java application is to sum the size of all the `.class` files that are loaded. Embedded systems that use Java interpreters or just-in-time compilers could use this metric to estimate the amount of RAM needed. Another method consists of generating native code using a compiler (such as gcj [13]) and then measuring the size of the resulting executable. The compiled code is more suitable for comparisons with C and C++code. Moreover, embedded real-time applications are likely to precompile to native code for execution predictability. An overall observation has been that the size of the gcj produced object files are generally larger than the corresponding `.class` files [22]. This is commensurate with the design of `.class` files to be small so as to reduce transmission time over networks.

Here, we report results based on `.class` files that are interpreted and executed using the Sun JVM 1.4.0 with Just-In-Time compilation enabled. The experiments were performed on a dual-Xeon processor machine running RedHat Linux 8.0 at 2.40 GHz, with 512 MB of RAM.

With Java and `.class` files, the footprint of the running program increases as classes are loaded. We report the maximum footprint, achieved when all code has been loaded; such measurements are most appropriate for an embedded system. For a native-code compiled version, both the footprint and the resulting throughput are expected to increase.

### 3.3.1 Common Configurations

The following are the 10 event channel configurations used in collecting our experimental data :

1. *Configuration 1 (Base):* Although the applications requested by developers all required more functionality than the base, it is useful in that it is a lower bound on the footprint. Note that all subsequent tests use the full functionality provided by the base.

2. *Configuration 2:* Several developers only needed configurations similar to the standard CORBA COS Event Service specification. This configuration has CORBA Any payloads and does not support filtering. The pull interfaces were not included in this configuration since they were not used.

3. *Configuration 3:* This configuration is the same as the previous except that the tracing feature is enabled.

4. *Configuration 4:* Structured events and event sets are enabled. This configuration also adds the TTL field processing to eliminate loops created by federating event channels. This configuration is still minimal, however, and does not support any kind of event filtering.

5. *Configuration 5:* This configuration has support for dispatching events based on event type. It uses a CORBA octet sequence as the payload type and is a common optimization over using a CORBA Any. This configuration is similar to that used in the TAO RTEC.

6. *Configuration 6:* This configuration adds support for the event pull interfaces to configuration 5 and uses a CORBA Any as the payload.

7. *Configuration 7:* This configuration enhances configuration 5 by replacing the simple event type dispatch feature with the event correlation feature. In the corresponding application, event timestamp information was also needed, but the event pull feature was not.

8. *Configuration 8:* This configuration represents one of the largest realistic configurations of FACET. It supports the pull interfaces, uses event correlation, and adds support for statistics collection and reporting. It uses structured events carrying CORBA Any payloads and headers with all possible fields enabled.

9. *Configuration 9:* This configuration adds the tracing feature to configuration 8.

10. *Configuration 10:* This configuration is representative of that used in the Boeing Bold Stroke architecture. It includes a number of features like type filtering, event correlation, event timestamps and the real time dispatcher feature, a feature that allows consumers and suppliers to set real-time priorities on event delivery.

### 3.3.2 Footprint Measurements

As shown in Figure 3.8, the base FACET configuration (config 1) is 3 times larger when CORBA is present: 166,921 bytes with CORBA and 55,250 without. At the other extreme, one of the fuller-featured FACET configurations (config 9) has a size of 475,100 bytes with CORBA and a size of 342,226 bytes without — approximately 1.4 times larger for CORBA. This is expected since there are a significant number of

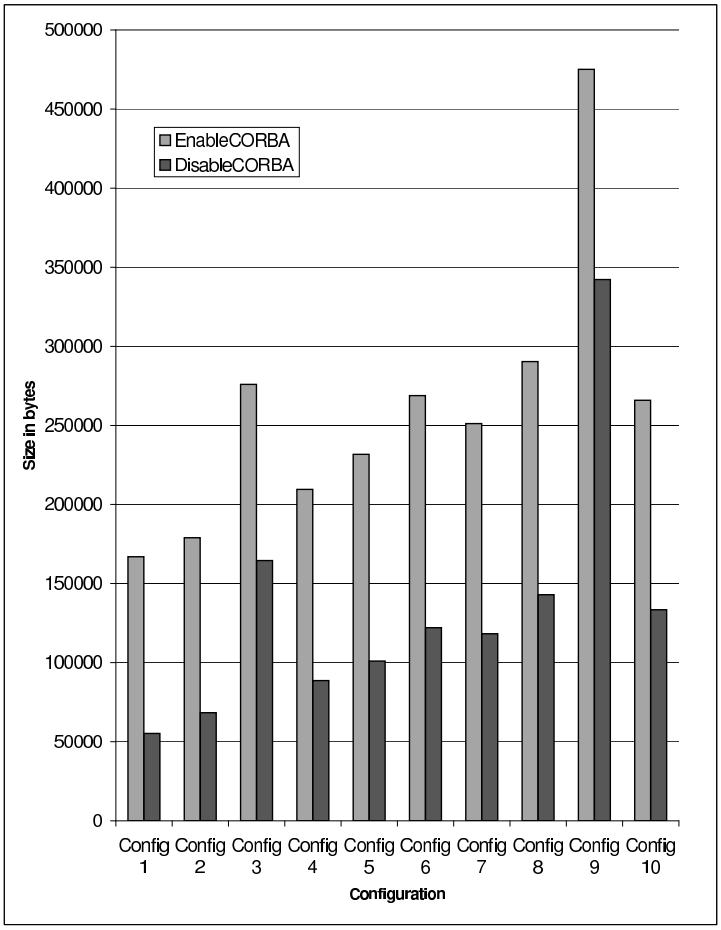


Figure 3.8: FACET footprint under different configurations

Stub and Skeleton classes that are generated by the IDL compiler, which are absent in the no-CORBA case.

It must be noted that the size of the ORB has not been included in this study. It follows that if it were indeed included in these measurements, there would be an even bigger difference in the footprint observed. Generally, in full-featured ORBs such as JacORB [5] that are not subsettable, the most casual reference to the ORB causes the entire ORB to be included in the resulting executable code. While ORBs vary in size [12], and some ORBs do offer reduced-feature versions, the choice of which features to include or omit is not made on an application-specific basis. Conceivably, our AOP approach for including features in an event channel could be extended to include only those ORB features needed to support a given event-channel configuration.

Figure 3.8 shows that the disabling of CORBA for the configurations we considered reduced footprint by about half in most cases — an appreciable savings for small embedded systems.

### 3.3.3 Throughput Measurements

Figure 3.9 shows the difference in throughput performance with and without CORBA. When configured as the standard CORBA COS Event Service [29], the throughput with CORBA enabled was 1651 events/sec as compared with 131,758 events/sec without — a difference of 2 orders of magnitude! This can be explained by the fact that the Java ORB, JacORB, does not include optimizations for collocated objects which means that the Stubs and Skeletons perform marshalling and communication over network sockets assuming a truly distributed system. With an ORB such as TAO that does include such optimizations, the performance difference is likely to be less dramatic but still substantial.

This level of improvement without CORBA held for all configurations of the event channel that we studied with the exception of configurations which included the tracing feature (configs 3 and 9). The reason for this can be attributed to the enormous amount of code weaved in by the AspectJ compiler onto all the methods of every class in the event channel, when the tracing feature is enabled. The overhead of these extraneous method calls to the log4j [1] logging library contribute significantly to performance degradation and to the size of the footprint. This observation is consistent with findings in a previous study [22].

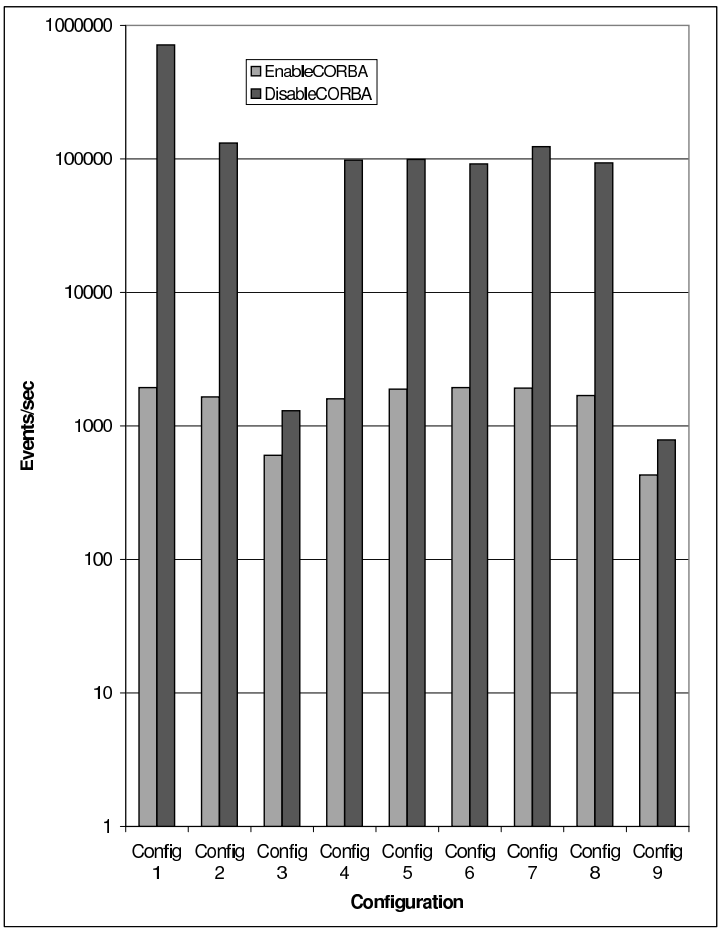


Figure 3.9: FACET throughput under different configurations

### 3.3.4 By-Feature Study

We next measured footprint and throughput for various configurations in which only a single feature (and features upon which it depends) was enabled at a time. This experiment quantifies the the size contribution and throughput degradation of a given feature.

Figure 3.10 shows footprint reduction by-feature, with and without CORBA. For an embedded system, even modest savings can be crucial to a component's cost.

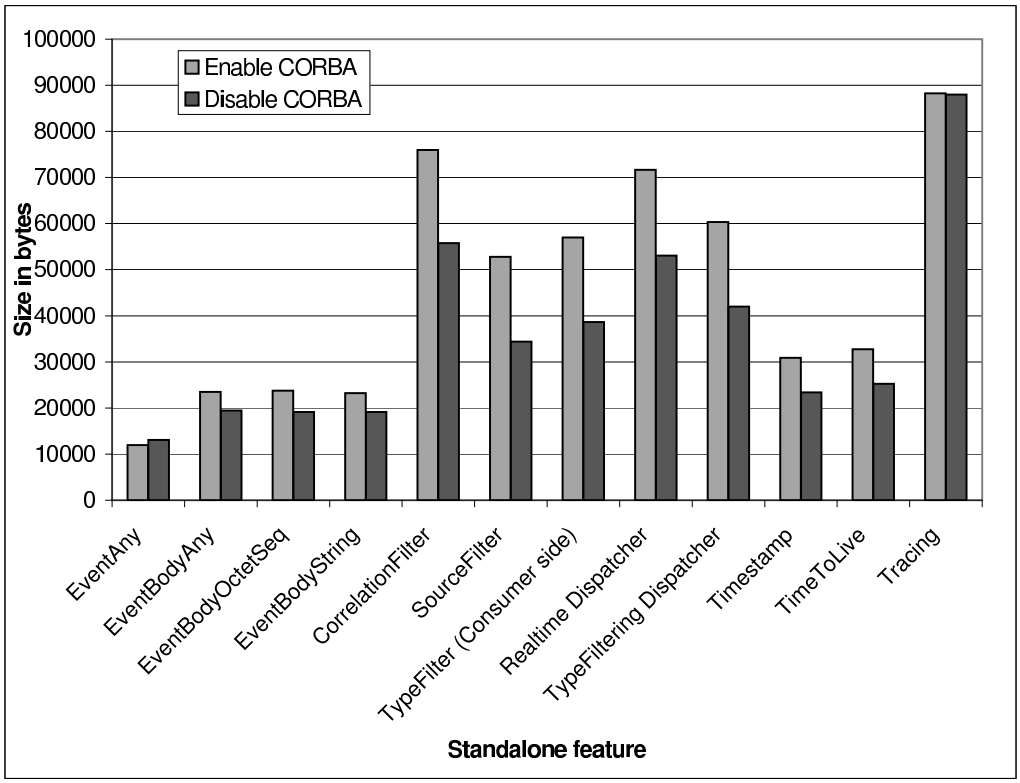


Figure 3.10: Impact of different features on footprint

A much greater impact can be seen as we study performance. As shown in Figure 3.11, the difference for each feature with and without CORBA is dramatic. The interesting observation is that no difference is observed *among* the features when CORBA is disabled. An explanation for this is that the aggregation of features on the base and the overhead associated with the code weaved in by the AspectJ compiler is negligible,so that the throughput at this point is limited by the operating system and/or hardware. This indicates that the throughput performance of the

event channel with CORBA disabled is at a maximum and is quite unaffected by the feature set (again with the exception of the tracing feature).

It can be argued that a true measure of the average effect of a feature on the footprint and throughput of the event channel can be obtained by measuring the overhead over the set of all possible valid combinations that differ by that one feature [22]. We plan to investigate this line of experimentation in future work. However, when the number of features is large as is the case in FACET's current code base, the number of valid combinations make it quite impractical to run through each one of them. In such cases, a more intelligent method of grouping features is necessary.

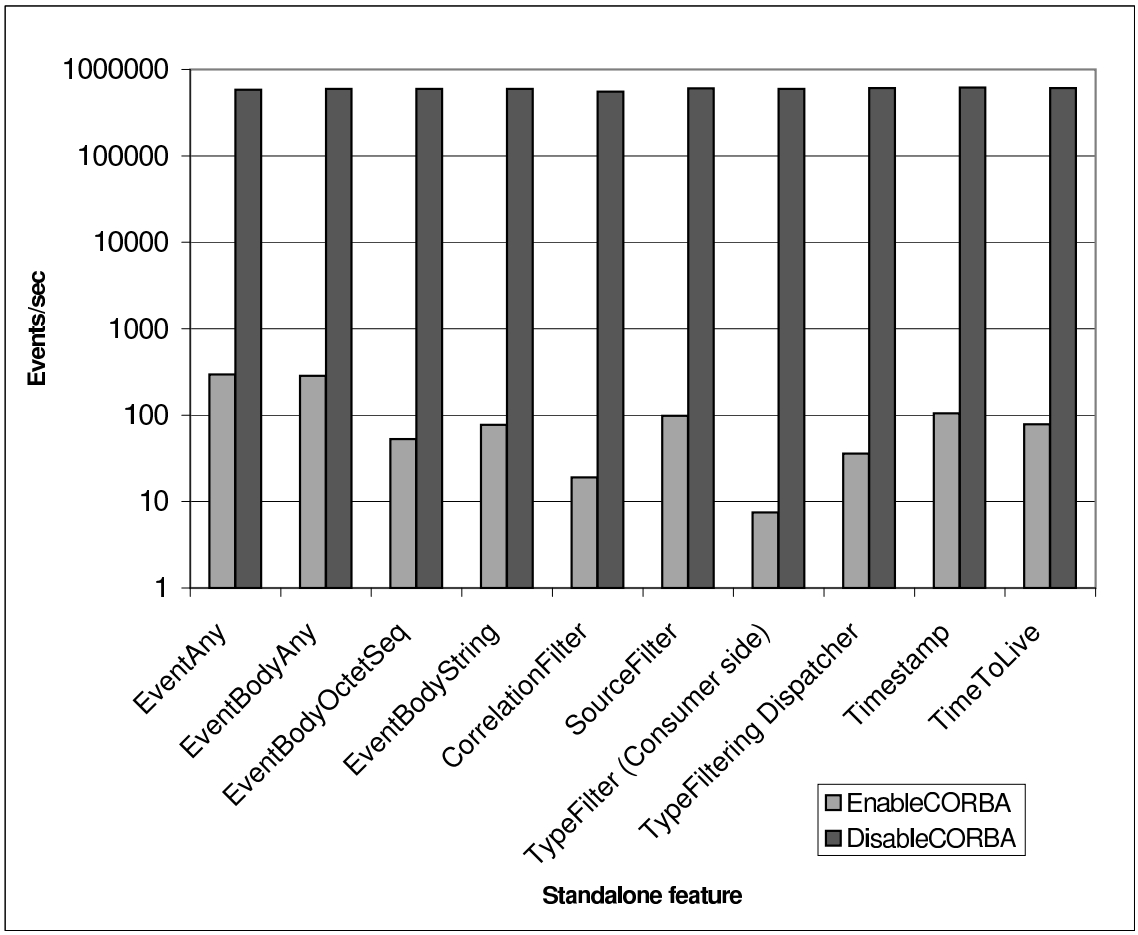


Figure 3.11: Impact of different features on throughput

## 3.4 Using Java RMI for the Transport Layer

We have demonstrated how it is possible to abstract the use of CORBA as the transport layer in FACET. Since CORBA is yet another object-oriented mechanism, it should be possible to take this idea further and provide a way to select between multiple Remote Procedure Call (RPC) mechanisms merely by selecting the appropriate feature.

Java RMI is one such object-oriented RPC mechanism very similar to CORBA in that it provides a way to to invoke a method on an object that exists in another address space (either on the same machine or a different one). However, CORBA differs from RMI in a number of ways:

- CORBA is a language-independent standard.
- CORBA includes many other mechanisms in its standard (such as a standard for transaction processing monitors) none of which are part of Java RMI.
- There is also no notion of an ORB in Java RMI.

In what follows, we take a preliminary look at how the patterns and techniques developed earlier apply in the case RMI is used as the transport layer.

### 3.4.1 Applying the Placeholder Class pattern

RMI has a model similar to that of CORBA that involves the interaction of three processes:

1. A *Client* is the process that invokes a method on a remote (or *Servant*) object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are registered with the Object Registry (similar to the Naming Service in CORBA).

Since there is no IDL, pure Java interfaces are used to publish the operations supported by a Servant object. A remote interface has the following properties:

- It is a public interface

- It extends java.rmi.Remote
- Every method in the interface must declare that it throws java.rmi.RemoteException. It may also throw other exceptions.

Let us consider the PushConsumer interface as before. In the case Java RMI is enabled, we have:

```
public interface PushConsumer
   extends java.rmi.Remote {

   public void push (Event data)
      throws java.rmi.RemoteException;
}
```

And in the case RMI is disabled, the interface looks like the following:

```
public interface PushConsumer {
   public void push (Event data);
}
```

Clearly, since we need to change the inheritance hierarchy dynamically as before, we apply the Placeholder Class pattern.

A Java RMI Servant must extend the java.rmi.server.UnicastRemoteObject class in addition to implementing the interface. In the non-RMI case, the class merely implements the interface. The Placeholder Class pattern once again makes this possible in an efficient manner allowing aspect-oriented polymorphism.

In a similar way, the PlaceHolder Method pattern can also be applied to allow locating an object either through the Object Registry in the RMI case or through the CORBA Naming Service in the CORBA case.

# Chapter 4

# Integrating FACET with the Boeing OEP

The Boeing OEP is a framework that aims to provide the fundamental reference architecture for the next generation of large-scale component-based embedded systems. Its open run-time framework includes middleware services and architectural support for multiple strategies — such as multiple processors and system demonstration platforms.

To provide a robust and highly-configurable component-oriented system, the OEP makes use of much of the infrastructure provided by the ADAPTIVE Communication Environment (ACE) and the ACE Object Request Broker (ORB) (TAO). For instance, the real time event channel used in the OEP is the TAO Real-Time Event Channel (RTEC) [17], an implementation of the Object Management Group (OMG) Event Service Specification, with support for important features and QoS optimizations required by advanced distributed simulation systems.

However, the RTEC suffers from problems common to most traditional middleware libraries—it was not designed from the ground-up to support pluggability of finely composed features and cannot be easily customized for a particular set of features. As a result, applications can pay a performance and/or footprint penalty due to present, but unused features in the library. Since the Framework for Aspect Composition for an EvenT channel (FACET) is a compliant Event Service implementation with support for fine decomposition of features, we consider here the replacement of RTEC in the OEP with FACET.

In this chapter, we describe our evaluation of the alternatives and how we used gcj and CNI to solve the problem. We describe our methodology and present data on the efficiency of such an approach.

## 4.1 Motivation

A compelling reason driving the adoption of FACET in small-scale embedded systems is its significantly lower footprint. Figure 4.1 compares the contributions of various components to the footprint of the liborbsvcs.so library that forms an integral part of the OEP. Since RTEC depends on ACE, TAO and others such as the Naming Service, these libraries are included automatically. FACET, however, has a footprint that is nearly 1/4th that of RTEC. In addition to being much lighter, FACET also does not depend on ACE and TAO. Completely removing those libraries from the footprint would result in a significantly downsized liborbsvcs.so. The only additional dependency introduced through the use of FACET would be on libgcj. Although libgcj is currently a large monolithic library (about 32MB), since FACET makes use of only a small fraction of the library, it is reasonable to argue that the increase in footprint of the running application, due to FACET, is a significantly smaller number.

Tackling the problem of integrating FACET with RTEC and TAO involves the fundamental problem of making code in C++ interact with code written in Java and as part of the integration, we aim to replace the RTEC, written in C++, with FACET, written in Java. In doing so, we want to preserve the external interfaces of the RTEC and ensure that all clients are completely unaware of the change.

In attempting this, we are naturally interested in:

- A highly efficient implementation—There should be almost no loss in performance in using Java code. Moreover, there should be a reduction in footprint attributable to using FACET [22].
- Transparency—All clients of a particular class are unaware whether it is implemented in C++ or Java.
- Ease of programming.
- Small Adapter layer—The glue layer between the C++ and Java classes must be minimal in size.

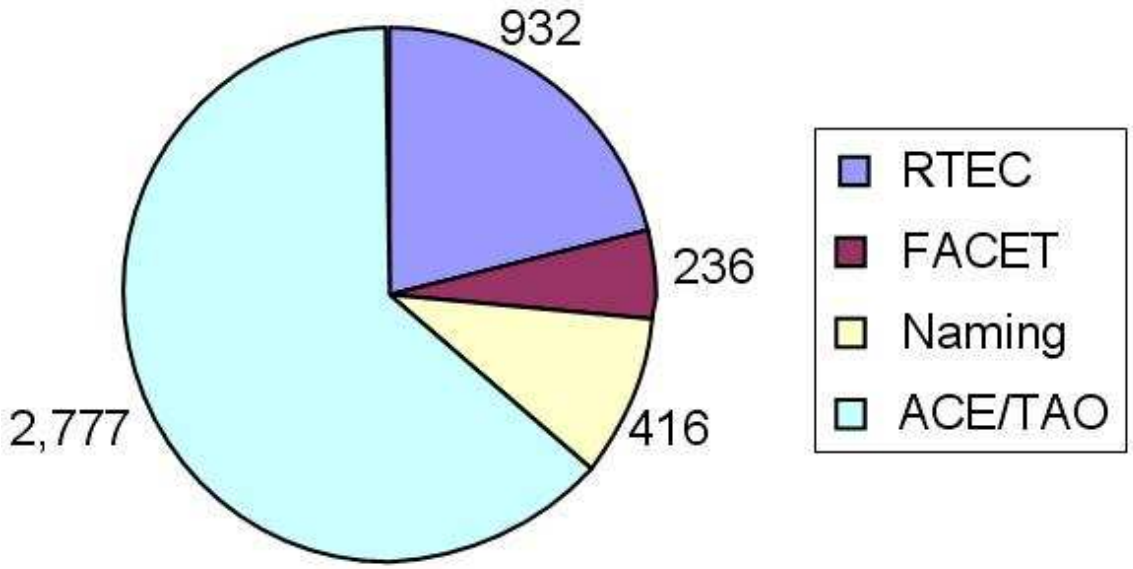


Figure 4.1: Footprint of liborbsvcs.so (kilobytes)

Interoperability between Java and native code (written in C or C++) is in currently possible using one of the following approaches:

**JNI:** A standard programming interface for combining Java and native methods and for embedding the Java virtual machine into native applications. The primary goal is binary compatibility of native method libraries across all Java Virtual Machine (JVM) implementations on a given platform.

**CNI:** An alternative to JNI, CNI simplifies authoring Java native methods using C++. It is an efficient, less tedious, but somewhat less portable alternative to JNI since it is currently implemented only by gcj [13]. However, there is nothing that impedes the implementation of CNI in other JVMs [7]. CNI is completely different from JNI in that it deviates from the JNI philosophy of completely hiding the Java object model from the native programmer.

What remains is an evaluation of JNI and CNI to see which alternative offers the greatest benefits.

## 4.2 JNI vs. CNI for the Boeing OEP

One of the major advantages in using JNI is that of portability. Since it is a standard supported by most commercial and free JVMs, it is possible to write native code that works without modification on any JVM on a particular platform. This benefit however, is offset by a number of other disadvantages:

- JNI code is rather tedious to write. Type safety is sacrificed since access to object pointers is not guarded.
- Effort required to maintain code is high because of the increased size of the code base (Java code, native code, and interface code).

CNI is the interface-to-native infrastructure provided by gcj. Although not strictly a standard, it is implemented in the GNU family of compilers and is therefore available on a reasonably large number of platforms. It essentially allows C++ and Java code to interact with each other as if there were no distinction between the classes written in either language. CNI uses C++ namespaces to implement Java packages, leading to relatively intutive refererences for a specific class. For instance, the Java class java.lang.String maps to java::lang::String in C++.

```
int runTest (TestJavaClass *tjc)
{

  /* Call the method */
  tjc->methodNoArg ();

}
```

Figure 4.2: Java method call using CNI

However, there are some disadvantages associated with CNI. A major drawback is the lack of portablility—using CNI entails using a C++ compiler that understands the CNI interface, such as gcc. There is, however, a growing interest toward CNI-like native interfaces, especially in JVMs for embedded and real-time systems. For instance, the Juice++ JVM [7] which was specifically designed for high performances and small footprint embedded systems, provides a native interface which is based on, and extends, CNI.

The other disadvantage is the reduction of safety — a pointer to any object in Java programming environment can obtained in the C++ environment. Since Java

has consciously been designed to restrict the user from obtaining low-level pointers to objects and to prevent other such unsafe operations, this feature of CNI actually allows native code to perform unsafe pointer operations on Java objects.

```
JNIEXPORT void JNICALL
Java_TestLauncher_testLogic (JNIEnv *env,
                             jobject obj)
{
  /* Get class object from JVM */
  jclass cls = env->GetObjectClass (obj);

  /* Get method id from JVM */
  jmethodID method_id =
  env->GetMethodID (cls,
                    ``methodNoArg'',
                    ``()v'');

  /* Call Java method on 'obj' Object */
  env->CallVoidMethod (obj, method_id);
}
```

Figure 4.3: Java method call using JNI

Figure 4.2 demonstrates how making a Java method call using CNI is completely seamless. In this case, the caller is completely unaware that the callee is actually a method implemented in a Java class. This feature satisfies a key criterion of our evaluation – that the callee code be oblivious to the language in which a particular class is implemented.

By contrast, Figure 4.3 demonstrates how the same task can be achieved using JNI. This essentially involves the following steps:

- Using the JVM environment parameter a call, GetObjectClass (), is made to the JVM requesting the Java object.
- Using the class handle, another call, GetMethodID (), is made to the JVM, this time requesting the handle for Java method. To obtain a handle, the method name and its signature are passed are arguments.
- Lastly, using the appropriate invocation method *e.g.*, `CallVoidMethod`, and the method handle, the call to the Java method is completed.

As is evident, the steps involved are tedious and non-intuitive. In addition, there is also a basic lack of type safety.

## 4.3 Comparing the Performance of CNI and JNI

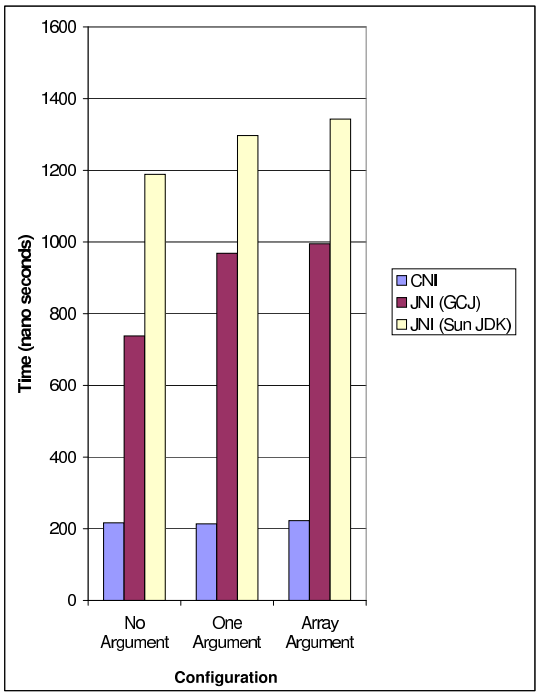


Figure 4.4: Average time for a method call — CNI vs JNI

Experiments were conducted on a single processor machine running at 2.53GHz with 512 MB of RAM, to measure the time taken to complete a method call in both JNI and CNI, in a variety of cases:

1. Invoking a method with no arguments
2. Invoking a method with a fixed number of parameters and no return value
3. Invoking a method with an array as the only parameter and no return value

Figure 4.4 shows that a CNI call is up to 5 times faster than the corresponding JNI call. The average time for a method call, irrespective of the signature, is almost constant in the case of CNI. Since gcj also implements JNI, it was possible to compare its performance with that of the Sun JDK compiler. As expected, ahead of time compilation of JNI code performs much better than JIT compilation.

The superior performance of CNI coupled with the ease of programming, provided a compelling enough reason to go with using CNI in integrating FACET with the OEP.

## 4.4 The Adapter Layer

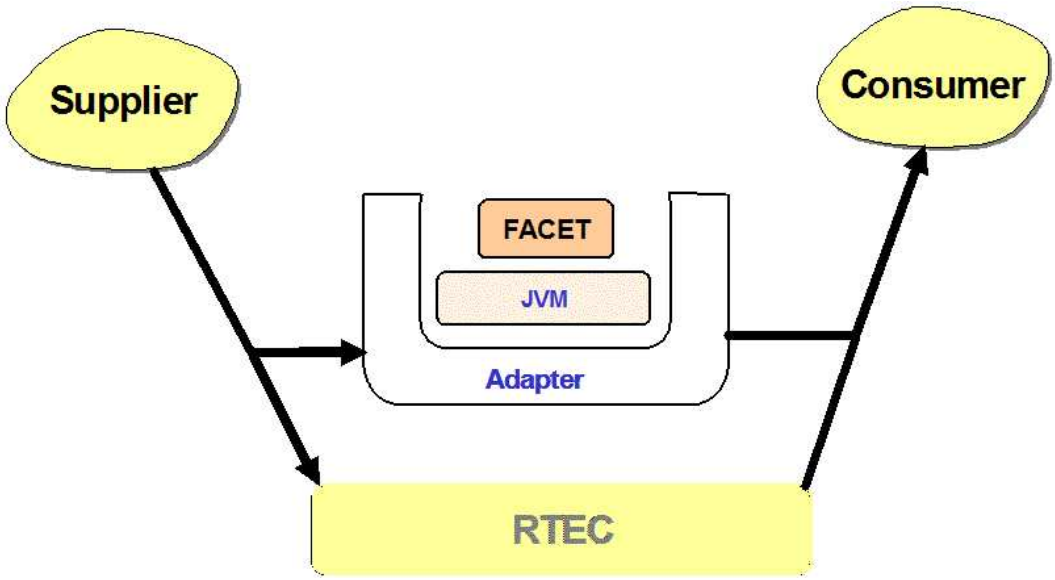


Figure 4.5: Adapter Using JNI

As stated earlier, the basic goal of our effort was to replace RTEC with FACET in the OEP. Our approach was to achieve the replacement in a non-intrusive way so that suppliers and consumers are oblivious of the actual agent implementing the event channel interface.

This was achieved by the use of an *adapter layer* that intercepts interface method calls to RTEC and routes them instead to FACET using peer objects that correspond to their RTEC counterparts. There is almost no overhead in marshalling and demarshalling in the case that there are no array arguments. When there are array arguments, however, the overhead in marshalling is minimal and is far more efficient in CNI than it is in JNI. This is clearly shown in Figure 4.4.

Figure 4.5 shows the possible approach of building the adapter layer using JNI. Here the layer acts as the interface between the supplier and consumers, and FACET. In this case, however, every call has to pass through the JVM leading to

a considerable slow down in method invocation in addition to increasing the overall footprint. Considering the scale of the OEP and given the potential environment of its deployment, the performance degradation can be substantial.

Figure 4.6 highlights the approach we followed in the integration of the OEP and FACET. The Java runtime environment is now unnecessary because all code is now run natively by the operating system. The only support needed by Java classes executing natively is provided by libgcj. The slight disadvantage of this approach, however, is that the footprint of libgcj is considerable. This is attributed to the fact that libgcj is currently too monolithic and hard to subset. However, efforts to address this issue are currently in progress.

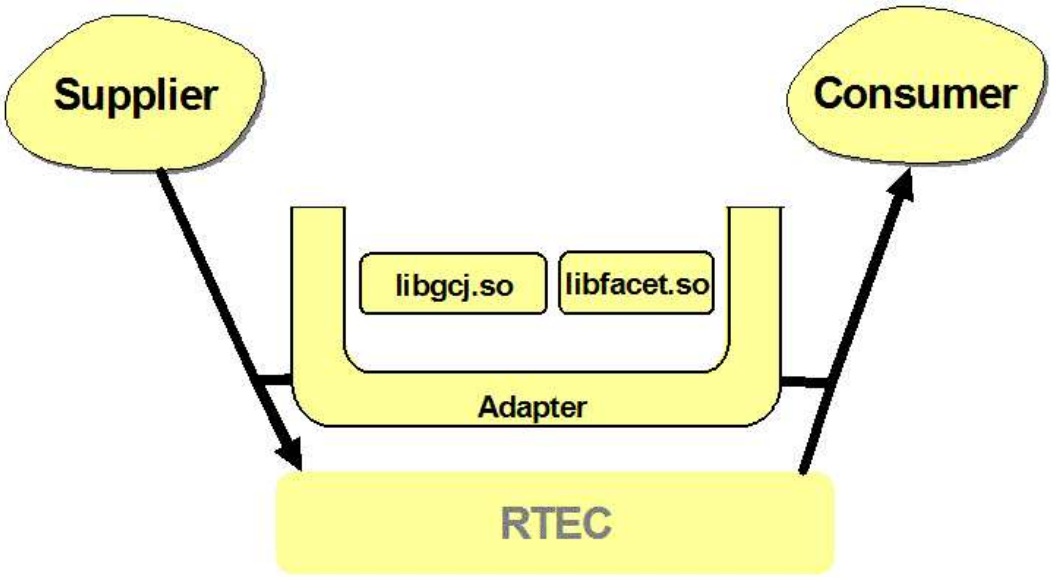


Figure 4.6: Adapter Using CNI

The adapter layer solution proposed has certain other interesting applications. One of the things that such an approach gives us is the ability to build cross-language the Common Object Request Broker Architecture (CORBA) Servant [27] objects. As described in Section 4.6.1, the CORBA Servant object is implemented in Java and registers itself with TAO, a C++ ORB. In this manner, Servant objects written in Java can exploit the features provided by a C++ ORB. Examples of such features include the collocation-optimization and support for advanced features such as Interoperable Object Group Reference (IOGR)s [27].

## 4.5 Issues in Integration

The process of mixing Java and C++ using CNI was not without its hiccups. In what follows, we describe some of the issues and limitations that we came across and some of the workarounds needed to circumvent them.

**Bootstrapping the Java Runtime** A C++ application intending to use Java objects needs to ensure that the Java runtime (libgcj) has been loaded before creating Java objects or invoking methods on them. To do this, we use a "bootstrapping" Java class, whose only job is to transfer control to the relevant C++ start function, as the entry point for the executable. Essentially, a launcher Java class is written as shown in Figure 4.7. The responsibility of this class is to delegate control to a native function. During the execution of this class the Java environment would have been intialized and booted up allowing operations on Java classes.

```
public class Bootstrap {

  /* Native method to transfer control */
  private static native int transferControl
                          (String[] args);

  /* Invoke the native test logic.*/
  public static void main (String[] args)
  {
    Bootstrap.transferControl (args);
  }
}
```

Figure 4.7: Bootstrapping Java class

Figure 4.8 is the C++ function `transferControl` which coverts the necessary command-line arguments before transferring control to the actual C++ application. Note that the `main` function of the C++ application should be altered as shown in Figure 4.9. The conditional statements change the signature of the `main` by renaming it to `j2cmain` so that there is no conflict with the `main` of the boostrapping Java class.

```
#include "Bootstrap.h"
#include <gcj/cni.h>

extern int j2cMain(int argc, char **argv);

Bootstrap::transferControl (
  JArray <::java::lang::String *> *jargs)
{
  int argc;
  char **argv;

  /* Converts Java arguments
     into argc, argv */
  j2cArgs (jargs, argc, argv);

  /* Call the C++ application */
  j2cMain (argc, argv);
}
```

Figure 4.8: Transferring control to C++

```
/*
 * This is the actual C++ program's main.
 *
 * Since there already exists a main in the Java world, we
 * suitably rename this
 */

#if defined (WITH_J2C_MAIN)
  int
  j2cMain (int argc, char *argv[])
#else
  int
  main(int argc, char* argv[])
#endif
```

Figure 4.9: Remapping the `main`

**Interaction of C++ objects and the Garbage Collector (GC)** A key feature of Java is its garbage-collected heap, which takes care of freeing dynamically allocated memory that is no longer referenced. Because the heap is garbage-collected, Java programmers don't have to explicitly free allocated memory. The GC runs in a separate thread and collects objects that are no longer referenced by the program. Since Java compiled to native code is not run by a JVM, libgcj features a conservative garbage collector — the Boehm garbage collector — to support this important language feature.

When Java objects are instantiated in C++ threads that the GC cannot see, it may not correctly account for references that may exist from these threads and may thus reclaim objects even if they are actually live! This problem usually arises when the C++ world does not register threads with the GC. To solve this problem, we ensure that the `pthread_create` call is mapped to the appropriate call provided by the Boehm GC. This ensures that all C++ threads are registered with the GC allowing it to correctly track all Java object references.

Since the GC uses `SIGPWR` to start and stop all threads that are running, it is essential for C++ threads to appropriately handle these signals. Fortunately, remapping the `pthread_create` as described above takes care of this for us too.

## 4.6 Experimental Results

In this section we present quantitative data on a number of experiments that were conducted to measure the:

- Performance of a simple CORBA client and server application implemented using different techniques and on different ORBs.
- Relative performance of RTEC and FACET when integrated into the OEP.

These experiments were performed on a single processor machine running at 2.53GHz, with 512 MB of RAM and all processes were collocated.

### 4.6.1 CORBA with CNI

As described in Section 4.4, using CNI it is possible to implement a CORBA Servant in Java and have it run on top of a C++ ORB like TAO. Following up on this, we measure the performance of a simple client and server application in which the

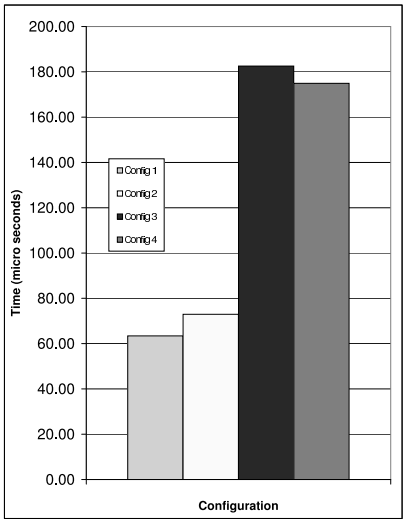


Figure 4.10: Average time for a method call with a single parameter

servant is implemented in a number of different ways. The configurations used for the experiment are:

- *Configuration 1:* C++ client; C++ servant on TAO.
- *Configuration 2:* C++ client; Java servant with C++ Adapter Layer on TAO.
- *Configuration 3:* Java client; Java servant on JacORB [5].
- *Configuration 4:* Java client; Java servant with C++ Adapter Layer on TAO.

The server provides the following operations in its Interface Definition Language (IDL):

```
interface Operator
{
  void performOperation
     (in Operation Event);

  void performOperationSet
     (in OperationSet EventSet);

  long resultOfOperation ();
};
```

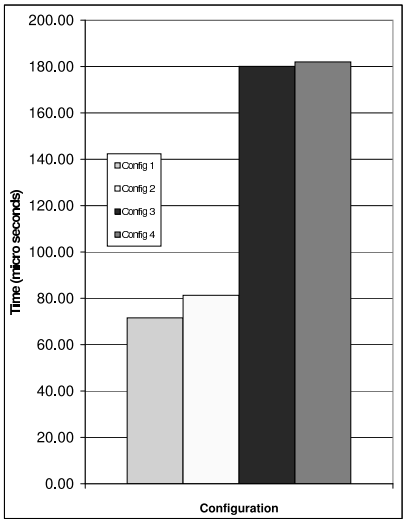


Figure 4.11: Average time for a method call with an array parameter

An operation which takes an array parameter was provided in order to measure the overhead in marshalling array data.

In each of the configurations listed above, we measure the average time required for the completion of the invocation of a method on the server. The goal of these experiments is to observe the impact on performance, if any, that CNI has, due to overhead from marshalling and intervention from the libgcj runtime.

By adopting the CNI approach, in addition to being able to host a CORBA servant written in Java on top of a C++ ORB (TAO), there is only a 13% slowdown in the case of the single parameter method call and a 11% slowdown in the case of the array parameter. Figure 4.10 and Figure 4.11 illustrate these clearly.

Another point worthy of note is the considerably better performance of TAO as compared to JacORB. A possible explanation for this could be that TAO performs a number of optimizations when objects are collocated.

### 4.6.2 FACET vs. RTEC

As a final test of the efficacy of the techniques investigated in this thesis, we compare the performance of RTEC with that of FACET. Figure 4.12 compares the throughput of RTEC and FACET (in events/sec) when both event channels were configured to perform basic filtering. We find that RTEC performs only slightly better than

FACET. The performance degradation due to slipping FACET, a Java event channel, into the OEP seems to be a very modest 5.5%. Since our FACET Adapter is currently unoptimized, it should be possible to improve upon the current performance. Some of the optimizations that we are plan to investigate in the the future are:

- Efficient marshalling of array data
- Re-use of miscellaneous helper objects
- Efficient caching of C++ data needed on the Java side

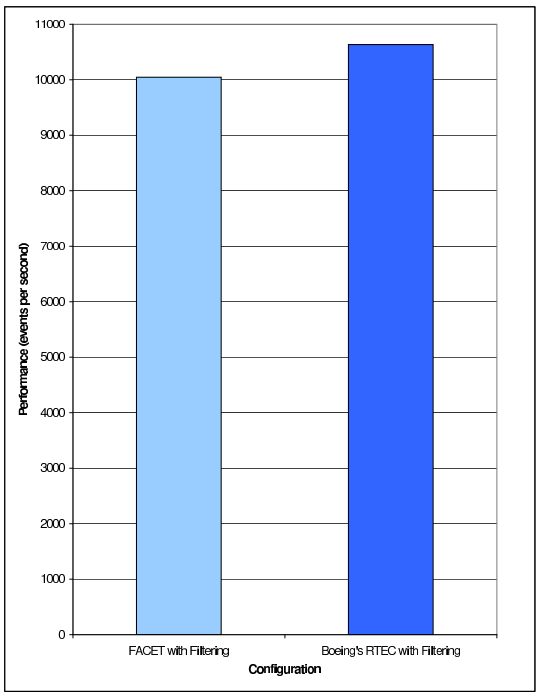


Figure 4.12: Original RTEC vs Integrated FACET

# Chapter 5

# Conclusions and Future Work

As embedded software continues to grow more complex and participate even more in distributed systems, the need to use standard middleware becomes even more imperative. While frameworks such as the ADAPTIVE Communication Environment (ACE) and the ACE Object Request Broker (ORB) (TAO) do reasonably given the constraints of embedded systems and meet the needs of existing large-scale embedded systems, small-scale embedded systems need more and the use of Aspect-Oriented Programming (AOP) does seem promising. In this context, the goal of this thesis was to discover methods for building highly efficient, customizable middleware.

While significant advances have been made in subsetting middleware, with precise control over footprint and feature set, the use of transport mechanisms such as the Common Object Request Broker Architecture (CORBA) is redundant in scenarios where objects are collocated and written in the same language. In this thesis, building on known AOP techniques, we have abstracted the transport layer of the the Framework for Aspect Composition for an EvenT channel (FACET) event channel such that use or non-use of CORBA can be specified at build-time. We thus provide more efficient customization of an event channel for a particular application. In carrying out the abstraction, we discovered two aspect oriented patterns — the Placeholder Class and the Placeholder Method patterns — and found an efficient method for Interface Definition Language (IDL) generation by reflection on Java classes. We have also presented the results of the measurement of the impact of CORBA on the footprint and throughput of the event channel for popular event service configurations presently in use by members of the TAO user community.

As future work, we intend to take the transport layer abstraction further and to support other transport mechanisms such as Java RMI [35] through encapsulation

in a feature. We are also investigating extending FACET to make real-time guarantees about event delivery and studying the design patterns involved in building such customizable middleware embedded systems.

In this thesis, we also investigated the possibility of replacing the Real-Time Event Channel (RTEC) in the OEP with FACET. In the process we evaluated two possible approaches for achieving this — JNI and CNI — both in terms of performance and footprint. We found that using CNI was the most efficient in terms of performance and ease of use. Although our results show that we are slightly behind RTEC in terms of performance, we are yet to optimize our code and we believe with tuning it should be possible to match, if not surpass, RTEC.

Some of our future plans, in this area, involve optimizing the adapter layer and possible automation of the process of generation of the adapter layer. In this connection, we plan to work on reducing the footprint of FACET by subsetting libgcj. We also plan on exploring the implementation of client-side adapter layers allowing cross-language CORBA Stubs in a style similar to the cross-language CORBA Servants described in this thesis. This would make it possible for Java clients, for instance, to access exclusive features provided by a C++ ORB like TAO. We also intend to provide additional interoperability and reconfigurability of Fault Tolerant RTEC implementations, starting with the pioneering fault-tolerant real-time event channel developed at Washington University.

And finally, we have done preliminary exploration of the problem of exhaustive testing of all possible configurations of FACET. Since the number of possible configurations rises exponentially with the number of features, we propose an intelligent grouping of non-interfering features so that we could reduce the tractability of the problem. We have examined some basic conditions under which a set of aspects can be termed non-interfering. As future work, we hope to take this idea further and develop more accurate (and therefore, less restrictive) requirements on aspects to be non-interfering.

# Appendix A

# Glossary

**advice:** Code contained in an aspect that is executed at the locations of its associated joinpoints.

**aspect:** An *aspect* is a specification of a cross-cutting concern.

**base:** As used in this thesis, the *base* refers to the core set of code that supports a fundemental level of functionality. This functionality is indivisible, and features are used to extend and enhance it.

**cflow:** A *cflow* or control flow specification describes an execution path joinpoint. Variables and data available at both the beginning and end of the execution path can be used in advice.

**feature:** A *feature* is a cohesive set of code (classes and aspects) that provides a specific functional or structural enhancement to the base.

**introduction:** An *introduction* statically adds member variables or methods to existing classes and interfaces.

**joinpoint:** A *joinpoint* is a well-defined point in a program such as a invoking a method or accessing a class member variable.

**pointcut:** A *pointcut* is an expression containing joinpoints that can identify a set of well-defined points.

# Appendix B

# Non-interference of Aspects

The architecture of the Framework for Aspect Composition for an EvenT channel (FACET) is quite different from other event channels such as Real-Time Event Channel (RTEC) in that the various features it provides have been decomposed into highly granular units that each represent some fundamental functionality of the event channel. Since some features are dependent on some others, this inter-dependence needs to be satisfied in any composition of the features to produce a working event channel for a given scenario. Figure B.1 describes the various relationships between features in FACET.

An important part of FACET's build framework is the support for extensive testing for all valid combinations of features. While the number of features in earlier versions of FACET made it possible to actually perform the testing, as the number of features have grown, this has quickly become impractical since the number of valid configurations increases exponentially with the number of features.

In this appendix, we attempt to take a preliminary look at this issue by examining under what conditions features can be deemed *non-interfering*. By identifying non-interfering features, we hope to reduce the intractability of the problem by reducing the extensive testing to a smaller subset of inter-dependent features.

## B.1 Testing Non-interfering Features

Software testing is a necessary and important part of any application. In this context, proper testing for FACET is even more important for two main reasons:

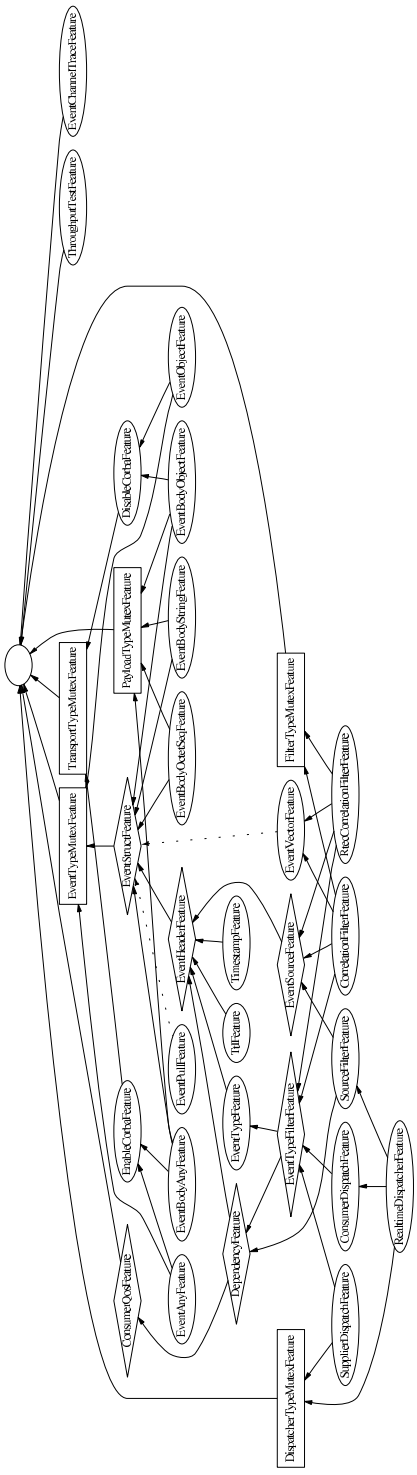


Figure B.1: FACET Feature Dependence Graph

1. Since FACET supports a large number of different configurations of features, validating a subset of legitimate configurations does not guarantee that every configuration will work or even compile.

2. It is difficult to verify that a change made to the base or a feature does not remove or change the semantics of a joinpoint used in another feature.

Given $n$ features, the total number of possible configurations is $2^n$. Since there are inter-dependences between the various features [20], the number of valid configurations is a smaller number, but still rises exponentially with the number of features.

A key observation is that if we find that a set of features are non-interfering, it would suffice to test each feature in that set in isolation — all combinations of those features would then be guaranteed to work correctly because of the fact that they are non-interfering.

So if the $n$ features are divided into $k$ sets in which each set consists of features that are mutually non-interfering, the number of possible combinations that have to be tested is $2^k$, which is still exponential in the number of non-interfering sets. When k is significantly smaller than n, the intractability of the problem is greatly reduced. The disadvantage with such an approach, however, is that in the worst case, when each non-interfering set has a cardinality of 1, there is no improvement.

In what follows, we draw on program slicing [38] and program interference [19] theory to determine under what conditions features are non-interfering and can be grouped together into a set.

## B.2 Program Slicing

Program slicing, originally introduced by Weiser [38], is a decomposition technique which extracts program elements related to a particular computation from a program. A *program slice* consists of those parts of the program that may directly or indirectly affect values computed at some program point of interest, referred to as a *slicing criterion*. Informally, a slice provides the answer to the question, "What program statements potentially affect the computation of variable $v$ at statement $s$ ?"

Although there are numerous ways of defining a slicing criterion, we adopt the approach similar to Ottenstein and Ottenstein [32] who define a slicing criterion to consist of a program point $p$ and a variable $v$ that is defined or used at $p$. The slice

is computed as a graph reachability algorithm on a program dependence graph to compute a slice that consists of all statements and predicates of the program that may affect the value of $v$ at $p$.

The following terminology is used in the discussion on the computation of slices:

**Directed Graph** A directed graph $G$ consists of a set of vertices $V(G)$ and a set of edges $E(G)$, where $E(G) \subset V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from $b$ to $c$.

**Program Dependence Graph** The program dependence graph for a program $P$, denoted by $G_P$, is a directed graph whose vertices are connected by edges that represent either *control dependence* or *data dependence* [32] between the various statements in the program.

**Program slice** For a vertex $s$ of a program dependence graph $G$, the slice of $G$ with respect to $s$, written as $G$ / $s$, is a graph containing all vertices on which $s$ has a transitive flow or control dependence (*i.e.* all vertices that can reach $s$ via a data or control dependence edge): $V(G/s) = \{w \in V(G) | w \rightarrow^*_{c,f} s\}$

## B.3 Program Interference

Given a program *Base* and two versions $A$ and $B$ of that program that may be arbitrarily different, it is possible to identify the conditions under which the two variants do not interfere with each other [19]. In the case that they do not, it is possible to produce a merged program $M$ that preserves the behaviour of both $A$ and $B$.

When applied to our problem of determining if two aspects $\mathrm{L}_a$ and $\mathrm{L}_b$ are non-intefering, the following analogies hold:

- *Base* is equivalent to the set of classes which $\mathrm{L}_a$ and $\mathrm{L}_b$ weave themselves into.
- $A$ is equivalent to *Base* with $\mathrm{L}_a$ applied.
- $B$ is equivalent to *Base* with $\mathrm{L}_b$ applied.
- $M$ is equivalent to *Base* with both $\mathrm{L}_a$ and $\mathrm{L}_b$ applied.

It follows therefore that the theory developed by Horwitz et al [19] can be applied to our problem here.

To express the condition for non-interference, we need the definition of *affected points*. The affected points $AP_{A,Base}$ of $G_A$ is defined as the subset of vertices of $G_A$ whose slices in $G_{Base}$ and $G_A$ differ:

$AP_{A,Base} = \{v \in V(G_A) | (G_{Base}/v) \neq (G_A/v)\}$

From results derived previously [19], the test for non-interference is to verify that:

$AP_{M,A} \bigcap AP_{A,Base} = \phi$ and $AP_{M,B} \bigcap AP_{B,Base} = \phi$

## B.4 Conditions for Non-interference

Given the test for non-interference, we state some sufficient conditions for two aspects to be independent and give an example to illustrate our point.

For two aspects $L_a$ and $L_b$ to be independent, the following conditions must hold:

1. No variable defined in $L_a$ is used in $L_b$.
2. Variables present in the *Base* can be used by either aspect but all definitions (assignments) of such variables must happen only in one aspect.
3. Methods introduced or overridden by $L_a$ are not called by code present in $L_b$.

Given these conditions, it would be possible to group together aspects (and therefore, features) into non-interfering groups. The problem of exhaustive testing is then reduced to testing all valid combinations in the $2^k$ possible configurations, where $k$ is the number of non-interfering groups. Unfortunately, in the worst case, $k = n$ and so there is no improvement.

```
class Base {

  int sum = 0;
  int x = 1;

  static void bar ()
  {
    foo ();
    foo ();
  }

  void foo ()
  {
    sum = sum + x;
    x = x + 1;
  }
}

aspect La {

  float Base.mean;

  after () : execution (void Base.bar ())
  {
    mean = sum / 2;
  }
}

aspect Lb {

  int Base.prod = 1;
  int Base.m = 1;

  after () : execution (void Base.foo ())
  {
    prod = prod * x;
  }

  before () : execution (void Base.bar ())
  {
    m = prod * sum;
  }
}
```

Figure B.2: *Base* with aspects $L_a$ and $L_b$

# References

[1] Apache Software Foundation. log4j. `http://jakarta.apache.org/log4j/`.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[3] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1975.

[4] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[5] Gerald Brose. JacORB: Implementation and Design of a Java ORB. In *Proc. DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, pages 143–154, September 1997.

[6] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[7] Angelo Corsaro and Corrado Santoro. A C++ Native Interface for Intrepreted JVMs. In *International Workshop on Java Technologies for Real-time and Embedded Systems*. Springer-Verlag, November 2003.

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[9] Douglas C. Schmidt. Successful Project Deployments of ACE and TAO. www.cs.wustl.edu/∼schmidt/TAO-users.html, Washington University.

[10] Erich Gamma and Kent Beck. JUnit. `www.xProgramming.com/software.htm`, 1999.

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

[12] Christopher Gill, Venkita Subramonian, Jeff Parsons, Huang-Ming Huang, Stephen Torri, Doug Niehaus, and Douglas Stuart. ORB Middleware Evolution for Networked Embedded Systems. In *Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS'03)*, Guadalajara, Mexico, January 2003.

[13] GNU is Not Unix. GCJ: The GNU Compiler for Java. `http://gcc.gnu.org/java`, 2002.

[14] Pradeep Gore, Ron K. Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and Optimizing a Scalable CORBA Notification Service. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, pages 196–204, Snowbird, Utah, June 2001. ACM SIGPLAN.

[15] DOC Group. ACE Success Stories. `http://www.cs.wustl.edu/~schmidt/ACE-users.html`.

[16] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[17] Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1998.

[18] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, 1999.

[19] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[20] Frank Hunleth. Building customizable middleware using aspect-oriented programming. Master's thesis, Washington University in Saint Louis, 2002.

[21] Frank Hunleth, Ron Cytron, and Chris Gill. Building Customizable Middleware using Aspect Oriented Programming. In *The OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, October 2001. ACM. `http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html`.

[22] Frank Hunleth and Ron K. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 38–45. ACM Press, 2002.

[23] 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*, June 1997.

[24] Ravi Pratap M., Ron K. Cytron, David Sharp, and Edward Pla. Transport layer abstractions in event channels for embedded systems. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems*, 2003.

[25] J. P. Morgenthal. Microsoft COM+ Will Challenge Application Server Market. www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[26] Object Management Group. *Notification Service Specification*. Object Management Group, OMG Document telecom/99-07-01 edition, July 1999.

[27] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, October 2000.

[28] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, December 2001.

[29] OMG. *CORBAServices: Common Object Services Specification, Revised Edition*. Object Management Group, 97-12-02 edition, December 1997.

[30] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2), March 2002.

[31] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[32] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 9(3), 1984. Also appears in Proceedings of the ACM Symposium on Practical Programming Development Environments, Pittsburgh, PA, April, 1984, and in SIGPLAN Notices, Vol. 19, No 5, May, 1984.

[33] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[34] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.

[35] SUN. Java Remote Method Invocation (RMI) Specification. java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/ rmiTOC.doc.html, 2002.

[36] The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.

[37] The Object Management Group. OMG's site for CORBA and UML Success Stories. www.corba.org/, 1999.

[38] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE(10)-4:352–357, July 1984.

[39] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4), November/December 1996.

# Vita

Ravi Pratap Maddimsetty

**Date of Birth** June 3, 1980

**Place of Birth** Adilabad, Andhra Pradesh, India

**Degrees** B.Tech. Chemical Engineering, 2001,
from Indian Institute of Technology, Madras.

**Publications** R. Maddimsetty, R. Cytron, D. Sharp and E. Pla, "Transport Layer Abstraction in Event Channels for Embedded Systems," in *Proceedings of LCTES 2003,* (San Diego, California), ACM SIGPLAN, June 2003.

December 2003